2022

Cooperative caching for object storage

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

COOPERATIVE CACHING FOR OBJECT STORAGE

by

EMINE UGUR KAYNAR TERZIOGLU

B.Sc., Bogazici University, 2011
B.Sc., Binghamton University, 2011
M.Sc., Binghamton University, 2013

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2022

Approved by

First Reader

Orran Krieger, Ph.D.
Professor of Electrical and Computer Engineering

Second Reader

Renato Mancuso, Ph.D.
Assistant Professor of Computer Science

Third Reader

Peter Desnoyers, Ph.D.
Associate Professor of Computer Science
Northeastern University

Fourth Reader

Larry Rudolph, Ph.D.
Professor of Computer Science
Massachusetts Institute of Technology

Acknowledgments

I am incredibly grateful to many people, including advisors, collaborators, friends, and family, for their generous support in my Ph.D. journey. First and foremost, I would like to thank my advisor Prof. Orran Krieger. I was fortunate to be advised by such a fantastic supervisor, and I am grateful for his tremendous support and guidance during my Ph.D. and for all the independence he gave me with my research.

I am grateful for Larry Rudolph and Peter Destroyers for all the quality they added to this dissertation. None of this work would have been possible without their support and guidance. I would like to thank my other Ph.D. committee member, Renato Mancuso, for providing valuable feedback for this dissertation.

I want to extend a special thanks to Ata Turk for his mentorship throughout my Ph.D.

Next, I would like to thank my amazing collaborators for the research described in this dissertation, as well as for other research I worked on during my Ph.D. Many thanks to Mania Abdi, Mohammad Hossein Hajkazemi, Amin Mosayyebzadeh, Matt Benjamin, Ali Maredia, and Brett Niver for all our research discussions and for helping me to build complex systems.

I am thankful to the team of engineers at RedHat, especially Hugh Brock, Heidi Dempsey, Rick Sussman, Ben England, and Mark Kogan, for supporting my research. I would also like to thank the Mass Open Cloud team, especially Naved Ansari, Jennifer Stacy, and Michael Daitzmen, for their constant help and support.

Next, I would like to thank my labmates and friends in Boston for making the entire Ph.D. journey experience so enjoyable and kept me going during my lowest moments: Burcu Yigit, Ulya Aviral, Nabeel Akhtar, Gizem Umur, Ebru Ercan, Funda Sarican, Humann Hodjatzadeh, Sahile Tikale, Ali Raza, Parul Sohal, Han Dong, Tommy Unger, Apoorve Mohan, Ali Raza, Golsana Ghaemi, James Cadden, Eric

Munson, Arlo Albelli, Sanskriti Sharma, and Yara Awad. Their friendship, encouragement, and support have made this dissertation possible.

Finally, I would like to thank mom, dad, and my brother for always being there for me and proudly supporting me through all my endeavors. Last but not least, I want to express my deep gratitude to Gokhan for his unrelenting patience, endless support, and encouragement during my Ph.D. journey.

COOPERATIVE CACHING FOR OBJECT STORAGE

EMINE UGUR KAYNAR TERZIOGLU

Boston University, Graduate School of Arts and Sciences, 2022

Major Professor: Orran Krieger, Ph.D.
Professor of Electrical and Computer Engineering

ABSTRACT

Data is increasingly stored in data lakes, vast immutable object stores that can be accessed from anywhere in the data center. By providing low cost and scalable storage, today immutable object-storage based data lakes are used by a wide range of applications with diverse access patterns. Unfortunately, performance can suffer for applications that do not match the access patterns for which the data lake was designed. Moreover, in many of today's (non-hyperscale) data centers, limited bisectional bandwidth will limit data lake performance. Today many computer clusters integrate caches both to address the mismatch between application performance requirements and the capabilities of the shared data lake, and to reduce the demand on the data center network. However, per-cluster caching; i) means the expensive cache resources cannot be shifted between clusters based on demand, ii) makes sharing expensive because data accessed by multiple clusters is independently cached by each of them, and iii) makes it difficult for clusters to grow and shrink if their servers are being used to cache storage.

In this dissertation, we present two novel data-center wide *cooperative cache* architectures, *Datacenter-Data-Delivery Network (D3N)* and *Directory-Based Datacenter-Data-Delivery Network (D4N)* that are designed to be part of the data lake itself

rather than part of the computer clusters that use it. D3N and D4N distribute caches across the data center to enable data sharing and elasticity of cache resources where requests are transparently directed to nearby cache nodes. They dynamically adapt to changes in access patterns and accelerate workloads while providing the same consistency, trust, availability, and resilience guarantees as the underlying data lake. We find that exploiting the immutability of object stores significantly reduces the complexity, and provides opportunities for cache management strategies that were not feasible for previous cooperative cache systems for file or block-based storage.

D3N is a multi-layer cooperative cache that targets workloads with large read-only datasets like big data analytics. It is designed to be easily integrated into existing data lakes with only limited support for write caching of intermediate data, and avoiding any global state by, for example, using consistent hashing for locating blocks and making all caching decisions based purely on local information. Our prototype is performant enough to fully exploit the (5 GB/s read) SSDs and (40, Gbit/s) NICs in our system and improve the runtime of realistic workloads by up to 3×. The simplicity of D3N has enabled us, in collaboration with industry partners, to *upstream* the two-layer version of D3N into the existing code base of the Ceph object store as a new experimental feature, making it available to the many data lakes around the world based on Ceph.

D4N is a directory-based cooperative cache that provides a reliable write tier and a distributed directory that maintains a global state. It explores the use of global state to implement more sophisticated cache management policies and enables application-specific tuning of caching policies to support a wider range of applications than D3N. In contrast to previous cache systems that implement their own mechanism for maintaining dirty data redundantly, D4N re-uses the existing data lake (Ceph) software for implementing a write tier and exploits the semantics of immutable objects

to move aged objects to the shared data lake. This design greatly reduces the barrier to adoption and enables D4N to take advantage of sophisticated data lake features such as erasure coding. We demonstrate that D4N is performant enough to saturate the bandwidth of the SSDs, and it automatically adapts replication to the working set of the demands and outperforms the state of art cluster cache Alluxio [25]. While it will be substantially more complicated to integrate the D4N prototype into production quality code that can be adopted by the community, these results are compelling enough that our partners are starting that effort.

D3N and D4N demonstrate that cooperative caching techniques, originally designed for file systems, can be employed to integrate caching into today's immutable object-based data lakes. We find that the properties of immutable object storage greatly simplify the adoption of these techniques, and enable integration of caching in a fashion that enables re-use of existing battle tested software; greatly reducing the barrier of adoption. In integrating the caching in the data lake, and not the compute cluster, this research opens the door to efficient data center wide sharing of data and resources.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today, data and its analysis are at the center of many businesses. The amount of data created every year continues to grow at exponential rates. Organizations are collecting all data that could be of future value and making increasing use of today's cloud data centers' rich services and elastic capabilities to process and glean critical insights from it. The growth of data and modern large-scale distributed applications (e.g., Facebook app, Netflix app, Spark, Tensorflow) have radically changed the way data is stored and processed. Today many data centers deploy datalakes, low-cost object-storage repositories that can store vast volumes of data [118, 27].

In most data lake realizations, the fundamental concept of the storage system is no longer a file or a block but an immutable object. For scalability, object stores access data using RESTful interfaces (e.g., the S3 interface [27]), discard many of the semantics offered by traditional files systems (e.g., POSIX semantics), and explicitly defer some of them to the application. For instance, immutable objects eliminate the need for journaling, the S3 interface delegates handling anticipated temporary failures to the application, and allows data to be aggressively cached. S3's relaxed consistency model allows objects to be lazily invalidated if a new version is written, resulting in performance improvements for applications. Finally, the composability of the S3 interface allows services to be simply layered on top of each other.

Originally, object storage systems were primarily used for storing large unstructured datasets, backups, and archives. Over the last decade, object stores' role has

expanded beyond their earlier use cases and object stores are increasingly being used for applications that have traditionally required file or block-based storage systems. The simplified semantics enable low cost highly scalable implementations of object storage over commodity servers that make them an ideal alternative to distributed file systems. Object storage is today used by a wide range of applications from data analytics [125, 95], to deep learning [72, 92], to serverless functions [105, 117, 93], file systems [75, 86], and even virtual block storage [45, 56].

Different workloads can have dramatically different access patterns, for example, ranging from write dominated operations on objects of a few kilobytes to read dominated operations on multi gigabyte objects [106, 78, 94]. This diversity has led to multiple implementations of object storage optimized for different object types and access patterns [82, 39, 89]. This means that, unless a data center wants to incur the high cost to operate multiple data lakes[89], the performance of applications with access patterns different from those the data lake has been designed for will suffer. Also, in many cases, the capabilities of shared data lakes may not meet the requirement of applications. Many data centers do not have a full-bisection bandwidth. Data access may be significantly constrained by over-subscribed network links and the limitations in performances of slow high-capacity drivers.

In many data centers, data lakes and the data in them are widely shared across clusters or frameworks deployed by independent entities. In industry, clusters may be owned or deployed by different groups or business units of a corporation. For example, within Two Sigma [107], a financial hedge fund (also an industrial partner of our group), different groups create individual clusters and deploy different analytical frameworks, and a wide number of these compute clusters repeatedly access the same datasets (e.g.,newly added market data) from a shared data lake. The reasons for multiple clusters include security, organizational structure, regulatory issues, or

preference. Similarly, in academic data centers sharing of data is very common. Researchers from different institutions collaborate or work on the same scientific data sets (such as ImageNet [63], Harvard Dataverse [15]), but from their own institutional or lab clusters. For instance, in our academic data center the Massachusetts Green High Performance Computing Center (MGHPCC) [13] dozens of compute clusters (totalling over 200K cores) access a shared 50PB data lake.

Caching systems are widely used to cache objects widely shared across users, and address the mismatch between application performance requirements and the capabilities of a shared data lake. The most recent caching works are 1) integrated into the framework/cluster (e.g., Pacman [28], Alluxio [25], Quiver [72], MRD [91], Hoard [92], or 2)integrated into the storage servers of data lake [118, 58, 82, 39, 90, 95, 106, 78, 94, 121]. The framework/cluster caching moves frequently accessed data into (logically) nearby high speed storage, and enables the systems to be customized for a particular framework or class of application to improve performance. However, expensive cache resources are dedicated to separate frameworks/clusters; therefore, they limit data sharing across frameworks/clusters and make it challenging to shift the costly cache resources to where they are most needed. While storage side caches reduce demand on slow disks, they are near the data lake, not the cluster, resulting in heavy use of bi-sectional bandwidth. Moreover, these caches are usually general-purpose caches and they are workload unaware.

Many previous works have studied "cooperative caching" [47, 104, 25, 30] for file systems, exploring how to create larger shared aggregated caches using distributed caches. These works implement cooperative caches by using all the client buffer caches as an extension of the file system to improve the performance. They enable data sharing, cache data near the client, avoid waste of idle cache resources, and reduce the slow disk load. We borrow heavily from the rich research in cooperative

caching systems and re-evaluate the design of "cooperative caching" for immutable object-based data lakes.

This dissertation explores *how **cooperative cache** architecture can be integrated into today's immutable object storage based data lakes, enabling data lakes to expand across the data center.* We carefully design two caching systems that distribute caches across the data center, a trusted extension of the data lake rather than part of the clusters that access the data lake. We show that this enables data to be efficiently accessed anywhere in the data center, preventing expensive cache resources from being siloed for a particular cluster/framework.

We explore two novel data center scale *cooperative cache* architectures for a data lake. In particular, we investigate the consistency, availability, and resilience requirements of each approach, how to enable data sharing and use the composable S3 interface to reduce the complexity of efficient caching and replication, and allow services to be simply layered on top of each other, and the protection mechanisms needed in a multi-tenant environment. We present the challenges and design tradeoffs of these two architectures and study their effectiveness on improving the performance of applications. We demonstrate that unique features of object stores (e.g, immutability, relaxed consistency) reduce the complexity of efficient caching and provide opportunities for investigating cache management and data placement strategies that have not been feasible in the past for file systems and block stores. Overall, in this dissertation, we present the design and implementation of **Datacenter-Data-Delivery Network (D3N)** and **Directory-Based Datacenter-Data-Delivery Network (D4N)**. Both cache designs are developed and implemented by modifying Ceph [118], an object-based storage system commonly used to implement data lakes.

Our first caching system, D3N, creates a multi-layer cooperative cache that mitigates network imbalances by caching data on the access side of each layer of hierarchi-

cal network topology, adaptively adjusting the cache capacity of each layer based on observed workload patterns and network congestion. A fundamental goal of D3N is to allow simplified integration into existing data lakes to enable caching to be transparently introduced into data centers and easily upstream code contribution into the existing code base of an open-source data lake solution (e.g. Ceph project [118]). D3N avoids the use of global state; instead it uses the local cache information for caching policies and consistent hashing [68] for distributing the data blocks across layers. We develop an algorithm that uses only the local information to partition the cache capacity between local vs global requests to adjust to changes in workload and network hotspots. D3N is focused on a read cache for analytical clusters [125, 54, 12], and it supports non-durable write caching only for reproducible intermediate data by exploiting the object stores' immutability. We showed that D3N's implementation is highly efficient, able to saturate the dual NVMe SSDs and the 40 Gbit NIC; result in almost 5× performance improvement over default RGW, and a 3x reduction in runtime for realistic workloads(6.7). The adaptive cache partitioning algorithm resulting in substantial gains(up to 30%) over static allocation(6.4.1).

Our second cooperative cache system, D4N, implements a global reliable directory that enables a richer set of caching policies, supports a durable distributed write-cache, and targets a broader set of use cases. The global state of the cooperative cache, maintained in a distributed directory, is used to enable local cache policies to employ the aggregate caching resources efficiently, e.g., replicating data widely or using the entire aggregate capacity to hold large working sets.

D3N focused on analytical workloads that were the dominant use of object-based data lakes when we started this work. D4N is intended to be more general, enabling a greater degree of application-specific specializations that make it appropriate for a broader set of use cases. D4N implements two such specializations: small object coa-

lescing and packing, and smart prefetching and caching for DAG-based applications. We demonstrate that D4N can reduce cumulative job run time by a large amount (e.g. 30% (6.5)), and reduce demand on the network and the backend data lake. Its analytic cluster-specific specialization improves the run time of real-word workloads by 20% (6.7), and the small object packing specialization reduces data lake workload by 400% over the case of no write tier and 200% over the case without object coalescing. Like D3N, D4N also does not require changes to client application, however the shared directory, write tiers, and maintaining global state increase the complexity of the system.

The D3N design is based purely on local information, uses consistent hashing for block lookup, stores all durable states in the underlying data lake, and can be easily integrated into an existing data lake. D3N has been upstreamed to the Ceph community [19] and is available as an experimental feature in Ceph today, enabling its use by the hundreds of data lakes based on Ceph. The D4N design, while more powerful, entails more complexity, changes operational assumptions of the data lake, and relies for correctness on state stored outside of the underlying data lake. We have succeeded in integrating the D4N into Ceph to show its value. While we expect our changes to be eventually integrated by the Ceph community, the process will be much more complicated.

This dissertation makes the following contributions:

- We show that the old idea of a cooperative cache can be adapted to the needs of today's massive scale data centers and bandwidth-limited data lakes. We explore two novel data center scale distributed cooperative cache architectures, where each caching system is a transparent extension to the data lake itself, dynamically adapting to different access patterns and providing acceleration to workloads.

- For D3N, we show that by carefully architecturing the system, it is possible to design a caching system that could be integrated into a real production data lake in a practical fashion and offers great value without requiring major modifications to a complex storage system. We demonstrate that the simplicity of design allows our changes to be easily upstreamed and it can be used in hundreds of data lakes based on Ceph. D3N work shows it is possible to improve the performance of analytical workloads significantly, without having a global state and using deterministic data placement. Finally, we present a novel local algorithm to enable a single cache to be efficiently shared by multiple levels, where cache space is shifted between the layers based on demand, and evaluate how it dynamically adapts to different access patterns and responds to network contention.

- For D4N, we show that D3N work can be extended further to support a wide range of applications. The advantage of D4N is to have a highly available shared directory, which maintains the global state and avoids meta-data accesses to the data lake, allowing us to separate caching policy from the caching mechanism and enabling more flexible object placement strategies. The durable write tier enable caching of newly generated data redundantly. With these advantages, D4N supports application-specific specializations at the caching layer that enable workload-specific tuning of caching policies and improves the performance gain for diverse workloads.

The rest of this dissertation is organized as follows. Chapter 2 provides relevant background information and motivation. Chapter 3 discusses related works. Chapter 4 presents D3N and D4N cache architectures, and chapter 5 shows the implementation of both systems. Chapter 6 presents our experimental evaluation of both architecture and finally, chapter 7 concludes and discusses potential future work.

Chapter 2

Background and Motivation

This chapter provides background information and motivation for cooperative caching at the data center scale. Section 2.1 describes object storage systems, and section 2.2 describes key properties of modern data centers and discusses the causes of the network bottlenecks. Section 2.3 discusses workloads' characteristics and their requirements from a data lake. Section 2.4 numerically demonstrates the value of cooperative caching to motivate D3N and D4N architectures.

2.1 Immutable Object Storage

Many data centers deploy cost-effective, centralized storage repositories, called **data lakes**, to store and share vast amount of data. In an enterprise this may be akin to the classic data warehouse; in a scientific environment, a repository for shared datasets. As seen in Figure 2·1, data lakes are often deployed as a separate cluster that typically use many standard commodity servers with locally attached disks. Such data lakes are typically implemented by object stores, such as Ceph [118], which support high capacity and economical storage for unstructured data with fine-grained access control to provide a varied level of access to datasets owned by different entities. Features like simplified management, ease of use, and high data durability led to the wide use of object stores in today's data center.

Figure 2·2 shows the ecosystem of an object-based data lake with object storage devices (OSD). Object stores store the data on OSDs consisting of an active daemon

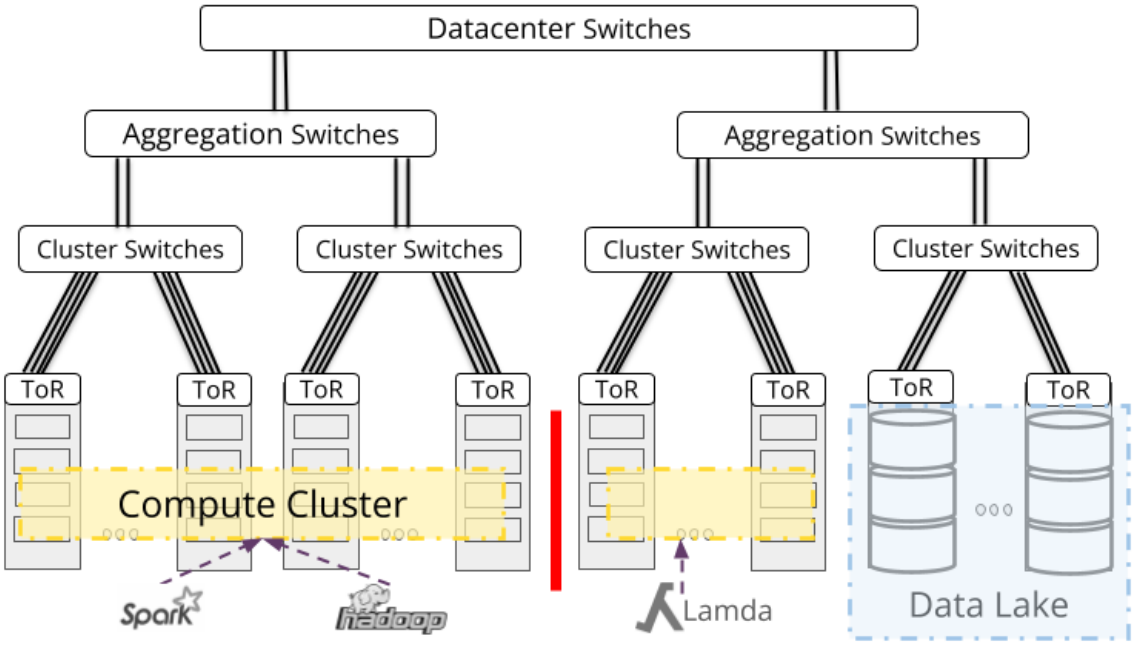


Figure 2·1: Modern Data Center Ecosystem. Many modern data centers create data lakes as a separate cluster which are typically implemented as object stores. Data center topology contains multiple clusters connected via an over-subscribed data center network. Every cluster in the data center shares the data stored in data lake.

and a multiple disks. OSDs are responsible for storage management functions such as replication, synchronization, fault tolerance, and each OSD manages its own storage space.

Object stores have different characteristics than traditional distributed file systems [58] or block storage [40], where these differences are crucial for accelerating the distributed storage stack. Object stores are immutable, data can not be modified after being created. Unlike traditional distributed file systems in which data is stored in a hierarchical model (files within folders), objects are stored in a flat namespace in object storage. The data is stored in large blocks (typically in MBs). To support scalability, object stores discard many of the semantics of traditional file systems (e.g., POSIX consistency).

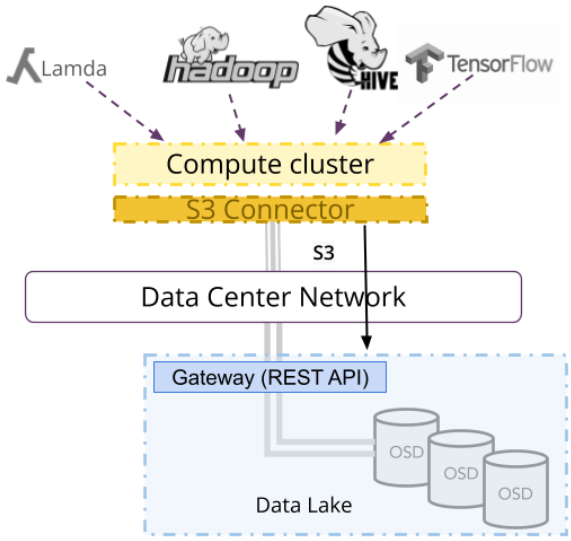


Figure 2·2: Immutable Object Storage with a Gateway

Object operations are **atomic**, where the object is either written entirely or not written at all, and objects are not visible until the write is complete. If there is a failure during a read or write operation the entire operation fails, and applications have to restart the operation. The atomic operations ensure that the object never ends up in an inconsistent state where the old data is not lost before new data has been stored. To increase resiliency to network errors during a (large) object writes, object stores limit the size of the object in a single write operation (e.g., 5GB in S3, Ceph). They provide a mechanism called "multipart-upload," where a contiguous portion of the object's data is uploaded independently and in any order. If the writing of any part fails, then the application rewrites that part without affecting the remaining parts. Once all parts are written, the object store assembles written parts and presents the data as a single object. Each object is identified in the system by a globally unique *object id/version number* (instead of a file name and file path), which is used for locating and accessing the object. If an object is deleted and a new object is created with the same name, it will have a new unique object id/version number,

and the previous version of the object is removed later on by garbage collection.

The immutability greatly reduces the complexity of efficient caching and replication. It enables adoptation of **erasure coding**, a space-efficient and highly fault-tolerant redundancy scheme for storage systems. For (n,k) erasure codes, each object is divided into n equal-sized data chunks and k additional parity chunks, calculated from the n data chunks. Any n out of n+k chunks can be used to reconstruct the original object. Updates in erasure coding are expensive because they require reading all chunks, updating the data, recomputing the parity chunks, and writing the updated data and parity chunks back to the disks. This overhead does not occur with immutable object storage because immutable objects can not be modified. In addition, erasure codes may have a high disk seek and rotational delay for reads. In object storage systems, the object size is large enough to largely amortize the cost of seeks and rotational delays of erasure coding; thus with erasure coding, object stores can provide much higher data durability than file systems or block storage.

As seen in figure 2·2, many object storage solutions provide a **gateway** to allow applications to access the object storage servers. These gateways provide an S3 Compatibility REST API, allowing access to objects and buckets using standard HTTP commands such as GET, PUT and DELETE. Any application that supports HTTP can directly access the object store gateway, allowing the service to be used by many different applications and platforms.

Due to the success and widespread use of Amazon's S3 object storage, S3 API, has become the dominant API for accessing objects. However, there are other interfaces (such as Swift [4]) as well. In the rest of this dissertation, we will focus on S3. Today almost every object storage [118, 78, 36] in the market supports the S3 API in some form or other. Like data lakes, many frameworks also support direct access to the object store using the S3 interface, via libraries or connectors such as S3A [102]for

Spark, which provides HDFS-compatible access to S3 compatible objects.

The S3 interface, in contrast to general-purpose file systems, is **composable**, where new functionalities can be implemented as an S3 service built on top of other S3 services. For example, the composability of the S3 interface allows D4N to deploy write tiers over an object storage system.

Today, due to their desirable features (e.g., scalability, cost, high availability) object stores are increasingly being used for applications that have traditionally required file systems or block storage such as data analytics [125, 95], deep learning [72, 92], serverless functions [105, 117, 93], and file systems [75, 86]. Different workloads can have dramatically different requirements from the data lake such as high throughput to support millions of concurrent requests, various redundancy levels for different applications, providing support large and small IO, or efficiently sharing storage resources between multiple users. This diversity has led to multiple implementations of object storage optimized for different object types and access patterns [82, 39, 89, 31]

It should be noted that object stores have many advantages over a file or block storage system: they provide a simple storage interface with immutable objects, avoiding the complex semantics and consistency requirements of general purpose file systems, and they provide opportunities for investigating strategies that have not been feasible in the past for block storage. Their simple consistency model significantly simplifies caching and tiering to accelerate the performance of applications. For example, there is no need to invalidate the read cache since simply changing object id/version will cause cached read data to be ignored. Larger-granularity object access (in MB) allows investigating cache management approaches that were not feasible for block storage. The expense of more complex algorithms (e.g., machine learning) can be amortized over longer transfer time, and more information can be tracked per access unit without loss of efficiency. Moreover, the S3 interface enables composable services to be

layered on top of each other, allowing object stores to provide rich functionality and support more applications.

In this dissertation, we argue that the rich functionalities to efficiently support a wide range of applications should be layered on top of the underlying immutable data lake while enabling a single data lake to be used for long term durability. This way, the underlying object-based data lakes can be optimized for storage efficiency (e.g., supporting large immutable objects that can take advantage of erasure coding and storage technologies like SMR). This, in the future, may lead to the re-invention of the storage stack in data centers, basing it on immutable write-once objects stores and seeking mechanisms for efficiently implementing mutability, resiliency, and consistency at the most appropriate layers.

Throughout this dissertation, we use the following three terms from the object storage terminology; i) **object**, ii) **bucket** and iii) **block**. In object storage, the file is called **object** (also called s3-object in S3 API terminology). Objects are stored in a container called **bucket**(also called s3-bucket in S3 API terminology), where buckets can contain unlimited number of objects, and **block** is fixed-size chunk of an object (e.g., 4MB in Ceph).

2.2 Modern Data Centers

In figure 2·1 illustrates a very common data center or co-location facility for academic and enterprise environments. We first describe key elements of the data center in detail (Section 2.2.1), and then describe why network bandwidth is limited in many data centers.

2.2.1 Key elements

As seen in Figure 2·1, data centers typically have some form of hierarchical network, where TOR switches allow computers on the same rack to communicate at their

maximum speed, some number of racks may be part of a tightly coupled compute clusters, and bandwidth becomes constrained as communication has to traverse higher levels in the network. In many data centers, compute clusters are independent units owned and managed by different entities: a single organization or different units of an organization (private data centers) or multiple organizations (co-location facilities).

The massive demand for data today, has resulted in many data centers deploying large object-storage based data lakes, typically deployed as its own cluster (light blue), that can inexpensively store huge amounts of data and can be used by software running anywhere in the data center. Many data centers usually deploy a single data lake, which is shared among the various entities involved with the data center. In general the data lake is managed by the data center or co-location facility provider.

Entities involved with the data center create compute clusters and run different computation frameworks like Spark [125] and Tensorflow [12]. These clusters may be comprised of a single compute cluster (as shown in Figure 2·1), multiple compute clusters, or portions of compute clusters. These clusters can be launched on-demand [59] or may be long-lived. They can run on virtual machines [37], containers [110], serverless platforms [3] or baremetal nodes [80].

2.2.2 Reasons for Limited Network Bandwidth

The performance of a data lake is limited by its underlying network. In these environments, a critical bottleneck can be getting the data from a large shared data lake due to limited network bandwidth in data centers due to the **network oversubscription**.

Network oversubscription means that the maximum amount of traffic that can be sent to a given switch is greater than that which can be routed through it. Most data center networks are constructed so that higher layers of the network topology are more heavily oversubscribed compared to lower layers. In many data centers, servers in the same rack are connected at full bandwidth via one or two switches. There is a

modest degree of oversubscription in the network within a compute cluster (e.g. 2x) but a very large degree of over-subscription (relative to total server NIC bandwidth) in the data center network that connects these compute and data lake clusters. This corresponds to the intuition that servers that are topologically closer to one another will be more likely to communicate with one another (e.g., servers within a rack or servers within a compute cluster).

Although previous research has assumed full bi-sectional bandwidth [85, 118]; in practice cost prevents their widespread deployment. Even for hyperscalars, rack uplinks - i.e. the connections between top-of-rack (ToR) switches and the rest of the data center - are over-subscribed, with e.g., a ratio of 3:1 in Google's Jupiter [109] interconnect. This is a natural consequence of today's technology, where servers and cost-effective switches often have interfaces of the same or nearly the same speed (e.g., 100 Gbit/s uplinks vs. dual 40 Gbit/s NICs). In this environment, the cost, complexity, and even physical volume of cabling required for full bisection bandwidth (i.e. 1 rack uplink per server NIC) puts it out of reach of all but the most extreme applications.

One reason for the high network oversubscription in many data centers is that the portions of the data center are upgraded independently of other portions. Such upgrades may be coordinated to preserve balance (or desired oversubscription) within the upgraded portion, however hardware clusters may be deployed at different times, with different network technologies, and different clusters and network links may be owned or deployed by different entities: sub-units of a corporation, research groups in a university, or entire companies in a co-location facility. For example, in the MGHPCC [13] data center, hardware clusters of some research groups can have up to 200Gbps links to each server, with massive intra-cluster bandwidth, yet these clusters may be connected to an institutional network at 40Gbps, which in turn is connected

to the cross-data-center switch at 40Gbps or 100Gbps.

2.3 Understanding Workload and Their Requirements

In this section, we describe key properties of modern data centers' workload, their needs, and the implications for cache system design. Our design is motivated by key properties (i) data sharing is common, (ii) locality is important, (iii) storage demand is highly variable, and (iv) workloads have diverse requirements. We discuss these properties and the four requirements (RQ1, RQ2, RQ3, RQ4) they imply for efficient caching of storage.

2.3.1 Data sharing is common

Previous work has shown that applications running on different frameworks often share data [20, 72, 33]. For example, Abdi et al. [20] analyze a trace from a production cluster with multiple analytic frameworks, and demonstrate that 36% of Spark jobs share objects (42% of total data) with Hive, while all tables accessed by Oozie were shared with Hive. A similar observation is reported by the authors of Quiver [72], a cache for deep learning training (DLT) jobs, which observe that "A few popular input datasets are used across numerous DLT jobs and across several model architectures".

Even though clusters can support multiple frameworks, much of this sharing happens between clusters. In many academic institutions, researchers often work on different institutional clusters, but share scientific datasets. For instance, the machine learning community train their models on common datasets such as ImageNet [63] or Youtube-8M [21]. As another example, in our datacenter MGHPCC [13], Harvard Dataverse Repository [15] stores scientific datasets which are accessed by researchers from different groups. These reuse and data sharing patterns also exist in industry. For instance, Two Sigma [107] has a various number of compute clusters, divided according to organizational structure or analytic platform type, each access data from

a shared data lake. As soon as a new dataset appears (in this case, recent market data), jobs accessing that data will be launched on many of these clusters.

(RQ1) Support sharing: If data is cached as a result of any workload, other workloads should be able to take advantage of that cached state to avoid both demand on the data lake and expensive cache storage.

2.3.2 Locality is important

The compute-storage disaggregation and oversubscribed networks in data centers reduces the data locality for applications. As we described in section 2.2.2, in many data centers, the bandwidth internal to racks and between some racks (e.g., of the same institution) is much larger than the bi-sectional bandwidth of the data center. Accessing data stored from a remote data lake creates an additional network load, thus the data lake network can become a bottleneck due to over-subscription of data center network.

(RQ2) Preserve locality: To reduce use of limited bi-sectional bandwidth, data should be cached near where it is being accessed.

2.3.3 Demand is variable

The success of hardware virtualization technologies and allocation and provision technologies for bare-metal clouds enable high elasticity of compute resources, where the size of a compute cluster can grow and shrink in a few minutes. Compute clusters hosted in a data center serve a wide range of applications/frameworks who may change their behavior dramatically. Applications can be deployed on many nodes using elastic compute resources and executed in a distributed way. Not only does its storage demand change as a cluster grows or shrinks, but the location of the demand may change, e.g., as an elastic cluster expands beyond its original location. As a result, demand for storage can change rapidly. This is visible in IBM's COS traces [49, 61],

where one large set of the tenants have dramatically different rates of access at different times, while another set accesses the storage predictably in bursts (e.g., every day at midnight), and yet another only accesses data for a few days. This variability is made more complex by cluster elasticity, available in all public ([34, 51, 35]) and many large-scale private clouds.

(RQ3) Enable Elasticity: To respond to changing demand, cache resources should not be dedicated to a particular framework or cluster, rather a cache should be able to dynamically shift expensive cache resources to the source (workload and location) that most need them.

2.3.4 Demands are diverse

Today, object storage is used by a wide range of applications.Different workloads may have very different demands on storage. This was shown in previous research, that observed diverse data access sizes, frequency, operation types, and durability requirements [64, 28, 70, 99]. We see this also in the IIBM cloud based object store service (IBM COS) [49] traces. Here we find that while 17 traces include only read requests, two traces have 83% write accesses; while some traces are dominated by requests for objects under 1KB, others access objects greater than 1GB, and that the rate of object overwrites in a trace varies from 0.1% to almost 99%.

(RQ4) Support specialization: To support wide range of applications caching system should enable workload specific tuning of caching policies. It should be possible to specialize cache management, e.g., cache bypass, pinning, eviction, write-back, and prefetching.

2.4 Value of Cooperative Caching

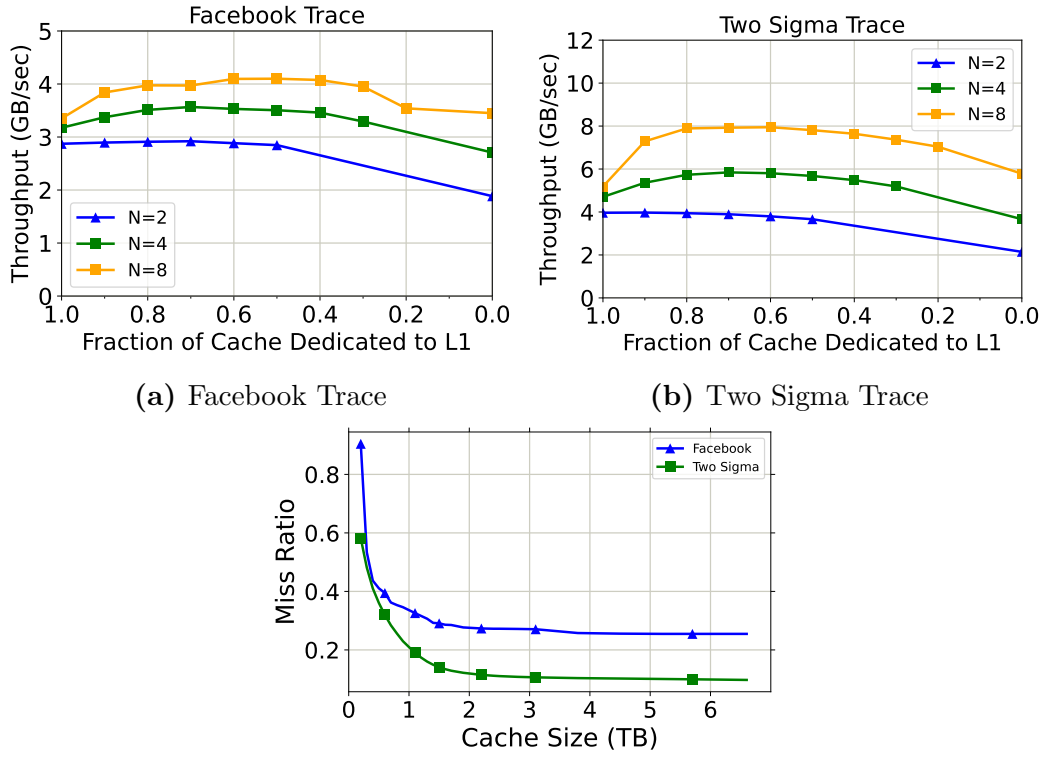
To motivate the cooperative caching of D3N and D4N, we developed a simple numerical model parameterized from micro-benchmarks (Section 6.3.1) and real-world traces (Section 6.1). In our model we compare three different caching approaches: 1)*Pure local caching (L_1)*, where there is no cache cooperation and the entire cache is dedicated to local requests, 2)*Multi-level cooperative caching (L_1 + L_2)*, where each cache manages a portion of its local cache "greedily" (L_1), and the remaining portion "globally"(L_2). 3)*Distributed caching (L_2)*, where the entire cache space is managed globally without any replication.

Our model shows that cooperative caching (L_1 + L_2) that caches both local and remote requests has a significant advantage over single layer caches such as pure L_1 or pure L_2 when the data center has a hierarchical network topology with oversubscription.

We modeled a cooperative cache with two level, assuming a cache node with 1 TB of SSD on each rack that is shared between L_1 cache dedicated to caching rack-local accesses and L_2 cache dedicated to caching remote accesses, a 10Gbit switch connecting between 2 and 16 racks, and a 10Gbit link to the datalake. We use miss rate curves (MRCs) (Figure 2·3c) calculated from Facebook and Two Sigma traces (see Section 6.2), then estimate the hit rate at each level of cache, and from that numerically derive the aggregate storage bandwidth seen by clients. The model is:

$$\begin{aligned} r &= \frac{1}{\frac{1-m1}{r_{L1}} + \frac{1-m1-m2}{r_{L2}} + \frac{m2}{r_{DL}}} \\ m1 &= MRC(F_{L1} \cdot C) \\ m2 &= MRC((1 - F_{L1})C \cdot N) \end{aligned} \tag{2.1}$$

where r_{L1}, r_{L2}, r_{DL} are L_1 hit, L_2 hit and L_2 miss bandwidths, F_{L1} is the fraction of cache devoted to L_1, C and N are the capacity of a single cache and number of



(a) Facebook Trace

(b) Two Sigma Trace

(c) Miss ratio curves

Figure 2·3: a)MRCs of Facebook and Two Sigma trace, b - c) Modeled throughput of storage cluster for varied numbers of caching nodes (N), ratio of L_1 and L_2 cache. 1 TB cache per node, 40 Gbit cache / 20 Gbit for storage traffic inter-rack / 10 Gbit to datalake.

caches, and MRC is the miss ratio curve for the workload.

Figure 2·3 shows the storage throughput for a two level cache for 2, 4, and 8 cache nodes, as we vary the L_1 fraction of the cache from 100% (fully-local) to a minimum of $\frac{1}{N}$ (due to unified caching). The leftmost point on each plot (i.e. 100% L1) estimates the performance of a pure-local caches, and the rightmost point on each plot (i.e. 100% L2) estimates the performance of a distributed cache without replication. We set r_{L1} r_{L2} and r_{DL} assuming a datacenter with 40 Gbit ToR switches, 40 servers per rack and 100 Gbit uplinks, resulting in a 16:1 oversubscription. [1]

[1] We set r_{L1} to 40 Gbit, r_{L2} to 20 Gbit and r_{DL} to 10 Gbit/s; with a 16:1 oversubscription it seems optimistic to assume 20 Gbit of the 100 Gbit uplink is available for storage traffic, and 10 Gbit/s is

Our simple numeric model underestimates the value of the two-level cache in that it assumes that the bandwidth between L_2 caches is not contended, and that there is no locality in requests from the same rack or cluster. Even with these pessimistic assumptions we see that the multi-level approach offers better performance than either a pure L_1 or pure L_2 approach for 4 cache servers or more. The industry trace, with its 90% reuse rate at eight cache nodes shows a 40% improvement over a pure L_1 cache and a 25% improvement over a pure L_2 cache.

roughly the bandwidth we have seen from our 90 spindle Ceph cluster.

Chapter 3

Related Work

This chapter provides the larger context for this work and gives an overview of the caching system research related to our works. Caching is a well-studied problem in various computer systems ranging from multi-processors to web systems. In this chapter, in particular, we focus on the caching systems designed to improve the performance of data lakes in data centers and the original cooperating caching systems for file systems.

We group caching systems for data lake, based on how they are integrated into the existing storage stack, into two categories; i)*storage level caching* that uses the caching resources distributed to the nodes that are providing the storage service, and i)*client level caching* that is either deployed within the framework or compute cluster. In section 3.1.1 and section 3.1.2, we describe work done in each of these areas, and in section 3.1.3 we discuss how they relate to the key requirements (RQ1-RQ4) described in section 2.3. Finally, in section 3.2, we discuss the rich cooperating caching research for early file systems, which deeply influenced the design and implementation of D3N and D4N.

3.1 Data Lake Caching

3.1.1 Storage level caching

With the reduced cost of higher speed storage devices such as solid-state drives (SSDs) [53], they are widely used at various levels in data centers, and are tightly

integrated into the storage-system stack. Many storage vendors provide either a transparent flash cache layer(e.g., IBM XIV SSD cache [62]) or a flash tier(e.g., IBM Easy Tier [60], EMC FAST [48]) within the high-end storage server to reduce disk latency. Data lake implementations generally [118, 58, 82, 39, 90, 95, 106, 78, 94] cache data at the nodes where the data is stored, with the fundamental goal of reducing demand on the slow disks; however, they don't prevent generation of excessive traffic to the data lake.

An alternative solution, tiering, is implemented by many data centers [57, 77, 95, 41, 6, 24]. The tiering approach is popular because it can utilize multiple types of storage media (e.g., memory, SSD, Disk) with different performance and capacity characteristics to create tiers for accommodating various data types, thus reducing the storage cost significantly [69]. The major drawback of this approach comes when migrating the data and its replicas between tiers, which usually adds complexity to the design and implementation, and may degrade the performance of applications. For instance, the Ceph team reported that a tiering approach reduces most workloads' performance [41].

These solutions are generally workload unaware, and caches/tiers are automatically used by all the clusters and frameworks accessing data.

3.1.2 Client level caching

Framework caching

We use the term framework caching to refer to caching done either at the level of framework that supports applications of a particular class or caching integrated into a specific application. Many frameworks, from serverless to data analytics to deep learning, have implemented framework and workload specific storage caches [91, 124, 72, 92, 116, 99, 117, 79]. In addition, caches at this level can be integrated into the general resource management of the framework. For example, Faa$T [99], a

storage cache for serverless functions, employs the memory used for the VM running a serverless function to cache the storage accessed by the function.

Framework cache solutions can adopt highly specialized cache management policies matching the applications they support by exploiting the application-level semantics, and can provide maximum performance gains for applications. These policies rely on either the hints from applications or highly repetitive and predictable data access patterns of workloads. For example, Faa$T [99] implicitly prefetches storage by loading the cache when starting a function. MRD [91], a caching system for analytic platforms, exploits information about the schedule of jobs to prefetch data before it is needed. MRD [91], extracts rich information from the analytics platforms in the form of Directed Acyclic Graphs (DAG), and utilizes DAGs information to optimize both eviction and prefetching of data to improve cache management in Spark. In a very different fashion, Quiver [72], a cache for deep learning, takes advantage of the phenomena that learning algorithms can handle the data in random order to serve the data in the order most efficient for the cache.

Cluster Caching

A number of systems have been developed that cache data using cluster level resources (e.g., memory, SSD, local disk) to avoid access to shared data lakes [25, 28, 96, 16, 64]. Cluster caching solutions are usually co-located with the computation, and use the compute nodes' resources (e.g., memory, SSD) to form a cache layer, and therefore they provide high locality. Organizations do not have to co-locate caches with computation and can dedicate separate resources for caching at the cluster level to allow elasticity for compute clusters; however, this approach is not very common and may decrease the data locality. Multiple frameworks/applications running in the same cluster can share the cluster-level caches, therefore specialization based on application semantics or their access patterns is limited compared to framework

caches.

Much of the recent work has focused on data analytics [28, 25], coordinating caching across nodes to optimize these workloads. For instance, Pacman [28] is a distributed in-memory cache designed for data-analytic clusters to reduce the impact of stragglers by increasing the memory locality of parallel tasks. The most related approach work to D3N and D4N, Alluxio [25] (formerly known as Tachyon [74]), implements a distributed cache layer on compute-local memory or SSD to improve the performance of the data analytics workloads. Alluxio is a cluster-level cache because typically Alluxio is deployed co-located with compute clusters, is controlled by the cluster not by the underlying storage provider and Alluxio does not support multi-tenancy due to its security model. It allows access to the immutable data lake by mounting the S3-based bucket but implements the user authorization for accessing the cached data based on the POSIX permission model. When Alluxio fetches the data from the data lake, it only inherits bucket-level ACLs when determining file system permissions for a mount point, and ignores the ACLs of a set to individual objects [2, 1]. This means the user has the same access rights for all the objects under the same S3-based bucket. Moreover, there is a lack of access control for short-circuit operations (local read-write), where users can directly access the locally stored data [123].

3.1.3 Meeting the requirements

RQ1 - Support sharing: Data lake level caches naturally support data sharing since all frameworks and clusters running in the data center are sharing cache resources. On the other hand, caching at the client level introduces challenges for data sharing since caching is separate for each framework/cluster these systems are not visible to other frameworks/clusters. As a result, in framework/cluster caching frequently accessed data ends up becoming replicated in multiple caches, and requests are forced

to go to the data lake, even if the data is already available in other caches.

RQ2 - Preserve locality: Data lake caches do not preserve locality. Generally, these solutions are not concerned with preserving locality, since the cache resources are deployed near to the storage nodes and far from the compute cluster where data is being accessed.

On the other hand, client level caching often caches data in the memory of the node where it is being accessed. If they are deployed on a set of co-located physical computers, caching at these levels will have a good locality. However, if caches are deployed in a virtualized environment, then these caches usually do not preserve locality. The lack of data locality is also reported by the Alluxio community when users deploy Alluxio and compute framework on the Kubernetes environment. [7].

RQ3 - Enable elasticity: Data lake level caching naturally supports demand elasticity due to the disaggregation of the storage and compute. If a framework or cluster shrinks, its cached data will be saved in the data lake caches, and the saved data can be used by other frameworks and clusters or by future accesses.

In contrast, cluster level caching does not support elasticity. As we mentioned in section 2.3 demands vary a lot across different applications/frameworks, which may lead some cache resources to underutilized. Shifting caching resources elastically between different frameworks or clusters is challenging; releasing cache resources as the demand shrinks means losing some of the cache states. If the caching system is deployed within the framework or cluster, then we need to grow/shrink the cache as the framework/cluster grows/shrinks. In addition, if the caching system has a write cache, then the cache resources cannot be released immediately when the framework/cluster shrinks until the dirty data in the write-cache is written back to the data lake.

RQ4- support specialization: Data lake level caching does not support specialization for workloads. Understanding the access patterns of different workloads is challenging at this level, therefore these caches usually are workload agnostic. While caching is not specialized, specialized data lakes have been used for particular workloads. For example, a recent study [31] suggests deploying multiple micro object stores, where each one is independently configured and tuned for a particular access pattern to allow the tenants to choose the proper storage for their workload. Facebook originally had in each data center several different HDFS data lakes for analytics, f4 [82] data lake for cold objects, and Haystack [39] for hot objects. To avoid the management complexity of multiple data lakes and the cost of stranded resources, Facebook recently has developed a common data lake, Tectonic [90]. In the same way, we argue that specialized caching is important to match the cache behavior to the workload requirements, Tectonic argues that specialization is critical to supporting a wide range of workloads efficiently.

Framework caches by their nature are specialized to the workload. Cluster caching introduces limitations for specialization since there are different applications and access patterns.

3.2 Cooperative Caching

In the 90s and early 2000s, there were a large body of important research [47, 30, 50, 103, 65, 71, 38, 84, 114, 46, 113, 83, 55, 22] on *cooperative caching* for file systems that extended the file system across different machines accessing it. The pioneering work on cooperating caching by Dahlin et al [47] defines the cooperative caching for distributed file systems in 1994, as a system to *improve network file system performance by coordinating the contents of client caches and allowing requests not satisfied by a client's local in-memory file cache to be satisfied by the cache of another*

client. Cooperative caching forms a new cache layer in the storage hierarchy by using the client's memory as a global file cache, usually located between the client and storage disks, to reduce the disk accesses by increasing the global hit rate..

The original cooperative caches focus on distributed file systems, where the servers connected with LAN(e.g., ATM, Myrinet) have low latency and high bandwidth. The underlying assumption in these studies is that the high-speed local area networks are faster than disks, which means accessing data in a remote memory over the network can be much faster than accessing it from a local disk. In this aspect, the problem is similar in today's data centers with full bisection bandwidth within the rack, an over-subscribed inter-rack network within clusters, and a further over-subscribed data center network between clusters and an enterprise data lake. Most distributed file systems are implemented as an extension of the operating system, where a remote file acts the same as one on a local file system. They use POSIX semantics (or similar interface), which forces some of them to have expensive consistency semantics; whereas, file systems relax their consistency for better performance. Different features of distributed file systems and applications' requirements strongly impact the design of cooperative caching, resulting in a wide range of caching techniques. Management strategies in these systems must deal with trust issues and resource allocation, and move or replicate data between different caches for performance or consistency.

D3N and D4N borrow many ideas from the rich research on cooperative caching in file systems. This section compares our works with past cooperative caching systems, re-visit critical design decisions of post-works, and discusses how we adapt some of these design choices for immutable data lakes.

Trust and Cache Cooperation: In many studies [47, 50, 103, 65, 114] mutual trust was assumed among all participating clients, thus caches on these clients are naturally extension of the file system. In most of the works, caches are owned and

managed by the file system, and all caches participate in cooperation. Shark [30], a peer-to-peer file system with cooperative caching, differs from prior studies by considering mutual distrustful clients. Shark uses encryption and opaque tokens to enable cache cooperation and secure data sharing. Another work, Utility-Based Cooperative Caching (C-Util) [122], shows a new cooperation model for trusted caches belonging to different owners. Each cache is *selfish* and only participates in cooperation if its performance is guaranteed to be improved. This model might be adopted in public clouds where different users access another user's cached data; since public clouds explicitly price their services and resource, thus users can calculate their cost and benefits.

D3N and D4N distribute caches all around the data center, in a similar way network file systems distribute servers on LAN. Because D3N and D4N are an extension of the data lake, unlike Shark, caches in our design are trusted, allowing us to sidestep many of the issues related to data authenticity and trust. To allow sharing and elasticity, in contrast to the C-Util, all caches in our designs are controlled by the data lake and must participate in cooperation for improving the overall data lake throughput.

File/Block Lookup: The location of data in the cooperative cache is found either by hashing [47, 29, 38, 46], where each cache is responsible for storing the portion of keyspace, or by some form of a global directory (e.g., centralized metadata server) [50, 47, 122, 114]. Locating blocks by hashing simplifies the implementation, allows a direct cache to cache communication, thus avoiding the overhead of metadata lookup, and reducing redundant data duplication. However, as reported by previous research [47, 122], placing the data using hashing prevents data replication at the cache layer, and reduces the local hit rate.

An alternative approach is to use a centralized metadata server to maintain the global view of the cache state. The downside of this method is that caches must

contact to the server for every block access, and inform the server whenever a block moves in and out of the cache. For instance, the N-Chance Forwarding algorithm by Dahlin et al. [47] uses a centralized server to determine where the block is cached. Later on, the Berkley xFS file system [29] implements the N-Chance Forwarding algorithm by decentralizing the metadata server by splitting it into many metadata servers. xFS maps files to a particular metadata server and distributes this mapping to all clients and metadata servers. Furthermore, it tries to co-locate the home metadata server of the file with the cache that is accessing the file to improve performance. Other systems have adopted the de-centralized metadata server approach [50, 114, 30]. A different lookup strategy is proposed by hint-based cooperative caching [103], where clients share hints about block locations with each other to minimize the metadata lookup. The first copy of a block to be fetched from the storage by any cache is called the master copy, and only the master copy can be forwarded to another cache. When a cache forwards the master copy to another cache, it will update its hint to show where the block is sent, and if the cache later needs to access the block, it follows the path indicated by the chain of hints to reach the block. The location stored in a hint might not be accurate because the block might already be evicted. As reported in the paper, the location hints become inaccurate at higher rates when the working set approaches the aggregate memory size, resulting in a decreased hit ratio on the client caches, increased costly server accesses, and degraded performance.

To implement highly scalable caches, we avoid using a centralized approach. D3N avoids global state and uses consistent hashing [68] to locate the blocks. D3N uses multi-layer design to overcome the low locality issues reported in previous studies with hashing approach [47, 122]. This way, D3N can still have a good amount of local hit rate since the first layer replicates data for the locality. On the other hand, D4N uses a de-centralized directory to maintain the global state.

Unlike file systems that use small blocks (e.g., 4KB), D4N uses large blocks (e.g., 4MB), and at this granularity, D4N benefits from the key advantages of block-level caching. In addition, directory lookup overhead for caching at a block level is low in D4N due to large blocks and data immutability. D4N has a similar approach to xFS and GMS, where we use a distributed directory and co-locate every cache node with a directory instance. xFS tries to map files to the nearest metadata server of the file's owner(writer); in contrast, D4N uses consistent hashing to map the key to a particular directory instance.

Cache Management: Cache management policies can be implemented based on local decisions with explicit cooperation [47, 30, 103, 65] or based on the global cache state [47]. Dahlin et al. [47] proposed the "Greedy Forwarding" algorithm, where clients serve data to each other, however, every client manages its local LRU cache greedily, without regard to the contents of the other caches in the system or the potential needs of other clients. The same block may be cached multiple times. The greedy method improves the performance by duplicating data if there is a lot of locality in the workloads. The greedy approach is adopted by many other systems, including our work D3N, due to its simplicity and performance [38, 30].

Another common approach is to use the global cache state for management. The cooperative caching algorithms are based on a global state and they tend to distinguish a *local block* that is accessed locally from a *global block* that is cached by a node on behalf of another cache node. For instance, in the "N-Chance Forwarding" algorithm by Dahlin et al. [47], every cache greedily admits blocks into its LRU queue, however if the block is a *singlet* then instead of removing the singlet from the cooperative cache, the cache forwards the singlet to a random cache, otherwise it discards the block. To limit the lifetime of a singlet block, a recirculation count (N) is maintained to limit the number of times a block is forwarded. The algorithm tends to replace

global data with local data in busy clients, and accumulates the global data on the idle caches. "N-Chance Forwarding" algorithm later was implemented in xFS [29].

While "N-Chance Forwarding" uses global information to identify singlet blocks, other systems [122, 50, 114, 47] implement cache replacement policies based on global state. These systems mimic/emulate the global LRU replacement algorithm to identify the least valuable block in the system for eviction, and they also utilize the idle memories of different clients. For instance GMS [50] and PGMS [114], implement a global Weighted LRU. Their algorithm tries to balance cache space between local and global accesses and avoids duplication of global blocks. The GMS algorithm periodically collects the age of all local and global blocks from every cache, determines less loaded caches and the least valuable blocks in the entire system for eviction, and broadcasts the distribution of all old blocks in the cluster to each cache. This way, GMS approximates global LRU. PGMS [114] later extended the GMS work by prefetching blocks into the idle caches. Like PGMS, other early studies also mention the use of idle client's memory as a backing store for evicted blocks. Unfortunately, in data lake implementations, the caches are busy all the time. Therefore, we have to ensure that forwarding a block to a remote cache does not cause eviction of the valuable data. On the other hand, in C-Util [122], clients share their future accesses with a central server, which uses this information to determine for all clients which blocks to store and which peers to serve data. The "hint-based cooperative caching" protocol [103] explores the global cache state by exchanging their local hints, which only approximates the global state of the caching system. The hints are helpful for caches to determine where to evict singlet blocks or where to fetch a block. The management decisions are not optimal but significantly reduce the communication cost.

D3N uses the greedy approach for simplicity and performance, where each cache

manages its own cache based on the local information. On the other hand, D4N implements a cache management policy based on the global cache state. Unlike prior studies, D4N's directory can maintain much richer information about objects, blocks, and cache nodes (e.g., request load, hit rate), because the lookup cost is negligible, metadata overhead is much smaller, and the storage capacity of modern caches are very large (TB capacity for data plus hundreds of MB memory for metadata). Maintaining all this information allows D4N to tune caching strategies per cluster/framework. D4N's algorithm identifies and compares the value of every block globally by maintaining a global age. Unlike GMS [50] or PGMS [114], D4N's algorithm does not need any offline computation for calculating the oldest blocks in the system, D4N only maintains a global age which allows each cache to run the same algorithm independently and compare the blocks in different caches(See 4.5).

Write Cache: Most cooperative-caching solutions [47, 30, 103, 65, 50] implement a read-only, write-through cooperative cache. Previous works with write-enabled cooperative caching have either relaxed consistency or not address data sharing. One of the reason is that prior studies reported that files are usually updated only by their owner, and no files are simultaneously updated and read by many users [46, 22, 115]. Another reasons is that the strong consistency requirements of file systems increase the management overhead of cache coherency; therefore, most cooperative caches avoid providing a write-back cache [38, 103]. Approaches with a block lookup mechanism that ensures cache exclusivity and avoids replication, such as hashing (e.g., C-DHT [122] or Hash-Distributed Caching [47]), can easily provide a write-back cache since there is only a single copy of the block, however, these caches do not provide durability.

For instance, DEFER [84] implements a log-based write-back cooperative cache, and it does not provide consistency in the event of conflicting requests therefore

not suitable for shared environments. NFS-CD [38] provides a write-back cache with support of both relaxed and strong consistency. Strong consistency is provided with a locking mechanism. The writer cache first acquires the lock from the cluster delegate, writes data to its memory, and replicates data on other caches for durability. Dirty data is written to the storage asynchronously at a later time.

In D3N, we take a different approach for non-durable write-back cache than previous studies. D3N writes data blocks to the highest layer and uses consistent hashing to distribute the blocks. It avoids using locks or leases since blocks have a single home location and data lake is immutable, however, it does not provide data durability. D4N, on the other hand, provides a durable write tiers where each write tier maintains a write-back cache that supports both replication and erasure coding [98] for the durability. To maintain cache consistency, D4N uses leases for updating the metadata, and delegates a cache (the primary cache), which maintains the master copy of the block and is responsible for the replication of the block to other caches. Users access only the primary cache for reading data from write-cache.

Granularity: Some existing works prefer to store data at block granularity [47, 50, 122, 29, 65], while others prefer file granularity [103, 30]. In both D3N and D4N architectures, we use block granularity for caching, which allow them to distribute the large files across many caches to balance the load. Since data set sizes vary in the data lakes, using block-based caching avoids external fragmentation and enable partial caching for large objects.

3.2.1 Meeting the requirements

In general most cooperating cache architectures, similar to storage level caches, naturally support wide sharing (RQ1), and preserve elasticity (RQ3). In contrast to storage level caches, cooperative caching preserves locality (RQ2). Generally these systems are general purpose, and there has been little focus on supporting specializa-

tion for caching (RQ4).

Chapter 4

Architecture

This chapter describes the D3N and D4N cooperative cache architectures for immutable object based datalakes. Section 4.1 describes our assumptions, section 4.2 describes common design features, section 4.3 describes differences, and in sections 4.4 and 4.5, we describe D3N and D4N architectures in more detail. We conclude in section 4.6 by discussing the tradeoffs between the architectures.

4.1 Assumptions

When designing D3N and D4N we have the following assumptions regarding the structure of the data center and how compute clusters and data lakes are being incorporated into them.

We assume that the data lake is a separate cluster in the data center, accessible over the data center network from compute clusters needing access. The data access may be significantly constrained by over-subscribed network links both within and between compute clusters, and between the compute clusters and the data lake, or that the data lake itself may have intrinsic performance limits (due to e.g., disk bandwidth), and that addressing these limits may be outside the control of the parties responsible for installing these compute cluster.

We assume that there is a sufficient re-use of the data read from a data lake to make a caching solution valuable, and data sharing is common both within and between the compute clusters. The popularity of data is dynamic and fluctuates

dramatically, and the re-use pattern is complex. The compute servers' local caches may work well, however they can not cache objects shared over different servers both within the same cluster and between different clusters. In addition, many workloads may have large reads that may not fit in a compute server's local cache. There is also some geographic spacial locality in the data centers, where some of the data sets are shared by the applications that are likely to be run on the same compute cluster or on physically close servers.

We assume there some workloads that may create objects which will not be accessed either ever or for a long time (e.g., logs, archival data). These inactive objects cannot be stored at the compute server local storage and may interfere with higher priority reads at the data lake side.

We assume that applications require both high capacity and performance, and storage hardware in the market have different cost, performance, and capacity characteristics. The data lakes store data on less expensive high capacity slow storage devices (e.g., disks, tapes), and caches use more expensive high speed storage devices (e.g., DRAM, SSD).

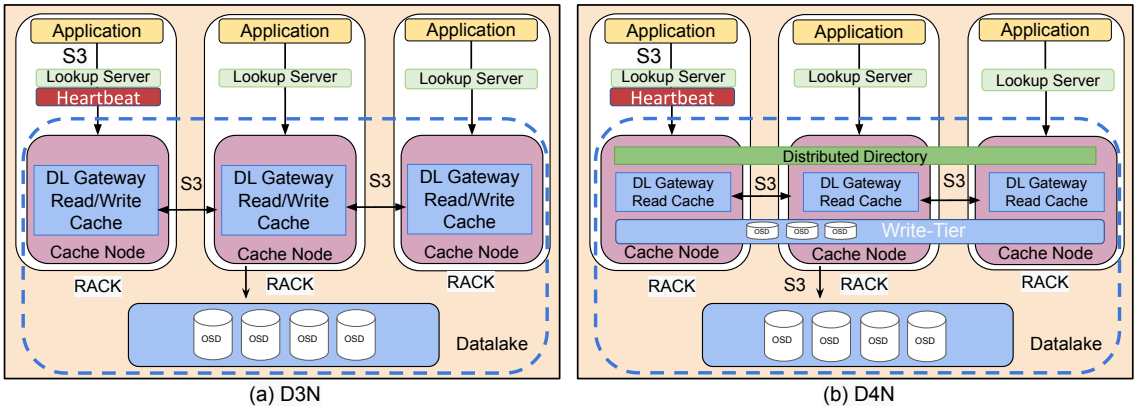


Figure 4·1: D3N and D4N architectures. Applications use the S3 interface to access data lake. *Cache nodes* (purple) are distributed across data centers, and they run *data lake gateway* (blue) that implements caching functionality of each architecture, *Lookup servers* (green) identify nearest cache nodes to client, *Heartbeat service* (red) tracks the set of active cache nodes.

4.2 Shared Goals and Design Features

D3N and D4N architectures share a number of key features:

Extending Data Lake: Both the D3N and D4N adopt cooperative caching, originally designed for file systems [47], to immutable object based data lakes allowing the data lake to be extended across the data center. This enables caching data on the access side of network bottlenecks, reducing the demand on the bi-sectional bandwidth. Multiple caches cooperating to provide increased throughput (via replication) or capacity (via exclusive caching) depending on workload characteristics. D3N and D4N architectures offer the same network interface (S3) as the data lake, requiring no changes to applications. In both designs, all caches are a **trusted** part of the back-end data lake service, and by distributing them, we extend the data lake across the data center; therefore each system support data center wide sharing where the cached data can be accessed anywhere by different clusters or frameworks (RQ1). All caches cooperate to store both locally and globally accessed data, enabling limited

cache storage to be elastically used to cache whatever data is deemed the most needed irrespective of which cluster or framework is using the data (RQ3).

Figure 4·1 shows D3N and D4N architectures in the context of a simplified view of a data center. **Cache nodes** are the basic building blocks, and they implement the **data lake gateway** (DL Gateway, see 2.1) that provides the S3 interface on top of the OSDs as well as the caching functionality. Cache nodes are distributed across the data center in both systems to improve the data locality (RQ2). For example, a cache node can be placed per rack that allows requests to be handled local to the requester, using workload locality to exploit high inter-rack bandwidth and reducing cross-data center traffic. Cache nodes store data for read/write requests but use different cooperation models (Section 4.3). We do not dictate a specific host type for running cache nodes or a specific storage medium to be used for caching by design. A cache node can be a dedicated server in the rack or a virtualized/containerized service. The preferred storage media for the cache nodes is high-speed SSD, providing a combination of high capacity and sufficient bandwidth to saturate network links; however other high-speed storage media such as memory may be used. In our testbed we equipped each cache node with high-performance NVMe SSDs, and we partition the cache capacity between a read and write cache.

Identifying Local Cache Nodes: One of the key advantages of integrating the caches into the data lake, is that they are part of the data center infrastructure. Therefore, we can exploit low-level approaches, to achieve locality.

A client uses DNS to find a cache node. We provide the IP address for the DNS server, we called **lookup server**, on an anycast network [17] that will direct the DNS request to the nearest lookup server, with a timeout of a few minutes. Lookup servers

maintain the logical topology of the caches, and a **heartbeat service** is used to track the set of active cache nodes. We do not use anycast to address the nearest cache directly but instead use anycast to get to the lookup server, that in turn provides the IP address of the nearest cache node. This two level approach is needed to support VM migration. Requests from clients to caches are stateful, and if a client is running in a VM and the VM is migrated, it needs to communicate to the original cache node until the request has been completed. After a VM is migrated, it will continue to communicate to the previous cache node until its use of the DNS mapping has expired and it opens up a new connection. For correctness, clients query the lookup servers both periodically and upon request timeout to handle events such as the recovery of failed caches, client VM migration, or other events which might affect client-to-cache pairing.

Caching at Block Granularity: In each architecture, the read caching is done in **large data blocks**[1] (e.g, 4MB) rather than on a per-object basis, for several reasons: (i) it avoids external fragmentation, and enables to cache the part of large objects, (ii) the blocks making up a large object are distributed across a number of caching nodes in the higher layers, enabling the aggregate resources of many cache servers to be used for a single object, and (iii) in a multi-tenant environment the use of fixed-sized blocks simplifies management of cache resources, allowing per-tenant resource use to be balanced fairly especially in the face of large variances in object size.

Exploiting Composability: The **S3 interface**, which is composable — i.e., capa-

[1]In object stores, the file is called **object**. Objects are stored in a container called **bucket**, and **block** is fixed-size chunk of an object(e.g., 4MB in Ceph). (See 2.1)

ble of being stacked in layers, each providing the same interface to the layer above, creating different systems and services. This enables the layering of rich functionality such as application-specific customization, replication, encryption, security, and similar services. D3N and D4N take advantage of this composability, using the same interface between cache nodes that the nodes expose to clients of the data lake, simplifying the implementation. In addition, D4N implements write tiering on high speed SSD for recently written objects, and employs the same S3 interface to lazily copy aged objects between the write tiers and the backend data lake that uses high-capacity storage. The composability enables both systems to use alternative S3 implementations for high-capacity storage (e.g., MinIO [78] or similar S3 services on top of tape robots)

Exploiting Immutability: Objects are immutable and cannot be modified after they are uploaded in the proper implementation of S3. Object operations are atomic; thus, an object is not visible until the upload completes.Data lakes store objects (and their blocks) with a unique *object ID* and maintains version numbers and the hash of the objects content in objects' metadata. The object ID along with hash of the content and version number changes if an object is deleted or replaced with a different object with the same name as the original. Most object based storage systems commonly provides these unique object ID and version numbers(e.g., S3 [27] and Ceph [118]). Both designs exploit the data **immutability** provided through the S3 interface. We use *object ID* (D3N) and S3 object name and hash of the object content (D4N) to identify the cached blocks. We note that in D4N design, we consider the hash of object content as an indicator for its version number, which allows D4N to identify objects globally both in the D4N write tier and the data lake. To provide

strong consistency, read requests always fetch object metadata from the data lake (D3N) or the directory (D4N); thus, they will always obtain a consistent version of the object. If an object is deleted and a new object is created with the same name, all read cached data is implicitly invalidated because the object will have a new object ID or hash. In contrast, previous cooperative caching approaches for file systems needed to aggressively invalidate any cached data in the read cache, imposing considerable overhead and complexity. D3N does not keep track of read cache, and D4N, while keeping track of where data is cached for performance, does not need to maintain this information resiliently.

Integrating into Data Lake Gateway: Instead implementing a new and separate cache layer, both designs integrated into existing gateway functionality, and they allow clients to rely on just one implementation of complex data lake services. Both architectures rely on mechanisms(e.g., authentication, serialization,synchronization) provided by the data lake (Ceph) software. For instance, D3N and D4N piggyback the IO serialization mechanism provided by the data lake gateway to maintain correct read-after-write and shared-write semantics between multiple clients(See Section 4.3). They rely on the data lake gateway logic for object fragmentation, flow control mechanism to order the incoming and outgoing requests, limits the number of outstanding requests, and S3-compatible authentication mechanism and access control list (ACL). D3N relies on the data lake for namespace operations (e.g., generating metadata for an object). D4N deploys the existing data lake software as a write tier and uses all mechanisms provided by the data lake (e.g., replication, synchronization, fault tolerance) to implement the write tier. This enables D4N to exploit existing erasure coding support of the data lake, which is one of the major reasons why D4N provides

better write performance than other caches that rely on replication(Section 6.3.2). In addition to the performance and complexity advantage of gateway integration, the fact that D3N and D4N rely on existing control flow and the gateway implementation greatly reduces the barrier to acceptance by the upstream community, and ensures that enhancements are made to the data lake will automatically apply to caches. Moreover, users have a strong guarantee that both the D3N and D4N data lake extensions provide exactly the same durability guarantees, not requiring them to trust two different implementations at the cache and data lake level for the resilience of their data.

Workload Adaption: Both architectures automatically adapt to workload characteristics, and neither of them requires applications to control the cache management. The cache nodes make caching decisions based on historical information without any input from the application. For each architecture, we developed a cache management algorithm that controls replication and dynamically adapts to the access patterns of the applications. D3N partitions the cache space between local and remote requests using miss ratio curves and measured latency; while, D4N uses global information to avoid redundant caching when the working set size is large.

4.3 Design Differences

The fundamental differences between the two systems is that; D3N is designed to be readily integrated into a data lake, minimizing the changes in the complex mechanisms for ensuring consistency, availability, and durability of the data lake. D4N, on

the other hand, is a more sophisticated design which demonstrates the greater value that can be achieved by modifying these mechanisms to implement richer support for cooperative caching and tiering.

Cache Cooperation and Block Lookup: D3N and D4N differ on how each system organizes its' caches and locates the blocks. D3N is a **hierarchical multi-layer cooperative cache**, where layers correspond to wider domains of cooperation, and misses are forwarded to the next higher layer using a *consistent hashing*. On the other hand, D4N is a **non-hierarchical cooperative cache**, where it creates a single aggregated distributed cache layer and uses a *a distributed directory* for locating the blocks.

Supporting Write Requests: The figure 4·1, shows that their support for write requests also differs in D3N and D4N. D3N provides a non-replicated write-back cache only for the intermediate data and writes the blocks into the highest layer. Once the object is written into the write-back cache, its metadata maintained in the data lake is also updated. On the other hand, D4N provides reliable write tiers where each tier stores data with erasure coding or replication. D4N exploits the composability of the S3 interface, layering the write tier on top of the existing data lake. In contrast to previous cache systems that implement their own mechanism for maintaining dirty data redundantly, D4N re-uses the existing data lake (Ceph) software for implementing a write tier. Thus, D4N provides exactly the same guarantees, using the same implementation for accelerating write operations through tiering as the underlying data lake.

Maintaining Consistency: Another difference between the two systems is how they maintain their consistency. In D3N, we rely on the underlying data lake for consistency, where the data lake maintains per object metadata, including the unique object ID. For every request, D3N first retrieves the per-object metadata and its object ID from the data lake before serving any cached data. This allows clients to *bypass* the D3N caches and to have direct access to the data lake. In the D4N design, the write tier (the existing data lake (Ceph) software) generates per object metadata for dirty data, including the the hash of object's content. D4N maintains the object metadata in the distributed directory, and for every request, it retrieves the object metadata from the directory; thus, D4N does not allow applications to bypass the cache layer since the writes must go through the cache nodes for consistency.

Design Complexity: In D3N, we explicitly focus on implementing a real working caching system and "upstream" the code into an existing open-source Ceph codebase to allow its use by a broader community; therefore, *simplicity* is one of the key principles of D3N design. To keep the design simple, D3N focuses on a *read cache* and supports a *write cache* only for intermediate data sets, avoids any global state by using consistent hashing to locate blocks, and uses the same cache management strategy for all workloads. D3N design focuses on read-mostly workloads, particularly for big data analytics, where objects are large and accessed in their entirety.

D4N explores the use of the global state to implement more sophisticated cache management policies and enable application-specific tuning of caching policies to support a broader range of applications than D3N. D4N architecture provides a reliable *write tier* and uses *a distributed directory* to maintain the cooperative cache's global state (object and block metadata) and locate blocks in the cache. This way, D4N

separates its caching mechanism from its caching policies, enabling caching policies to be customized per workload to have flexible data placement and cache management (RQ4). Thus, D4N can accelerate the performance of a wide range of applications with different access patterns. D4N architecture is significantly more complex, maintaining a global state and a reliable write tier.

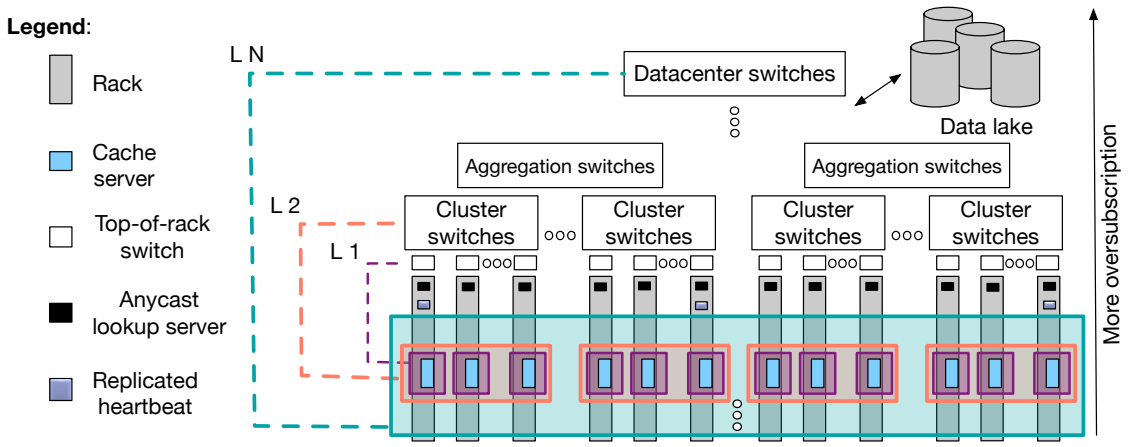


Figure 4·2: D3N multi-layer cache architecture. Cache layers: Layer 1 (L_1), Layer 2 (L_2), and Layer 3 (L_3) are introduced at the top of rack, aggregation, and cluster switching network layers, respectively. L_1 caches data for its local clients, L_2 caches data for the clusters, and L_3 caches data for multiple clusters.

4.4 D3N Architecture

D3N caches are designed to be integrated into an existing data lake, with the caching functionalities are implemented by modifying an existing object storage gateway to preserve the data lake properties. Figure 4·2 shows how D3N creates caching layers based on the hierarchical network topology. Here D3N is deployed across a set of racks (or *clusters*), caching data from a remote data lake. We assume that the resources within these racks may be optimized for their tasks, incorporating sufficient caching and network resources to support the expected demand from the client machines in the cluster. Conversely, we assume that bandwidth may be limited between the clusters and the data lake, or that the data lake itself may have intrinsic performance limits (due to e.g., disk bandwidth).

4.4.1 Components

Key elements of D3N's architecture are its *caching layers*, the *read cache*, and the *write cache*.

Caching layers: D3N introduces cache layers based on network topology to mitigate the performance impact of network limitations. As seen in Figure 4·2, cache layers: layer 1 (L_1), layer 2 (L_2), and layer 3 (L_3) are introduced at the top of rack, aggregation, and cluster switching network layers, respectively. Multiple cache layers can be co-located in the same cache node. Each cache node acts as a L_1 cache for its local clients, caching requested data, while successive cache layers are formed by aggregating resources across multiple caches. In each cache layer, every block has a single "home" cache node determined by *consistent hashing* [68]. For instance, in Figure 4·2, consecutive L_1 caches cooperatively form a L_2 cache layer and consecutive L_2 caches cooperatively form a L_3 cache layer.

A cache node can co-locate multiple layers, and it combines these caches in a *unified cache* where space can be dynamically traded off among layers based on where bottlenecks are observed. For example, when compute nodes on all racks are accessing remote data, significant cache space may be dedicated to the L_2, maximizing the amount of data cached within the cluster; if only one rack is busy the space on its associated cache server may be dedicated to L_1, eliminating the need to fetch data from other racks. Co-located layers on the same cache node enables a unified cache, only one copy of a block is stored even it is logically cached in multiple layerson that node.

Read Cache: All caching in D3N is done in block granularity, and cache nodes maintain a local mapping of blocks. A block[2] is identified by its object ID generated by the data lake and the block offset, and is cached as a file on an SSD-backed file system. The read cache uses local information for cache management, and runs the same cache management strategy for all requests. D3N allows "pluggable" cache management policies and can support any eviction policy, e.g., LRU, or LFU; currently it uses

[2]Note that object stores map object onto a sequence of fix size large data blocks (e.g., 4MB), and distributed blocks across storage servers, and D3N processes every block independently.

independent per-layer LRU.

Write Cache: D3N supports *write-around*, *write-through* and *write-back* caching, which applications can control on a per-object basis. An object is identified by its object ID, and the metadata of cached blocks are maintained in the data lake. Write-back cache does not replicate the objects therefore it is only suitable for temporarily datasets that does not require strong reliability guarantees.

4.4.2 Data Flow

We introduce metadata semantics, the main operations and the data flow to support read and write requests.

Metadata Semantics: The data lake maintains per-object metadata, including the unique object ID and the mapping of an object to corresponding data blocks. Objects are immutable, and uploading an existing object creates a new one, assigning it a new object ID number. Cache nodes in D3N maintain a mapping between every cached block and object. Per-object metadata is requested from the data lake before any cache operation. When a new data is written to the cache object metadata in the data lake is updated.

Read Request: For reads, clients send requests to the nearest cache node (L_1) as identified by the lookup service. D3N cache node retrieves the object ID from the data lake for the request object, then divides the object into (typically 4 MB) blocks, where each block is identified by their object ID and offset. If it is L_1 hit, the cache node returns the requested block immediately. Upon an L_1 cache miss, the cache node uses consistent hashing [68] to locate the block's "home location" within the higher layer (e.g., L_2), and the request is forwarded to the block's home location via S3 range request. If the block is a miss at the home location (i.e L_2) of the highest level, then the block is retrieved from the data lake, a copy of the requested block is stored both at the home (i.e. L_2) and client-serving (L_1) locations. Co-located

layers' read caches are *unified*: L_1 requests for a block received at that block's home location result in a single cached copy of the data.

Write Request: Clients send requests to the nearest cache node (L_1) as identified by the lookup service. To support different use cases D3N supports write-through, write-back and write-bypass. These modes can be selected on e.g., a per-object basis, by overloading the Put request (adding header). The *write-around* policy disables the D3N cache for write operations.The L_1 cache forwards the write request to the data lake without caching it. This mode avoids polluting the cache with data that will only rarely (or never) be used in the future (e.g., archival data).

With the *write-through* policy, clients must wait for data to be synchronously written to the data lake before continuing. This reduces the opportunity for data loss at the cost of performance (our caches are single points of failure, whereas the back-end storage is highly fault tolerant). Write-through caches data at the points it would be cached by reading: in a two-layer cache scenario, L_1 writes blocks locally, to the data lake, and to their last layer (L_2) home location. The data lake metadata is updated when all writes to the back-end are completed, and the write is then acknowledged to the client.

The a *write-back* policy caches blocks in the last layer (e.g., L_2), to give all clients a consistent view of newly-written data. As soon as all data blocks are cached in the last layer, the cache node which receives the write requests from a client notifies the data lake. The data lake generates the metadata of the object with a new object ID and then the write is acknowledged to the client. Dirty blocks in the write-back cache are flushed periodically, or under certain circumstances such as eviction of a block or a user flush command. The *write-back* policy does not replicate the data, which results in increased performance at the cost of reduced reliability. This mode is useful for storing data that does not require strong reliability guarantees, such as

workload's intermediate data.

In both write-through and write-back, failure during write may result in blocks prior to the point of failure being cached; however, since data lake metadata has not been committed, these stale blocks will be inaccessible and eventually evicted.

4.4.3 Dynamic Cache Size Management

In a data center, workloads change throughout the day and the demand for network and storage fluctuates. To react to these fluctuations cache and network resources must be allocated carefully based on the demand. For instance, the network to back end traffic might be congested, and in such a case the cache should store more data for global accesses, or if workloads repeatedly process the same data sets and there is limited sharing among workloads, then data sets should cache in rack local cache servers. We have developed an algorithm that dynamically adjusts cache sizes of each layer based on observed workload patterns and network latency to minimize mean request latency.

The algorithm dynamically adapts the fraction of cache devoted to local vs. global requests at each cache server, with a per-layer eviction algorithm (e.g. LRU) used within each pool; chunks shared between L_1 and L_2 are purged when they have been evicted from both. In particular, at each layer D3N tracks both miss overhead and miss ratio curve (MRC), using a shadow LRU list for MRC tracking. This information allows periodic adjustment of cache allocation: the MRCs may be used to predict the change in L_1 and L_2 hit rates if capacity is moved from L_1 to L_2 or vice versa, and mean response times for L_1 and L_2 misses used to estimate impact of such a change. This approach adapts the size of L_1 and L_2 based on the application working set and network bottlenecks. If L_1 is too small, its miss rate will be high and there will be many remote accesses to some L_2 and if L_1 is too large, L_2 will be too small and its miss rate will be large causing even longer latencies due to fetches to the data lake.

Algorithms 1 and 2 shows our algorithm where we based the predictions of miss rates as a function of layer size on observed access patterns and measured miss latencies in the nearest past.

To approximate the miss rates, a shadow LRU [76, 66, 43] cache SL, is maintained for both layers. Shadow caches have been explored in other works. Each shadow cache is of full size, S_t and only stores the keys but not the data. There is a hit counter, HC_l, associated with each shadow cache. For an access to block b to some layer, the associated shadow cache is accessed. If b is found in location i, then the hit counter for location $\mathrm{HC}_l[i]$, is increased and the blocks rearranged to maintain the LRU ordering. If the layer has size s, then the sum of all $\mathrm{HC}[i]$ for $i > s$ is the miss rate for that layer.

Algorithm 1 Re-use distance measurement

```
1: b: requested block
2: ℓ: layer (1 or 2)
3: S_t: total cache size (in blocks)
4: SL_ℓ: shadow LRU list (length S_t)
5: HC_ℓ: re-use distance histogram
6: s⃗ = (s_1, s_2): cache distribution, s_1 + s_2 = S_t

   ▷ Measure re-use distance for access to block b, layer ℓ
7: procedure MEASURE(b, ℓ)
8:     if b ∈ SL_ℓ then
9:         find i s.t. SL_ℓ(i) = b                          ▷ LRU position
10:        HC_ℓ(i)++
11:    end if
12:    reorder SL_ℓ LRU due to access to b
13: end procedure
```

Periodically we use the re-use distance histogram and mean miss latency measurements $\overrightarrow{L} = (L_1, L_2)$ to adapt the cache capacity allocation. We first considered a simple additive increase/decrease mechanism; however due to the wide range of possible allocations ($S_t \approx 10^6$ in our prototype) the response time of such an algorithm is very slow. Instead we search a fairly wide range[3] of possible allocations ($\pm q$, where q called adaptation limit, which has been set empirically to $0.05S_t$ in our prototype) and select the best from this range.

[3]We limit the space searched, and thus the absolute magnitude of any correction, in order to bound the eviction overhead after a large change in allocation.

More specifically, Algorithm 2 shows how starting at an allocation $\overrightarrow{s} = (s_1, s_2)$, we find a new assignment $\overrightarrow{s}'$ with a predicted miss rate $\overrightarrow{m} = (m_1, m_2)$ which minimizes expected latency $m_1L_1 + m_2L_2$. We first (lines 7, 8) calculate the miss ratio curve MR_ℓ for each layer, allowing us to predict the miss rate at that layer for varying cache sizes. We then search a range of possible cache allocations $\overrightarrow{s}'$, centered at $\overrightarrow{s}$, selecting the allocation $\overrightarrow{s}'$ which minimizes the expected request latency. After adapting the cache sizes (if $\overrightarrow{s}' \neq \overrightarrow{s}$), we scale the distance histogram HC_ℓ so that it represents a moving average[4] of re-use distance frequency, balancing accuracy (from accumulating data over multiple periods) with rapid adaptation (due to the rapid decay constant).

Algorithm 2 Cache distribution adaptation

```
1: b, ℓ, S_t, s⃗, HC_ℓ: As in Algorithm 1
2: MR_ℓ: miss rate (i.e. miss ratio curve)
3: L_ℓ: measured miss latency
4: q: adaptation limit (maximum assignment change in blocks)
5: i: cache server location

   ▷ Calculate updated L1L2 cache distribution s⃗_new
6: procedure ADAPT
7:     for ℓ in 1, 2 do
8:         MR_ℓ(i) = Σ_{k=i}^{S_t} HC_ℓ(k)          ▷ Calculate miss ratio curve
9:     end for
10:    C_min = inf
11:    s_new = ∅
12:    for s⃗' in (s1 − q, s2 + q) ... (s1 + q, s2 − q) do
13:        m⃗ = (MR1(s1), MR2(s2))                   ▷ Predicted miss rate
14:        C = m1L1 + m2L2                          ▷ Expected latency
15:        if C < C_min then
16:            C_min = C
17:            s_new = s'
18:        end if
19:    end for
20: end procedure
```

Memory Overhead and Algorithm Complexities: The overhead of the D3N adaptation algorithm includes (1) memory used for the shadow LRU lists SL_ℓ and re-use distance histograms HC_ℓ, (2) computation to track re-use distance statistics (Algorithm 1) and (3) computation to find an optimal capacity allocation (Algorithm 2).

[4]Since the incoming counts are not scaled, the expectation of HC_ℓ is actually $2\times$ the mean for a single collection period, a constant factor which does not affect the location of the optimal point.

The shadow LRU list must store S_t different block identifiers each of which is an RGW object ID (up to 128 bytes) plus an offset (4 bytes), or $132 \cdot S_t$ bytes each for the L_1 and L_2 shadow lists; with a 4 TB cache and a block size of 4 MB, this is a total of up to 264 MB for the two layers. We use a skip list [81] to calculate LRU position in $O(logN)$ time, for an asymptotic complexity of $O(logN)$ for Algorithm 1. Locating $\overrightarrow{s}'$ in Algorithm 2 searches $2 \cdot q$ cases, where $q = O(S_t)$, for an asymptotic complexity of $O(S_t)$, although this may be reduced by using a more sophisticated minimization algorithm.

Extension to 3 layers and more: In this case Algorithm 1 is unmodified, updating e.g. SL_3 and HC_3 in the same way as for lower layers. The exhaustive search in Algorithm 2, however, is clearly infeasible for 3 or more layers, and must be replaced by a more efficient minimization algorithm such as hill climbing [43, 44].

4.4.4 Edge Conditions and Failure Modes

Coherence: D3N serves only immutable data. Objects are written first into the write-back cache, and then the metadata is updated. For the write-back cache, in a multi-writers use case, D3N relies on underlying data lake that serializes the IOs. Cached blocks are stored with their unique object ID generated by data lake. Before serving read requests, the cache node fetches object metadata (object ID) from the data lake and does not serve cached data if the objects IDs do not match. Once the object ID is retrieved from the data lake, D3N constructs the block IDs based on the object ID; thus, the cost of this data lake access is amortized over the (typically) large read requests generated by workloads such as big data analytics. This way, read cache always obtain a consistent version of the object. For writes, as objects are immutable, blocks are cached with a new object ID and offset, and old blocks are invalidated.

Failures: If a cache node fails, any outstanding requests will fail; the application

(or lower-level cache) is responsible for retrying the failed request. In the case of in-process requests, the lookup service will redirect subsequent accesses to an alternate cache nodes, chosen randomly to uniformly distribute load across remaining nodes. For cache-to-cache access, the failed node will be omitted from the relevant cache layer, and requests which would previously be directed to that node will be spread across the remaining cache nodes. If a cache node fails, the dirty data in the write-back cache will be loss; however if the header has not yet been committed, these stale blocks will be inaccessible and eventually evicted. For clean data, there is no need to recover data stored on the failed node; clean data can be fetch from the data lake.

Scalability: D3N can scale horizontally by adding more cache nodes. Each cache nodes can query the list of active cache nodes from the look-up server and also periodically send a heartbeat information to announce that it is still alive with a flag associated with its layer. Any cache nodes that does not send periodic heartbeat messages for some time is considered dead and removed from the list. The consistent hash ring is updated whenever a cache node is added or removed; thus, before adding or removing a cache node, the dirty data on the write-back cache must be flushed back to the underlying data lake.

Load Balancing: Users query the lookup server for the L_1 server. Lookup servers distribute the file requests to L_1 cache node considering closeness and load to avoid hot-spots. On all other layers (L_2 to LN) hotspots are avoided as we use consistent hashing to distribute blocks.

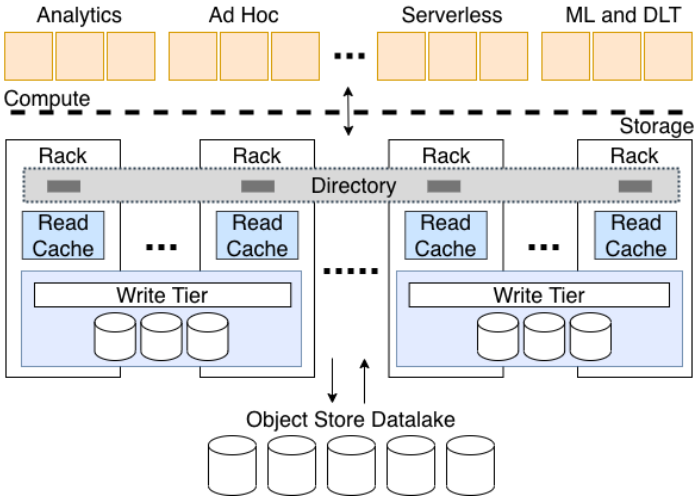


Figure 4·3: D4N architecture. D4N supports a wide range of applications. D4N deploys multiple write tiers in large data centers to increase the write locality for applications.

4.5 D4N Architecture

Figure 4·3 shows D4N's high-level architecture in the context of a simplified view of a data center. Like D3N, D4N also extends the data lake by distributing cache nodes around the data center. D4N uses a distributed directory to maintain the global state, and provides reliable write tiers to stores data either with erasure coding or replication, and the dirty data is periodically copied to the data lake from write tiers. In contrast to previous cache systems that implement their own mechanism for maintaining dirty data redundantly, D4N re-uses the existing data lake (Ceph) software for implementing a write tier and exploits the semantics of immutable objects. As seen in Figure 4·1, D4N implements its caching functionality by modifying the data lake gateway. D4N exploits the composability of S3 interface, layering write tiers using the existing data lake software on top of the existing data lake.

Every cache node supports a local read cache and participates with other cache nodes in a write tier. We choice to keep the read cache and write tier separate because;

i) We can deploy different cache management and replication policies for the read and write tier that is more suitable for the access patterns of each cache. ii) Read requests won't be affected by write requests and vice versa. iii) Write tier is durable, where it replicates data, however read cache is not durable. Read cache caches data not only for the data lake but also for the write tiers.

4.5.1 Components

Key elements of D4N architecture are the *directory* used to maintain global state, the *cache* for read data, and *write tiers* for recently written data.

Distributed Directory: A global directory maintains information about the state of all cached objects and the status of different cache nodes. The directory is distributed across the cache nodes using a highly available (replicated) in-memory key-value store. Consistent hashing [68] is used to identify the cache node responsible for each directory entry.

For each cached object, the directory maintains the state needed to authenticate requests (owner, ACLs) and identify objects in the write tier (dirty state & cache location). The key used to locate the directory entry is a combination of the object's bucket and name. We replicate the directory entries only for write tiers to prevent data loss.

On a per-block basis (e.g., 4MB), the directory also maintains a list of all the nodes caching a copy of the block. The key used to locate the block entry is a combination of the object's bucket and name, and the offset of the block in the object.

For every cached object and block, the directory supports extensible state that is used by specialization. For example, the directory can maintain additional state to enable pinning a block into the cache to prevent it from being prematurely evicted or to enable marking an object as temporary (short-lived) for reuse and to avoid the

write load on the data lake.

Read cache: The read cache is used to distribute load across the cache nodes and reduce demand on the data lake, write tier, and network. Blocks are cached in a portion of each cache node's local storage, and a local mapping is maintained to enable hits to be handled without a directory lookup.

If a requested block is not in the local cache, the directory is checked to tell if it can be fetched from another cache node (block can be found in another read-cache or in the write tier) otherwise it is fetched from the data lake. The cache node may store the block for requests it handles, even if those blocks are already in the write tier or another node's read-cache, but tries to avoid creating addition copies of cached blocks if it would result in evicting blocks that will need to subsequently be re-fetched from the data lake (see section 4.5.3).

Write Tier: The write tier is used as a "filter" to control and limit write IO to the data lake. Data objects are stored only in one place; either on the write tier or on the underlying data lake. To allow short-lived objects enough time to "die in the cache node", D4N lazily writes dirty objects to the data lake and ensures read-after-write requests are satisfied from the cache.

In a large data center, multiple write tiers can be initiated for write locality. The write tier is *distributed* across cache nodes. Cache node participates in one of the write tiers, and writes data at object granularity using a portion of cache node's local storage. Large objects ($\geq 4MB$) are stored using erasure coding (e.g. RS(4,2) [98]) and small objects ($< 4MB$) using triple replication across the cache nodes participate in the nearest write tier.

4.5.2 Data Flow

We introduce the main operations and data flows to support both S3 read and write requests.

Read Request: When a cache node receives a read request for an object, it retrieves the blocks belong to the object. For every block, upon a local cache miss, the cache node issues a directory lookup. If the copy of the block exists on either an another cache node or the write tier, then the block is fetched, a copy of the block is stored in the local cache. If the block doesn't exist in any cache node or directory lookup is timeout, then the block is retrieved from the data lake, a copy of the block is stored on the local read cache. In both cases the directory entry for the requested block and the local cache mapping is updated.

Write Request: When a cache node receives a write request for an object, it is responsible for durably writing the newly generated object into the caches in the write tier that the node is participated in. Every cache node maintains a FIFO queue to keep track of the IDs of dirty objects they received and their creation time. Once the object (all blocks) is written successfully, object is marked as "dirty", the write tier cache identity is updated in the directory and the cache node inserts the object id into its FIFO queue. Writes are atomic and overlapping, concurrent writes will reflect a particular order of occurrence.

Cleaning Dirty Objects: Every cache node is responsible for writing the dirty objects in its FIFO queue to the data lake. When the head object in the FIFO queue reaches its lifetime, the cache node retrieves all blocks belong to the object, and copies the object to the data lake, updates the object's directory entry, and deletes the object and its replicas from the write tier.

D4N uses a priority queue for the remote cache and data lake requests and all interactions are asynchronous. Reads and writes have priority over writing dirty data

back except in the case when the free space capacity drops below 10%.

To allow specializations, dirty object can be marked as "short-lived " on the directory (unless we are out of cache space), and D4N avoids moving it to the data lake. If requester fails to delete "short-lived " object, then D4N makes sure that it is cleaned up after some reasonable amount of time. The maximum lifetime for objects vary across different applications. In our design, we set a maximum lifetime for every object in the write tier(e.g., 5× of the average object lifetime). When an object reaches its maximum lifetime the cache node moves it to the data lake and delete it from the write tier

List Bucket Request: D4N provides a single namespace federating the objects in the write tier and the data lake. Upon a list bucket request, the cache node retrieves the list of objects in the bucket from the write tier and the data lake .

Delete Object Request: Upon a delete request, cache node removes the object and its replicas either from the data lake or the write tier, and update the directory. We don't need to do anything for the read cache, because the eviction algorithm removes the deleted objects from the cache.

4.5.3 Cache Replacement Algorithm

D4N employs a novel algorithm that uses the global knowledge of cache node contents stored in the directory and make decisions based on the global value of a block.

Our algorithm, *Globally Weighted Frequency (GWF)*, is based on LFU with Dynamic Aging [32]. The idea of LFUDA is to adapt to changing workloads by adding the "age" of the cache to the block value on each access, where the age is defined by the weight of the last block evicted. As age grows, recently accessed blocks' value get incremented by larger amount and therefore older polluting blocks will be eventually evicted.

The intuition behind GWF, is that with a global age the weight of blocks in

different caches can be compared. As seen in the algorithm 3, two weights for each block is maintained: a *local weight (lw)*, that is incremented for only local accesses and represents the weight (or value) of that block to the local cache, and a *global weight (gw)*, stored in the directory, that is incremented by the local cache age when the block is remotely accessed. The sum of *lw* and *gw* of a block gives you the value of that block in the whole cache. Age is a global value stored in the directory, and to reduce communication overhead, each cache node updates its (potentially) stale version of age, and the max value is periodically updated and broadcasted to all cache nodes.

For eviction, algorithm 3 lines 26-42, GWF picks a victim block with minimum *lw* in the target cache. If the victim block has another copy in a remote cache then its local weight is added its global weight. In the case of the victim block being the last copy, the sum of the *gw* and *lw* of that block is used to determine if it is still the best candidate for eviction.

To use the space of underutilized remote caches, algorithm 3 lines 33-36, GWF tries to push the evicted block to a remote cache by comparing the evicted block's *lw* against the average local weight of blocks in other caches. If *lw* of the block is higher than average weight of any cache node, GWF will push the block to a cache node with the lowest average weight.

4.5.4 Edge Conditions and Handling Failures

Coherence: D4N uses a directory-based cache coherence policy [73] and enforces strong consistency. Like the data lake, the write tier, which uses the data lake implementation is also immutable. Both write tier and the directory operations are atomic and they serialize the requests for consistency. An object is visible after it is written to write tier and the directory metadata is updated. D4N uses **leases** [52] over the directory keys (similar to NFSv4 locking mechanism) which allow cache

Algorithm 3 D4N's GWF GetBlock and Eviction Algorithms

```
1: b: requested block
2: size_b: size of block b
3: cache_j: cache node j
4: age: the maximum age of among all the cache nodes
5: lw_jb: local weight of block b in a cache node j
6: gw_b: global weight of block b in the directory
7: lw_jvictim: weight of the victim block in a cache node j
8: ---------------------------------------------------------------
9: procedure GetBlock(b, cache_j)
10:     if b ∈ cache_j then                                  ▷ Local copy
11:         lw_jb+ = age
12:     else
13:         while freeSpace_j < size_b do                    ▷ Not enough space
14:             freeSpace_j+ = EVICTION(cache_j)
15:         end while
16:         if b ∈ cache_{k|k!=j} then                       ▷ Remote copy
17:             Fetch b from R = rand(cache_{k|k!=j})
18:             gw_b+ = age
19:         else                                             ▷ No remote copy
20:             Fetch b from datalake
21:             lw_jb+ = age
22:         end if
23:     end if
24: end procedure
25: ---------------------------------------------------------------
26: procedure Eviction(cache_j)
27:     victim = victim block with minimum local weight
28:     if victim ∉ cache_{k|k!=j} then                      ▷ last copy
29:         if gw_victim! = 0 then
30:             lw_jvictim+ = gw_victim
31:             gw_victim = 0
32:         end if
33:         cm = cache node with minimum average weight
34:         if lw_jvictim > AvgWeight_cm then                ▷ block has better value
35:             PushBlockToRemoteCache(victim , cm)
36:         end if
37:     end if
38:     gw_victim+ = lw_jvictim                              ▷ Remote copy
39:     cache_j = cache_j - {victim}
40:     age = max(lw_jvictim, age)
41:     return size_victim
42: end procedure
```

nodes to operate on shared keys in a mutually exclusive way. To prevent performance degradation, the cache node acquires a lease on a particular key only for updating a string value to prevent overwrites. We use string values for location of objects and blocks, hash of the object content, objects' ACLs and timestamp values. For writes, the metadata in the directory is updated after the object is written into the write tier. The write tier generates a hash of the object content which is maintained in the distributed directory along with identification of a block(S3 bucket and object name, and offset). The hash of the object content can be considered as version numbers, which allow us to identify the final version of object across data lake and write tier. Similar to D3N design, identification of a block(S3 bucket and object name, and offset) and the hash of the objects is used for cache invalidation, where D4N does not serve cached data if these values of the block does not match.

If there are multiple write tiers, to serialize concurrent writes of the same object, the lease for a key in the directory must be acquired (blocks the other writer) before the object is durably written into the write tier, and once the write is completed then the directory is updated and the cache node release the lease.

Scalability: D4N can scale horizontally by adding more cache nodes, and a cache node contains a read cache and a directory instance, and portion of write tier.Each cache node can query the list of active cache nodes from the look-up server and also periodically send a heartbeat information to announce that it is still alive. Any cache node that does not send periodic heartbeat messages for some time is considered as dead and removed from the list. Since write tier uses the existing object storage based data lake software; thus it is highly scalable. Scalability of the distributed directory achieved by using a consistency hashing to map the keys to a directory instance.

Handling Failures:

Cache failure: For the read cache, a failure is not a concerning since blocks are already

stored redundantly either in the write tier or in the data lake.

Directory failure: The directory content for the write tier is triple replicated, so upon failure the request for the lookup will be forwarded to the another replica. The consistent hash ring is updated whenever a directory instance added or removed, and consistent hashing re-balance keys across the directory instances.[5] Even if the entire directory fails, caches are still operational and requests are served from either nearest cache or from the data lake. In such scenario, only the dirty data on the write tier, not yet written back to the data lake, will be accessible. To prevent permanent data lose in D4N, we can scan every entry in the write tier metadata and reconstruct the directory content.

Load Balancing: Lookup servers distribute the file requests to the cache node considering closeness and load to avoid hot-spots. D4N balances the load on the distribute directory via consistent hashing. Write tiers rely on the load balancing mechanism of the data lake that distributes blocks and replicas using globally known mapping function (such as CRUSH [119],which provides a good load balancing.

4.5.5 Cache Specialization

A key feature of D4N is its support for *specializations*, application-specific features to improve performance for specific workloads. In contrast to application-specific data stores, these specializations retain the advantages of a single shared storage system such as statistical multiplexing and avoidance of stranded resources. Implementing the specialization on data lake caches instead of the application or framework allow sharing.This serves our design goals of transparency. We have prototyped two such specializations as a proof of concept: small object coalescing and packing, and smart prefetching and caching for DAG-based applications.

[5]Consistent hashing moves the minimal amount of data between the nodes whenever there is a ring change.

Small object support: This specialization optimizes small object creation and storage by batching writes and combining them into a smaller number of large objects. The approach is similar that used in prior work by Kadekodi [67], which coalesces small objects to reduce AWS per-operation costs. The underlying strategy of batching writes for performance has a long history dating back to the log-structured file system [100] and earlier.

When copying from the write tier to the data lake, D4N can combine multiple small objects into a single large one, maintaining an index of the mapping from small object name to large object, offset, and length. When small object coalescing is enabled for a bucket, objects under a threshold (4MB) are tracked as they enter the write tier; when a configured size (32MB in our experiments) is reached or an idle timeout triggers they are written to the data lake as a single large object, and may be retrieved individually via range requests. Locations of the coalesced small objects are initially stored in the distributed directory, and are lazily persisted to the data lake as described below. Directory leases are used to ensure only one cache node will write coalesced objects to the data lake or update mapping tables at any one time, as well as protecting the metadata tables.

Small object map information is gathered into tables (i.e. SSTables [42]) sorted by bucket name, and written to the data lake as objects in an administrative bucket. Keys may be found by binary search in each object, searching the newest first. similar to an LSM tree [88], with a similar merging strategy to bound the cost of lookup. Unlike an LSM tree, the directory acts as both memtable and a cache: after finding an entry, a range of entries containing that key is copied to the directory, causing other entries to be evicted.

Finally a garbage collection mechanism (not yet implemented) is needed to handle deletion of coalesced small objects. The cost of this cleaning is expected to be

significantly reduced by the delayed batch writeback: short-lived small objects will be deleted in the write tier, and never coalesced into larger objects in the data lake.

Supporting analytical workloads with DAGs: We modified the previously developed system by our group Kariz [20] for analytical frameworks. Kariz is a customization of D4N for caching data from data lakes accessed concurrently by multiple compute clusters running analytic platforms. Kariz extracts rich information from the analytics platforms(PIG and SPARK) in form of Directed Acyclic Graphs (DAG) to prefetch input objects and pin intermediate objects in the cache. With extracted DAG information, Kariz can provides application level hints to the cache nodes and enable efficient caching. We modified Kariz and it obtains the cache content from the distributed directory, and based on the extracted DAG information it prefetches the objects which will be accessed in near future by compute cluster from the data lake to the caches. It also marks intermediate objects into the directory to avoid the short-lived objects to written back to the data lake.Once the running job is completed, application issues a delete request to remove intermediate objects.

4.6 TradeOffs

In this section, we discuss the different trade-offs we made for D3N and D4N architectures that are necessary to achieve each architecture's design goals.

Redundant Caching vs Design Complexity: D3N by design allows repeated caching of blocks across different layers since the cache eviction/admission decisions are performed locally within the cache pools in each layer. Redundant caching is a natural consequence of consistent hashing with multi-layer design and local cache management decisions, which is a required design choice for achieving simplicity in D3N architecture. For instance, a block replicated in L_2 may also be replicated on higher layers, or a few popular blocks can be replicated across all the L_1 caches flooding the capacity, and preventing caching of slightly less popular blocks. D4N design eliminates redundant replication by avoiding the layered cooperative cache design and maintaining a global cache state. The global state in D4N design allows the implementation of more sophisticated caching algorithms and adjusts its replication of blocks based on access patterns while increasing the complexity of the design.

Supporting Write Requests vs Design Complexity D3N's write-back cache supports only intermediate data sets, where resilience is less critical. For example, when a cache node fails, dirty data on the write-back cache will be lost forever, which is not tolerable for many write requests. On the other hand, D4N deploys a durable write tier using the existing datalake software and maintains the state in the global directory. Having a write tier increased D4N's complexity; however, improving the performance of write requests is critical since write-heavy applications and producer-consumer sharing patterns are very common in many data centers [61, 116].

Focusing on Big Data Analytics vs Wide Range of Applications: When we started working on D3N, object stores were used mainly by big data analytical frameworks with mostly read-only requests and non-critical intermediate datasets.

Therefore, the D3N design focuses on big data workloads and has a single fixed cache management policy (e.g., LRU) that applies to all clusters and frameworks. The lack of durable write-cache and global cache state prevents efficient caching for many applications. For instance, D3N does not support efficient caching for small object requests, which can be dominant in many workloads [61]. On the other hand, recently object stores are used by a wide range of applications(e.g., data analytics, serverless, deep learning). The D4N design enables a greater degree of application-specific specializations; thus can support efficient caching for a broader set of workloads. However, supporting more applications naturally, increase the cost of design complexity in D4N.

4.6.1 Limitation

Lack of Fairness vs Cache Utilization: rack space, storage, power, and network bandwidth. Even though D3N and D4N try to provide a common good by eliminating network bottlenecks, each cluster's individual benefits from D3N or D4N may be different and disproportionate to the resources they provide The goal of both systems is to reduce the load on the data center network and data lake; therefore, the caches aggressively storing data and cache resources used where they are most needed. Fair resource allocation may leave a large amount of underutilized expensive cache resources. Both designs prevent this by trade-off the cache utilization over fairness.

D4N's Static Read/Write Cache Partitioning: D4N statically partitions the cache space between a read and a write cache to reduce the complexity of the design. Allocating a large capacity to write-cache results in a waste of cache resources, and allocating a small capacity can increase workloads' latency significantly because write requests have to wait until the dirty data is flushed back to the datalake. Implementing a dynamic algorithm that identifies or predicts the future access types and

adjusts the sizes of the partitions based on workload behavior will have great value. However, partitioning the cache for read/write requests is not an easy problem; to simplify datalake integration, we avoid adding this complexity in D4N design.

Small Objects in Write Tier: Write tier performs well for large objects since the directory lookup does not impose any overhead. The directory lookup is more costly for small objects than large objects, which was a reported problem in object stores [9, 11]. For instance, when the data size of the object is smaller than 128KB in MinIO [9] and 64KB in Ceph [23], both store object content as part of metadata. A similar optimization can be implemented in write tier as well, D4N can store the object content along with its metadata in the directory.

Chapter 5

Implementation

We have implemented D3N and D4N by modifying Ceph RADOS Gateway (RGW). Section 5.1 gives a brief overview of Ceph Object store and Ceph RGW. Section 5.2 describes the implementation of D3N, and 5.4 describes the implementation of D4N.

5.1 Ceph RGW Overview

We have developed both implementations in Ceph [118], an open-source object-based storage system commonly used to implement data lakes. Ceph is a replicated, mutable object store, accessible via the RADOS protocol [120], and a suite of services implemented on top. In every Ceph cluster, there is a separate Object Storage Devices (OSD) daemon per local storage device. Each OSD handles I/O requests, and cooperates with other OSDs to replicate data, balance the load and to recover from failures. Ceph stores both data and metadata (e.g. users metadata, object metadata) on the OSDs. The Ceph RADOS gateway (RGW) [94] is one of the Ceph's services, providing S3/Swift compatible object storage interfaces, using multiple RADOS objects—much like a file system uses disk blocks—to store a single RGW object and its metadata. Today any framework which supports the S3 or Swift object interfaces (e.g. Hadoop, Spark, Storm, and Flink) can use Ceph RGW.

While the detailed implementation of RGW is outside of the scope of this work, it is a complex system of hundred of thousands of lines of C++ code. It has three layers: `a webserver front-end` (the Beast and the Civetweb library) which accepts

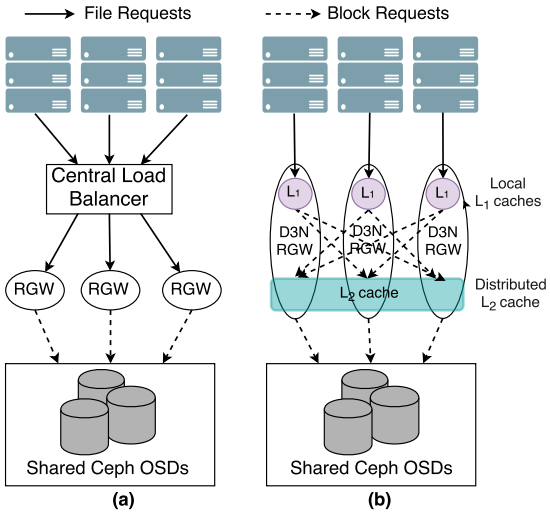


Figure 5·1: Deployment of Unmodified RGW and D3N Architecture in the Datacenter. (a) Load balancing for scale-out RGW deployment. (b) D3N deployment with two-level caching.

HTTP requests, `the radosgw layer` that implements flow control mechanisms per flow, and `the librados layer`, which issues requests to the underlying OSDs. The front end provides a RESTful HTTP API to store objects and metadata, which is that is compatible with S3/Swift APIs. RGW implementation is stateless, with everything persisted in OSDs, it can be easily scaled horizontally.

A typical Ceph/RGW deployment is shown in Figure 5·1(a). Client requests (e.g. using the S3 protocol) are load-balanced (via mechanisms such as round robin DNS) across multiple RGW instances. Each request is handled by just one RGW server, resulting in scaling of aggregate but not single-client performance.

RGW receives S3/Swift requests from clients and stripes them across multiple OSDs (using mechanisms provided by the core CEPH library, librados [8]).The requested object splits into fixed-size block (e.g. 4 MB by default). RGW forwards the entire fixed-size block (e.g. 4 MB by default) to an OSD, chosen by the CRUSH algorithm [119], in turn Ceph OSDs store the block either with replication or erasure coding. Ceph uses **primary-copy consistency** model [26, 120] where the primary OSD coordinates the other OSDs for replication and does not respond the application

with acknowledgment until all replicas have been committed to cache nodes. This way, the replication process ensures all replications are at the same state before any other object operations can be processed. Ceph has atomic object operations, and all the operations (read and writes) for the same object are directed to the same primary OSD, and the primary OSD executes them linearly.

RGW has a window of 4 outstanding requests per connection (16MB), limiting single-stream throughput to no more than four times that of a single device (e.g. disk, in capacity-optimized deployments), but with sufficient connections will spread the load across all devices in the cluster.

We originally consider two alternative approaches 5.3, but at the end we found that it was natural to integrate D3N and D4N into RGW code base. Integrating the cache logic within the RGW code base considerably simplifies the development and implementation of proposed architectures. For instance, RGW provides an S3-compatible authentication mechanism and access control list (ACL) policies for buckets and objects, implements object fragmentation, and flow control mechanisms to order the incoming and outgoing requests, and limits the number of outstanding requests per flow. These features simplify the implementation of the cache architecture, where corner conditions are handled by falling back to the behavior of unmodified RGW.

5.2 D3N Implementation

Our prototype, D3N, implements two levels of cache without a cache partitioning algoritm, and consists of modifications to the Ceph RADOS Gateway (RGW), allowing use by any framework which supports the S3 or Swift object interfaces (e.g. Hadoop, Spark, Storm, and Flink). D3N is integrated within the RGW without changing any client interfaces. The implementation added or modified 2,500 lines of code. We implemented all caching functionalities and also modified the `GET Object` routine in

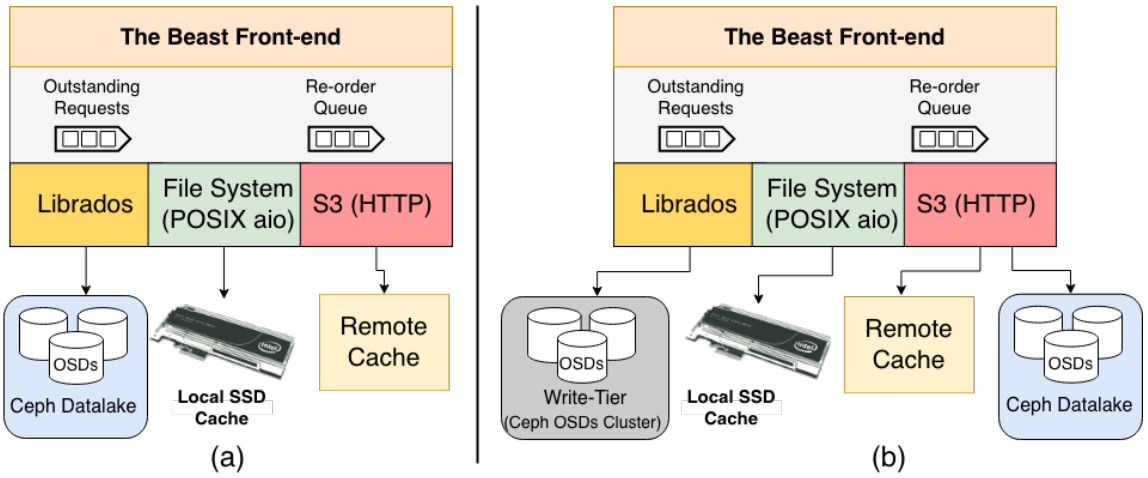


Figure 5·2: (a) Modification to RGW for D3N, (b) Modification to RGW for D4N.

the RGW, therefore reading or writing an object from/to the cache is not different than reading/writing an object from/to the data lake. It implements a L_1 and L_2 cache, storing cached data in 4 MB blocks as individual files on an SSD-backed file system. The final prototype has been upstreamed into the Ceph RGW code base with the assistance of our industrial partners (Red Hat) [19].

All client requests are routed to a nearest D3N L_1 cache, where each block in a request is identified by its object ID and offset, and stored as a file on a local file system backed by striped SSDs (see Figure 5·1(b)). As shown in 5·2a, two additional back-ends are added to RGW for retrieving the blocks: local storage (SSD) as L_1 for local cache access where cache hits are read from files, and recursive D3N as L_2, where misses are redirected recursively to another D3N for L_2 lookup via S3 requests on behalf of the client (using client's credential). Data retrieved from L_2 is also cached on local storage. The L_1 and L_2 caches are *unified*: one copy of a block is stored, although layer membership is tracked for eviction purposes.

Several alternatives for local file I/O were evaluated—memory mapping, GNU POSIX `aio`, and native kernel asynchronous I/O (libaio)—and POSIX aio was used due to its performance in our tested configuration. Native file system I/O was fast

enough to saturate our 40 Gbit network; thus SPDK [111] etc. were not considered.

As discussed in the architecture section, all requests from clients go to the nearest L_1 cache by the help of a local DNS server and anycast solution. We implemented `Anycast (RFC 4786)` [17] in the existing MGHPCC's Brocade bifurcated fabric using Brocade Fabric Virtual Gateway along with virtual routers, virtual Ethernet devices, and layer 2 VLANs. This anycast design allows clients to access the nearest cache node via a fixed IP address. If a cache node fails, the clients are directed to redundant cache nodes transparently.

Client caches the IP address of nearest L_1 cache for a certain period of time and during these time period client directly forward all his requests to L_1 cache. For any IO, as shown in figure 5·1(b), clients send requests to their local L_1 , which breaks the request into 4 MB blocks and handles each independently.

Optimizations: There are a number of optimizations that we considered but did not implement. First, we are currently use the same S3 interface to talk between cache nodes as clients talk to cache nodes. We could have added a new interface between cache nodes to handle L_2 requests, potentially eliminating some of the overhead inherent in handling client requests (e.g. authentication). Since we are forwarding large requests (4MB) the overhead of S3 authentication should have much overhead.

Second we could have improved efficiency for accessing the SSD and network by using SR-IOV-based techniques such as DPDK[111] or SPDK[111]. Finally, more advanced replacement policies, such as segmented LRU could be added to make more informed eviction decisions. Our implementation currently does none of these optimizations, however as we will show in the evaluation section 6.3, our implementation is sufficient to achieve good performance and saturate both the network link and the bandwidth of dual NVMe SSDs.

The absence of sophisticated implementation techniques has allowed us to imple-

ment these changes in a limited number of lines of code, as well as making it possible to submit these features for review and enable the changes to be upstreamed by the Ceph community. We do, however, expect to need to further optimize the code both to reduce the (fully used) occupancy of our server processors, and to exploit future higher performance networking and storage devices.

5.3 Alternative Approaches:

Initially, it was not clear to us to have a complete implementation of this complex storage service, therefore we investigate two existing off-the shelf alternative approaches: `HTTP Cache Varnish` [14] and built-in Ceph support for SSD tiering. The first approach, using a reverse proxy, initially seemed like a straightforward way to implement L_1 cache. Since all the S3 requests are standard REST requests, we though off-the-shelf web cache can work. We evaluated the Varnish reverse proxy cache, which can use consistent hashing to distribute requests to a back end cluster. Performance was, unfortunately, very poor achieving no more than 25% of the read throughput of a single NVMe device. Moreover, with very large requests, the object-at-a-time operation of the cache resulted in long delays and frequent timeouts, making it impossible to run any of our micro-benchmarks. An additional advantage of our D3N-RGW implementation is that requests are for blocks rather than entire objects; which supports better repeated access to partial objects and distributes work across multiple D3N-RGW for even a single request.

Second approach was using the `Ceph Cache Tiering` [41], which was also the original inspiration for this work. Ceph Cache Tiering allows data to be promoted to higher-speed storage when accessed. The design of tiering was problematic as it is only able to promote or demote the entire placement groups which contains data from multiple unrelated objects. Although this failing is specific to a particular design,

there are generic issues with a tiering approach such as the overhead of migrating objects in their entirety across tiers and locating the tier that objects reside in (See 3.1.1). Even tiering is implemented properly, it is not clear how to extend a tiering solution to address network bottlenecks.

5.4 D4N Implementation

We implemented D4N by modifying the previously developed D3N prototype with the capabilities needed to support specilizations for diverse range of applications. Similar to D3N, D4N is integrated within the RGW without changing any client interfaces. The implementation added or modified 14000 lines of C++ code to RGW code base. D4N requires much more modification and implementation than D3N. We implemented all caching functionalities and modified the `Get Object`, `Put Object`, `List Bucket`, `Get Object Info`, `Delete Bucket` and `Delete Object` routines in the RGW code base, and we added new functionalities such as retrieving object metadata from the data lake, directory operations, and additional caching functionalities (e.g., cleaning dirty objects, copying objects from write tier to the data lake, prefetching object, moving object to another rgw cache).

As shown in 5·2b, a new backend is added to previously developed D3N-RGW: a write tier for redundantly storing the newly created data, which is accessed via CEPH's core library librados [8].

To implement a write tier, we use the existing Ceph OSD code, and deployed a Ceph OSDs cluster as a write tier instead of implementing a durable and reliable write-back cache from scratch. Using the existing Ceph OSD code reduces the engineering and development effort significantly, because it has already support various erasure coding algorithms and replication to provide durability, maintains data consistency and cleanliness, and provide a mechanism for failure detection and recovery.

We maintain the local storage (SSD) as read cache for serving read requests from the underlying file system, and keep the S3 interface to access recursively to remote D4N caches. The read cache and write tier store data in 4 MB blocks and object granularity respectively on an NVMe SSDs. In D4N-RGW implementation, we switched from librados to S3 protocol to talk to data lake (Ceph OSD clusters should have additional RGW up and running).

The shared directory is deployed as Redis datastore [97], which is a reliable, durable and distributed in-memory key-value store and capable of dynamically scaling. The Redis keyspace is sharded across cache nodes, and keys are mapped to Redis instances via consistent hashing. Redis provides a leasing mechanism [52] over per key which allows multiple RGW instance to operate with shared directory in a mutually exclusive way.

We co-locate a Redis instance with a D4N-RGW on every cache node. D4N-RGW code uses cpp-redis [5], redis client library written in C++, to implement directory operations, which queries to redis either to retrieve, insert, update or delete a key. In the Redis, we use `Redis Hashes` data type, which are maps between string fields and string values, which allow us to store as much as data for a particular given key. Redis stores information per object and block bases, and per cache node.

Finally, we have implemented two specializations. We implemented *small object packing*, which is implemented by modifying the D4N-RGW code. Small objects ($<$ 4MB) are stored with triple replication on the write tier to avoid the overhead of erasure coding (since objects are small encoding and decoding overhead might be expensive). D4N-RGW coalesces writes within batches before writing to data lake. It packs multiple small objects into a large object (32MB) and stores the mapping in the Redis directory. The prototype has not implemented the persistence of these mappings to the data lake, and the garbage collection for small object packing.

We modified previously developed Kariz [20] cache management tool, which is implemented in Python. By adding only 100 lines of code, we allow Kariz to modify the Redis directory and pin intermediate datasets. Kariz code implemented directory operations uses redis-py [10], redis client library written in Python.

Chapter 6

Evaluation

The main goals of this dissertation is to enable the integration of a high-performance *cooperative cache* into modern data centers' immutable data lakes. To evaluate this, we deploy our prototypes of D3N and D4N in the MGHPCC [13], and we experimentally compare the performance of D3N, D4N, and the state of art cluster cache, Alluxio [25].

Through evaluating D3N and D4N, we aim to show that our implementations fulfills the following objectives: i) improve the performance of applications, ii) reduce demand on the data lake and data center network, iii) automatically adjust to access patterns, iv) provide a support for a broad set of applications via specializations (only for D4N).

In Section 6.1, we highlight the characteristic of real world traces. Our analysis shows tremendous amount of reuse (up to 96%) demonstrating our prototype can have significant value in real data centers. We find that over the lifetime of the trace other files become hot for periods of time and the re-use pattern is complex for some files suggest the value of a dynamic caching mechanism for adapting the workloads.

Section 6.3 shows the base performance for D3N and D4N and shows the overheads of implementation and the maximum performance gains achievable by D3N and D4N prototypes using micro-benchmarks. We find that both implementations are able to fully saturate the read speed of dual NVMe SSDs and the bandwidth of a 40GbE NIC, result in performance on cache hit (5 $\times$) faster than a 90 spindle Ceph cluster

accessed, the caches add no overhead on cache misses, and the directory lookup in D4N prototype has a negligible overhead for large (4MB) objects.

Section 6.4 evaluates the adaptability D3N and D4Nto the changes in the access pattern using macro benchmarks. D3N's prototype does not implement the cache partitioning algorithm, therefore we evaluate the algorithm using simulation. Simulation result shows D3N's novel partitioning algorithm dynamically adapts the fraction of cache devoted to local vs. global accesses at each cache node, showing gains of up to 30% against. fixed allocation in adapting to different access patterns and in responding to network contention. On the other hand, the GWF algorithm is implemented in D4N prototype, and we demonstrate that D4N adapts replication to the working set of the demands; creating fewer number of replica if the working set size is larger than the cache capacity, and more replicas if the working set size is small. Our results are verifying our hypothesis for D4N prototype that the benefit of flexible object placement using a directory outweigh the overhead of directory access.

Section 6.5 experimentally demonstrate the overall performance of D3N and D4N and their adaptability for realistic workloads. By replaying a realistic workload, we show that D3N (non-adaptive) achieves significant performance improvements for realistic workloads—up to a 3x reduction in runtime vs. uncached when bandwidth-constrained. We also show how D4N maximizes local hits and minimizes data lake accesses for realistic workloads, and outperforms D3N (non-adaptive) and Alluxio.

Section 6.6 evaluate D3N and D4N under real, computation-based workloads by running a PIG/Hadoop-based workload. We demonstrate that D4N can reduce cumulative job run time by large amount (e.g. 30%), reduce demand on the network and the backend data lake.

In section 6.7 we show that application-specific specializations at the caching layer can achieve dramatic improvements in application performance as well as a significant

reduction in data lake workload. A macro-benchmark, based on a publicly available storage trace demonstrate that small object packing specialization won't impact the performance of small object reads or writes, but reduce the load on data lake storage servers by 400% over the case of no write-back cache and 200% over the case without object coalescing. We also demonstrate the significant value of using the application hints for cache-specializations; by adding few lines of code to the existing Kariz code base can improved the run time of DAG based workload by 20%.

6.1 Trace Analysis

To drive our evaluation and understand the properties of real workloads, we analyze publicly available 2010 Facebook trace collected from 3000 nodes Hadoop cluster and 2017 Two Sigma [107] [1] trace collected from the 257 cache nodes which are serving to analytical clusters.

In addition to Facebook and Two Sigma trace, we also use the following two traces for evaluating our prototypes; a confidential trace[20] from a 300 node production cluster that is running Hive/Hadoop [112],Spark [125], native MapReduce [106], and streaming jobs, and recently released publicly available IBM object storage traces [49]. Table 6.1 lists the traces' detailed characteristics. While these may not be fully representative of access patterns to data lakes, they are the best large-scale traces described in the literature.

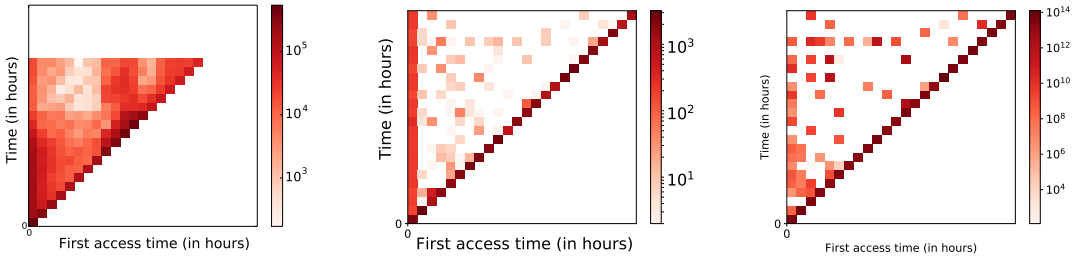
6.1.1 Access Patterns

Measurements from Two Sigma for their data lake shows around a 96% re-access rate across a trace representing 10 PB of data access, validating the first assumption and suggesting that a caching solution will be very effective. Previous studies which

[1] We are working with our partner to make the trace publicly available.

	Facebook	**Two Sigma**	**Confidential**	**IBM**
Trace ID	TR-FB2010	TR-TS2017	TR-CP2021	TR-IBM2019
Public	Yes	No	No	Yes
Year	2010	2017	2021	2019
# of days	1	12	1	7
# of clusters	1	-	1	-
# of nodes	3000	257	300	-
# of input files accessed	25428	185M	331	149M
# of unique input files accessed	17158	5936K	43	-
input bytes accessed	1097 TB	245 TB	2448 GB	161 TB
output bytes written	337 TB	-	-	188 TB

Table 6.1: Characteristics of the traces that we analyze to understand the properties of real workload. We also use these traces to drive our evaluation.



(a) Two Sigma **(b)** Facebook (files) **(c)** Facebook (bytes)

Figure 6·1: Re-use patterns for Two Sigma and Facebook traces. Files are identified by time of first access (X axis), actual access time is on Y axis; color intensity indicates access count.

analyze the IBM object storage traces [49] and a confidential trace from a production cluster [20] also verify this assumption.

Next, to understand their re-use pattern, we analyze TR-FB2010 and TR-TS2019 traces and present heatmaps of their access patterns. We note that both trace follows a Zipf-like distribution. Figure 6·1a and Figure 6·1b are heat maps showing access counts to each file over time, respectively for each trace, with files identified by their time of first access. Both figures show that a number of files are accessed early and remain popular throughout the trace while other files become hot for periods of time at intermediate points in the trace; suggesting the value of a dynamic caching

mechanism. In addition a second heat map (Figure 6·1c) for the Facebook trace shows access by byte rather than by file number. The value of block-level rather than file-level caching is seen in comparing Figure 6·1b and Figure 6·1c. Regions where byte re-use is significantly less than file re-use (e.g. along the Y axis) indicate partial rather than complete reads of files.

Our trace analysis validates three key assumptions behind the D3N and D4N designs which are;

1. there is sufficient re-use of data from a data lake to make a caching solution valuable,
2. the re-use pattern is complex and popularity of objects change over time,
3. it is valuable to cache data at the block rather than entire file basis.

6.1.2 Workloads

We evaluate our prototypes by replaying real-world traces described in the table 6.1. Since the traces are collected from a much larger compute environment than our environment, we replay part of some of the traces and scale down them to run on a smaller-size cluster. We preserve the original arrival order of the jobs/requests in all workloads to emulate access to a shared data lake.

We use trace TR-FB2010 to examine overall performance for workloads with more realistic distributions (Section 6.4 and 6.5.2), with results unaffected by computational overheads. It was the only publicly available trace during the evaluation of our prototypes, thus enabling reproducibility. TR-FB2010 contains jobs' input file sizes and the number of bytes they wrote as an intermediate and output. Unfortunately, TR-FB2010 trace does not contain mapper information needed to determine request locality; therefore we generated synthetic locality information.

To be able to test the trace over a smaller-sized cluster, we scale down the original file sizes by half (TR-FB2010-1) and by a factor of 10x(TR-FB2010-2). [2] Then we replay a part of the trace with 75% percent re-use (a low re-use ratio compared to TR-TS2017). **TR-FB2010-1** workload includes 853 jobs processing 40 TB of data with a 10.1 TB footprint; 8 TB of the data is repeatedly accessed while 2.1 TB of the data is accessed only once. TR-FB2010-1 considers only read traffic to emulate access to a shared read-mostly multi-institutional data lake. **TR-FB2010-2** workload considers both read and write traffic; 1.5 TB of intermediate data and 540 GB of output data is written.

We use **TR-CP2021** trace to understand the performance of both systems using real workloads, where the performance is limited by computation as well as I/O speed(Section 6.6). TR-CP2021 is an hour-long trace, including statistics (query submission rate, DAG structure, data access, data re-use) of submitted queries for Hive/Spark and Spark jobs. We use TR-CP2021 trace to construct a synthetic workload. To simplify evaluation and preserve the confidentiality of partner data, trace queries and DAGs were replaced with queries (and corresponding data) from TPC-H [18] (translated into Pig Latin [87]), modified to mimic the original queries. More specifically, for each query in the confidential trace we compare its structure and selectivity with the 22 TPC-H queries and compute cosine similarity, then map to the most similar query. The resulting synthetic queries thus closely mimic the query structure, data re-use rate, access rate, and selectivity of the logged queries. We scale down file sizes and query submission rate by 10x to fit our experimental environment. The trace includes 331 jobs that access over 43 unique input data sets, with the size of input objects varying from 1 GB to 256 GB.

To evaluate the small object specializations(Section 6.7), we use recently published trace, **TR-IBM2019**, which captures the object storage accesses of a single

[2] Prior work has shown this has little or no impact on performance [28].

tenant. TR-IBM2019 trace was the only publicly available trace available with small IO operations, where the object size varies from tiny requests (< 1KB) to large requests (> 1GB). The trace contains 98 different traces which are captured from variety of workloads such as SQL queries, Deep Learning, NLP, Spark data analytic, and document and media servers although they are not distinguishable in the traces. We use trace-1 as a workload that is dominated by small object writes (97%), and the trace includes only requests that access objects less than 4MB.

6.2 Experimental Setup

Infrastructure: Experiments are executed on a private academic data center, where racks interconnected in a partial-mesh topology, with the equivalent of 4×40GbE inter-rack links on each top-of-rack switch and an average distance of 3 hops between racks. Hardware and software configurations are shown in Table 6.2 and Table 6.3 respectively.

	Compute Nodes	**Cache Nodes**	**Data Lake Nodes**
`Number`	48	4	9
`CPU`	1x Intel E5-2690	2x Intel E5-2699v3	2x Intel E5-2660
`Ram`	128 GB	128 GB	128 GB
`Disk`	1x 500 GB HDDs 5400 RPM	2x Intel P3600 1.5 TB NVMe SSDs	7x 600 GB HDDs 7200 RPM
`Network`	10Gb/s	2x40Gb/s	10Gb/s

Table 6.2: Hardware Configuration

	Ceph	**Redis**	**Alluxio**	**Hadoop**	**Pig**
`Version`	15.2.7 Octopus	6.0	2.7.1	2.10.1	0.17.0

Table 6.3: Software Configuration

For D3N experiments, compute and cache nodes are located on 2 racks; on each rack we allocate 1 cache node with 2 NVMe SSDs each, and 8 compute nodes and the storage cluster is comprised of 10 storage nodes with 90 HDDs. For D4N experiments, we use 4 cache nodes with 2 NVMe SSDs each, 48 compute nodes, and a 9-node data

lake with 63 HDDs. (Due to the hardware limitations we scale down the size of data lake cluster from 90 HDDs to 63 HDDs for D4N experiments.)

6.3 Base Performance

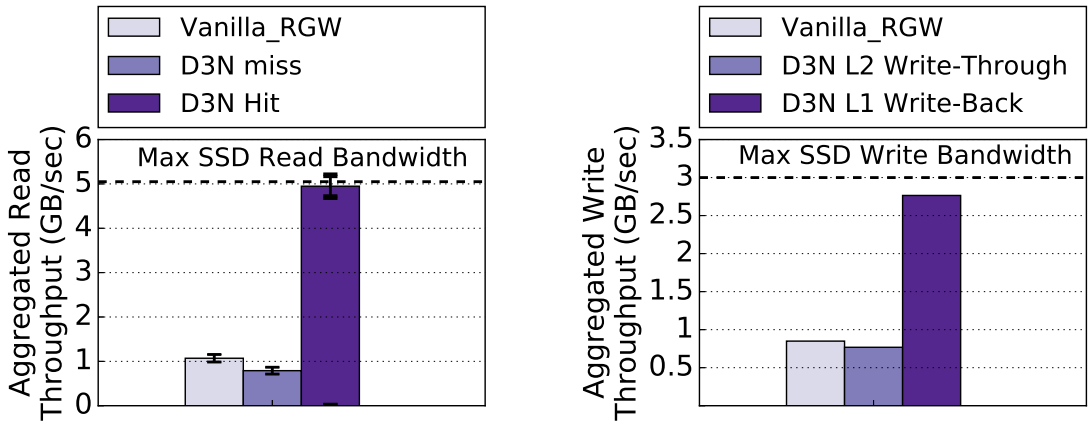
We use a series of micro-benchmarks to present the base performance of D3N and D4N properties under controlled circumstances. First, we evaluate the maximum performance gains obtainable using the D3N prototype and D3N's impact on miss performance. Next, we compare D4N's base performance against D3N and the state-of-the-art cluster cache, Alluxio [25].

6.3.1 D3N Performance

We evaluate and compare the base performance, hit and miss throughput, of D3N prototype with that of the *unmodified (Vanilla) RGW* that represents the direct data lake access. We submit read and write requests to the Ceph data lake via D3N and a "Vanilla" RGW. Read and write requests are submitted using Linux's `curl` tool, a high-performance HTTP client. In these experiments, for Vanilla RGW, the Ceph OSD buffer caches are flushed before each test run to ensure that data is fetched from the 90 OSD disks rather than the RAM of the OSD servers.

L1 and Write-Cache Performance: We evaluate L_1 (local cache) hit and miss, and write-cache performances. To summarize, micro benchmarks show that the implementation of D3N can support per-cache read speeds of 5 GB/s, fully exploiting the SSDs and NICs in our system, and adds negligible overheads on cache misses, and the write-back caching is able to saturate the write performance of dual NVMe drives.

For this analysis, we use eight compute nodes, each issuing eight concurrent 4 GB read or write requests using the curl benchmark. Compute nodes are co-located on the same rack with cache node.



(a) Read Throughput

(b) Write Throughput

Figure 6·2: a)D3N hits improve the read throughput 5×, saturating the dual NVMe SSDs and a 40 Gbit NIC. b) Write-back improves the throughput by ∼3× and write-through incurs a small (∼10%) overhead.

Figure 6·2a compares the read throughput of D3N L_1 cache with the Vanilla RGW. As seen in Figure 6·2a, our implementation is efficient enough to saturate the dual NVMe SSDs and the 40 Gbit NIC; meaning L_1 hit throughput almost 5× of higher than Vanilla RGW. The black dotted line (5.0 GB/s) represents the maximum read rate of the cache server SSDs as measured with "`dd`"; essentially the "speed of light" for our experiments. It is also coincidentally the throughput of the cache server's 40GbE NIC. Also, L_1 misses on D3N (which incurs the overhead of storing missed data to SSDs in both L_1 and L_2) has ∼26% impact on the read throughput. This overhead seems easily justified if it results in subsequent hits.

Figure 6·2b compares D3N write performance using write-through and write-back to vanilla RGW. Write-back mode improves the write throughput 3×. Our implementation is efficient enough to saturate the capacity of the dual NVMe SSDs. Write-through, which blocks until the write has completed to Ceph, incures additional overhead over vanilla RGW of traversing multiple D3N caches and writing the data to the SSDs. As we see this overhead is only around ∼10%. While the default policy

right now is write-around (given the experimental nature of our changes), we expect users to use write-back for intermediate data sets, where resilience is less critical, and write-through in all other cases given the modest cost and the significant value for subsequent hits.

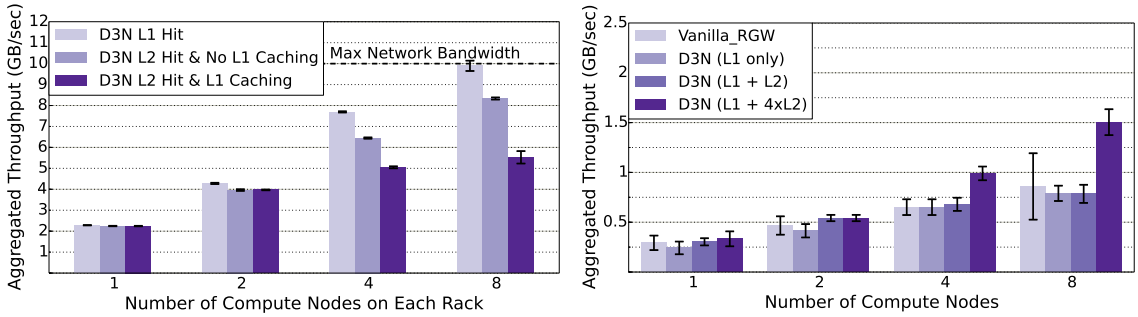


Figure 6·3: a)Comparison of L_1 vs L_2 hit performance
Writes to L_1 cache are seen to incur significant overhead in the case of L_2 hits. b)Read miss with multiple D3N caches.Traversing multiple caches (L_1 + L_2) is seen to have negligible effect on performance, while striping across 4 L_2 D3N caches is nearly twice as fast as unmodified RGW.

L1 vs L2 Performance: We run several experiments with micro-benchmarks to understand the performance and overhead of accessing L_2 (remote cache) in details, and compare the cost of accessing a local cache vs a remote cache. To summarize, in our environment (with a very high speed closely coupled data lake) D3N offers a 5× higher throughput on L_1 cache hits and 3× higher throughput on writes with a write-back policy (saturating the SSDs on cache servers in both cases) and incurs modest overhead on cache misses (∼26%) and write through (∼10%).

To compare L_1 and L_2 hit performance, three cases were measured and compared: (a) L_1 hit, (b) remote L_2 hit with L_1 disabled, and (c) remote L_2 hit with L_1 enabled. Two cache servers are used, one on each of two racks, with 1 to 8 compute nodes on each rack (16 nodes total) and 8 concurrent requests per compute node.

In Figure 6·3a we see that for L_1 hits, 8 compute nodes per rack (the right most

bar) are able to saturate both SSDs and 40Gbit NICs on each of the two cache servers, with an aggregate throughput of 10 GB/sec, while the overhead of remote L_2 requests reduces throughput slightly, to 8.5 GB/s in the same case. In this case the overhead of storing data on L_1 nodes is significant, dropping performance to 5.5 GB/s; however doing so allows additional full-speed L_1 access to this data.

To evaluate L_2 miss performance we measure four cases: (a) a single vanilla RGW server, (b) a single D3N L_1-only server, (c) two separate D3N servers configured as L_1-only and L_2-only, with cascading cache misses, and (d) a pool of four L_2 cache servers, with one serving as a co-located L_1 server. As before, eight concurrent 4 GB requests are made from each of 1 to 8 client nodes.

In Figure 6·3b we see equivalent performance for cases (a), (b), and (c), indicating that—even in the case of two cascaded cache misses—the overhead of D3N is not significant in comparison with the throughput limits of the backend Ceph cluster. In the final case (d) requests to the backend Ceph cluster are striped across four L_2 caches; this is seem to result in significantly *higher* throughput than the vanilla case, perhaps due to an increase in the number of connections to the backend Ceph cluster.

6.3.2 D4N Performance

We evaluate D4N prototype using *s3-benchmark* [101], a well-known and widely used benchmarking tool, to measure the overheads and the maximum performance gains, and compare its performance against *D3N*, *Alluxio*(ALL) and direct uncached access to the data lake, referred to as *Datalake* in figures. We configure the Alluxio to cache the data on a cache node whenever the data is accessed.

In these experiments we created a 48 nodes compute cluster running across 4 racks. Compute nodes submit 128 MB read and write requests, with 10 threads per compute node; the total amount of data transferred per experiment is 200 GB.

To summarize our result, we found that, like D3N, the implementation of D4N

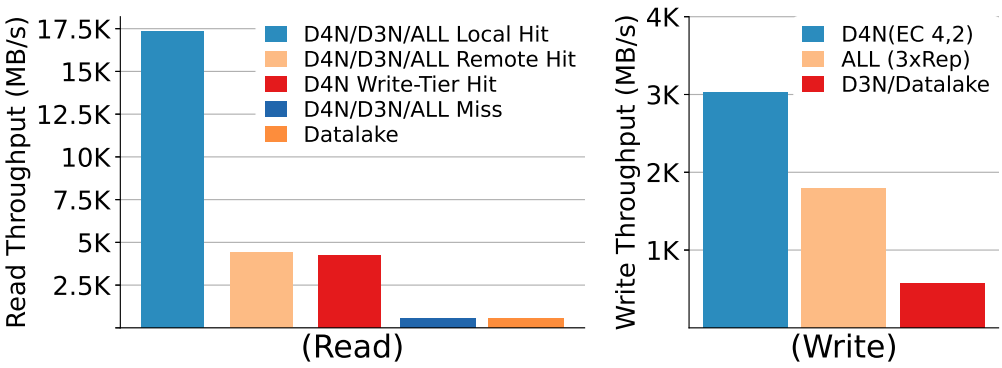


Figure 6·4: Cache Hit and Miss throughput of D4N, D3N and Alluxio. All three caches are efficient enough to saturate SSDs and NICs speed. D4N's remote cache and write tier hit throughput saturates the available network bandwidth between cache servers. D4N's write tier throughput is 1.6× higher than Alluxio due to performance advantage of the erasure coding against triple replication.

can exploit the full bandwidth of the SSDs and NICs in our system like D3N, and the directory lookup adds negligible overheads on cache hits and misses, the D4N's erasure coded write-tier throughput is 1.6 × higher than triple replicated Alluxio.

Read Performance: Read results are seen in Figure 6·4a. Upon a local read cache hit, all three systems (D4N, D3N, Alluxio) performed comparably, reaching speeds of 4.3 GB/s (35Gbit/s) per cache node and nearly saturating both the compute-to-cache link (40 Gbit/s) and the paired SSDs (nominally 5.2 GB/s). When all reads hit in remote cache, speed is limited by the 10 Gbit/s per-cache bandwidth to all other caches, and per-cache throughput is 1.1 GB/s (8.8 Gbit/s). Read hits in the erasure-coded write-tier have similar performance, as data is retrieved from remote cache nodes. Read miss performance is the same for all three caches, and is the same as for direct data lake access, indicating miss processing overhead is negligible relative to data lake access speed.

Write Performance: Figure 6·4b shows the aggregate write throughput of the write tier, which stores data using a (4,2) Reed-Solomon erasure code [98], compared

with Alluxio (3x rep) and direct data lake writes. (D3N does not support a reliable write-cache therefore we show direct data lake writes rather than show to the non reliable write-back cache.) D4N write-tier throughput is 2.6 × higher than direct writes to the data lake, and 1.6 × higher than Alluxio, the latter due to the reduced replication cost of the erasure code compared to Alluxio's triple replication. We note that directory look up and update overheads are negligible in both of these experiments. We also note that previous study by Rasmi et al. [96] implements erasure coding using Reed-Solomon [98] codes on Alluxio, however current publicly available Alluxio implementation implementation use only replication for data durability.

6.4 Workload Adaptability

We show how D3N and D4N adapt changes in the access patterns by evaluating each architecture's proposed novel cache management algorithm. D3N's dynamic cache partitioning algorithm was never implemented in our prototype, therefore we evaluate the algorithm using simulation. On the other hand, we implemented GWF algorithm in the D4N prototype, so in practice only D4N prototype adapts to changes in the access pattern. Therefore, the evaluation of D3N for real workloads in next sections(6.5.1 and 6.5.2) does not include the D3N's adaptability algorithm but does include D4N ones.

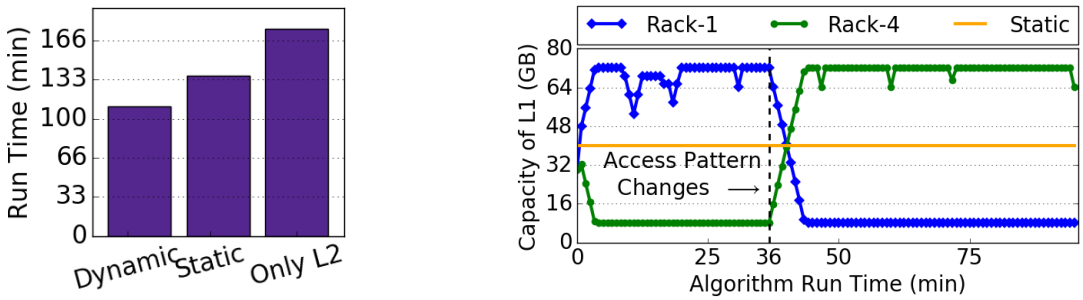
6.4.1 D3N's Adaptability

We evaluate the D3N's dynamic cache management algorithm using simulation. To summarize, our simulation results show that a novel adaptive cache partitioning algorithm using observed throughput and access patterns to optimize cache capacity division between rack-local and cluster-wide data, showing gains of up to 30% vs. fixed allocation in adapting to different access patterns and in responding to network contention.

Simulation: We implemented a cache simulator that mimics the D3N multi-layer cache architecture. The simulator contains 1500 lines of code implemented in Python using the standard `SimPy` simulation library [108].

In our simulation, we assume a data center with 10 racks, each rack containing one D3N cache node and 20 client machines. The aggregate bandwidth between clients and D3N nodes is 50 GB/s (i.e. one 40 Gbit NIC per node) and the bandwidth between L_1 and L_2 is 15 GB/s, and with a rack-to-rack over-subscription of 3.3:1. Each client issues concurrent 150 requests for 4MB objects. Both synthetic traces and TR-FB2010-1 trace(Section 6.1.2)were used. TR-FB2010-1 trace does not contain mapper information needed to determine request locality; therefore we generated synthetic locality information, assuming that a repeat access to a file occurred on the same rack with $p = 0.7$, and from another rack (chosen randomly) with $p = 0.3$. At simulation start, each cache was divided equally between L_1 and L_2; every 1.5 minutes the cache allocation was adjusted by up to 5% of capacity in either direction. (Sensitivity to these parameters was tested, and any combination able to adapt by at least 2% every minute was found to give equivalent performance.) Each experiment has a warm-up phase and run phase. In this section, we only report the results, which are collected during the run phase.

Adaptability to different access patterns: To analyze D3N's reaction to workload pattern changes over in time, we split the TR-FB2010-1 trace and assume that for the first 36 minutes of the trace all requests arrive from Rack-1 and after the 36 minute mark all requests arrive from Rack-4, mimicking behavior when different parts of the data center have high workloads at different points in time. In Figure 6·5(b) we see L_1 capacity for Rack-1 and Rack-4; after the access pattern changes at the 36th minute mark, Rack-4's L_1 capacity starts to increase while Rack-1's L_1 capacity decreases, indicating rapid reaction to workload pattern changes. Figure 6·5(a)



(a) Trace runtime.

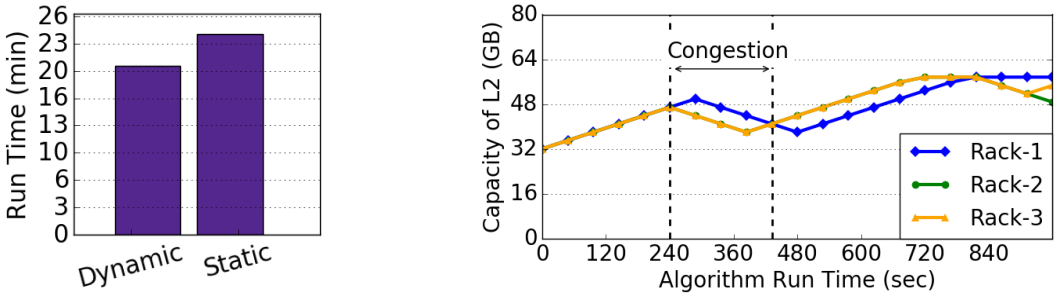
(b) L_1 capacity as access patterns change.

Figure 6·5: Dynamic cache allocation: (a) runtime, static / dynamic / L_2-only; (b) L_1 capacity changes as access patterns change.

compares the overall runtime of the workload under dynamic and static partitioning with a 50/50 L_1/L_2 assignment, and when only a single layer distributed L_2 cache is deployed. Runtime for dynamic partitioning is improved by 19% over static allocation and 36% over a single L_2 cache, respectively.

Adaptability to network load changes: Next we evaluate D3N's adaptability to changes in the network loads. We use synthetic trace, where all racks request the same set of files and the size of the requested files are bigger than the total size of the cache service. Starting from 240th second until the 440th second, we congest the link that ties Rack-4 to other racks and the bandwidth drops from 15Gbits/s to 1Gbits/s. As a result, it becomes costly for other cache-nodes to fetch data from L_2 of Rack-4. Therefore, all racks increased the capacity of their L_1 caches during the congested window.

Figure 6·6(b) shows the L_2 capacity of Rack-1, Rack-2, and Rack-3 before, during and after the congestion. We plot only 3 racks to make the figure more readable. We note that remaining 7 racks follow the same pattern. When the congestion is over, the dynamic algorithm slowly increases the capacity of L_2. As seen in the figure, D3N adjusts the cache capacities as expected. Figure 6·6(a) compares the overall job completion time of the workload under dynamic and static allocation(50/50 L_1/L_2) mechanisms. As shown in the figure, dynamic allocation completes 14% faster than



(a) Trace runtime.

(b) L_2 capacity after network congestion.

Figure 6·6: Impact of dynamic cache allocation of D3N when a network congestion occurs. (a) Runtime of static and dynamic allocation. (b) L_2 capacity with changing network congestion.

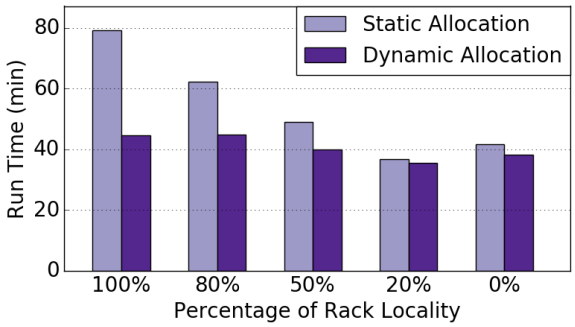


Figure 6·7: Runtime comparison of static and dynamic cache allocation for TR-FB2010 trace under different locality levels.

static allocation. Both figures indicate that D3N quickly reacts to network congestion and adjusts cache sizes towards ideal settings.

Performance under different locality levels: In Figure 6·7 we see job completion time for dynamic and static (50/50 L_1/L_2) allocation for the TR-FB2010 trace benchmark with different locality levels, from 100% (files always accessed at the same L_1) to 0%(access locations are random). Dynamic allocation improves job completion time for high-locality cases, e.g. by 42% and 26% for localities of 100% and 80%, respectively.

6.4.2 D4N's Cache Elasticity

We evaluate how D4N can automatically adapts workload using it's novel cache management algorithm GWF. One of the key features of D4N is its elasticity, enabling cache resources to be automatically used where they will do the most good. GWF exploits the distributed directory, enabling each cache to independently make replication decisions that are intended to automatically adjust cache usage based on the storage working set. To summarize, GWF algorithm automatically adapts replication to the working set of the demands, and D4N outperforms Alluxio even when Alluxio is statically optimized to choose the correct policy for large and small working sets.

We use a simple uniform random workload to compare to Alluxio, which enables applications to specify two strategies: *All-AR(always replicate)* using its "passive replication" feature to always replicate objects to the accessed cache, and *All-1R (one replica)* where data is cached opportunistically by the first cache that retrieves it from the datalake, and not subsequently replicated. Since the implementation of D3N has a fixed policy, we compare D4N to Alluxio with both always replicate and never replicate.

We expected All-AR to be the optimal policy for small working sets where data is fully replicated to all caches, and All-1R to be the optimal policy for large working sets, since data will only need to be fetched from the data lake once. We hoped that D4N would match the performance of each of these optimal policies automatically. As expected, D4N does automatically adjust replication to match the optimal, but as described below, it also outperforms both All-AR and All-1R in all scenarios.

We use a trace consisting of 4800 requests uniformly and randomly directed to 400 unique objects. We configure the per-server cache to 64 GB, and vary the footprint of the data from 50 GB to 200 GB, using the compute cluster and s3-benchmark described above.shows the accesses to the data lake normalized by the size of the

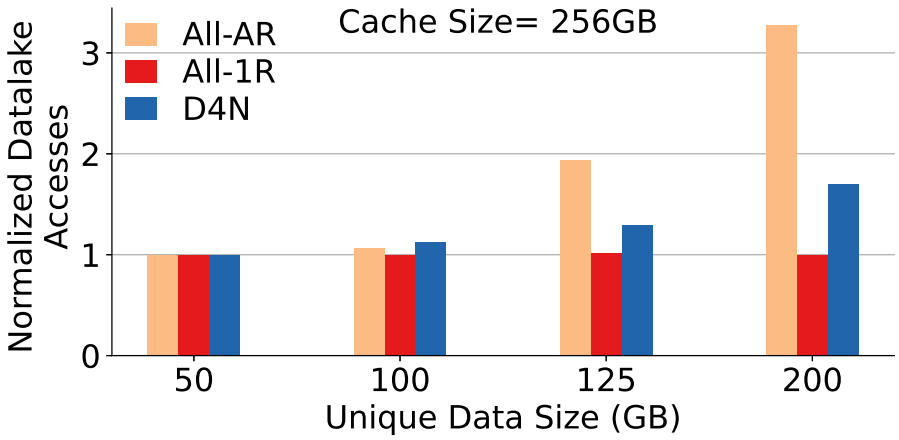


Figure 6·8: Compare D4N and two Alluxio configurations: All-1R replicates a single copy of an object, and All-AR replicates an object upon every request. Results of accesses tot he data lake normalized by the size of the footprint. With small working set all just miss the first access, with large working set All-AR results in many misses.

footprint.

Figure 6·8 shows the accesses to the data lake normalized by the size of the footprint. With the small footprint, all caches only suffer one miss to the data lake. As the footprint increases it still fits into the aggregate cache size, allowing All-1R to continue to only hit the data lake once for each object. All-AR, which aggressively replicates data, suffer $3x$ more misses to the data lake, since the additional replication causes eviction of cached data. For the largest footprint D4N still incurs roughly 60% additional data lake accesses, as it attempts to discern hot and cold blocks within a uniform distribution, giving blocks accessed twice the weight of blocks accessed a single time. Even with this increase, however, it reads less than half as much from the data lake as All-AR.

Figure 6·9 shows the time to complete the experiment as the working set size grows, normalized by the run time using D4N. For the small working set, D4N achieves 20% better performance than All-1R and around 40% better performance than All-AR. The improvement over All-1R is expected: both D4N and All-1R must (slowly) load the data into cache, however once this is done, D4N will replicate it to all caches,

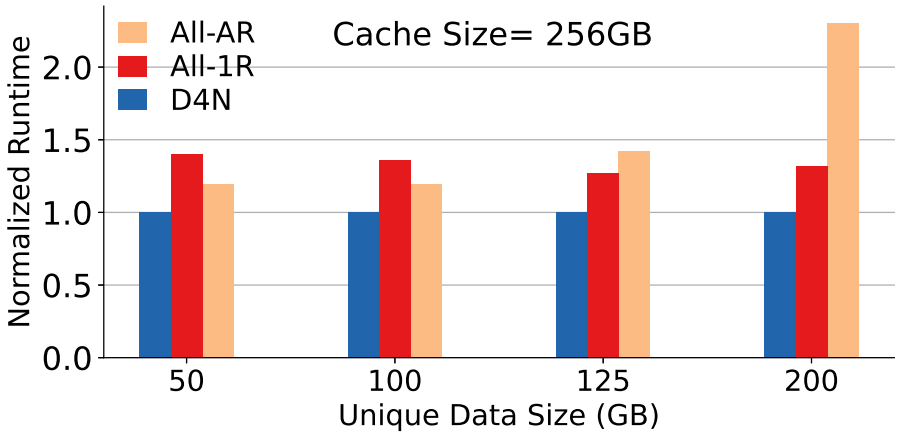


Figure 6·9: Normalized run time for D4N and Alluxio with two configurations where D4N is 1. D4N adjusts the number of replicas based on the global demand, reduce the data lake accesses and has the shortest run time for all scenarios.

enabling an aggregate bandwidth of close to 4×40Gbit/s, while 3/4 of data accesses with All-1R will be from a remote cache, allowing a peak speed of $4/3 \times 10$ Gbit/s. We are surprised that D4N has an advantage over All-AR, and are investigating to determine the performance issue with Alluxio causing this difference.

With large footprints All-AR run time is more than 2.3x slower that D4N, which is expected since, as we have seen from Figure 6·8, it performs many more accesses to the data lake. Surprisingly, D4N also outperforms All-1R, despite its additional accesses to the data lake. We believe that the reason stems from D4N caching data at the chunk rather than object level, enabling the modest number of additional blocks fetched from the data lake to be retrieved in parallel with other data being served from the cache. In addition, the demand on the data lake is still less than the available bandwidth, and hence the advantage of a modest increase in local cache hits with D4N compensates for the additional misses to the data lake.

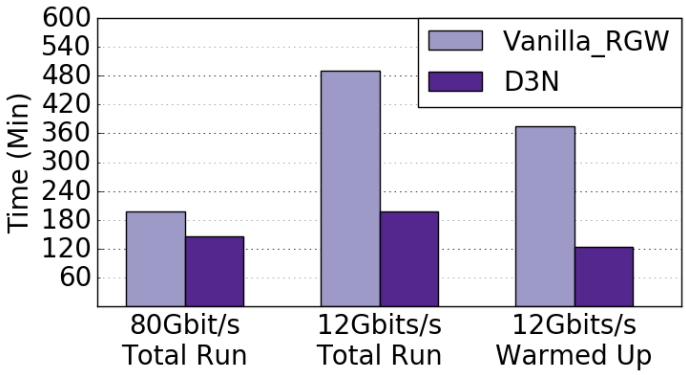


Figure 6·10: Facebook workload runtime: D3N vs. vanilla RGW, full-bandwidth (80 Gbit) and throttled (12 Gbit) network to data lake, full run and after 1/3-trace warmup. D3N improves runtime by 25% with (unrealistically) high data lake bandwidth, and by 4× with a more realistic network.

6.5 Realistic Workloads

In this section, we run macro-benchmarks using a large scale TR-FB2010-1 trace to evaluate the performance of D3N and D4N for realistic workloads.

To summarize the results we had, macro benchmarks show that D3N and D4N capture the enormous temporal locality in data center workloads to greatly reduce the accesses to the data lake and improve the performance up to 3×, and D4N can reduce cumulative job run time by large amount (e.g. 30%), reduce demand on the network and the backend data lake.

6.5.1 Overall Performance for Realistic Workloads

Via macro-benchmarks, we measure the runtime impact of D3N for more realistic read-only workloads. D4N will perform similar to D3N, and as we will see in the next subsection 6.5.2 where we evaluate the workload adaptability of D3N and D4N for realistic workloads.

We create a two layer D3N configuration using two cache servers with a total 5 TB capacity. We use the production (non-adaptive) version of D3N, with a static cache

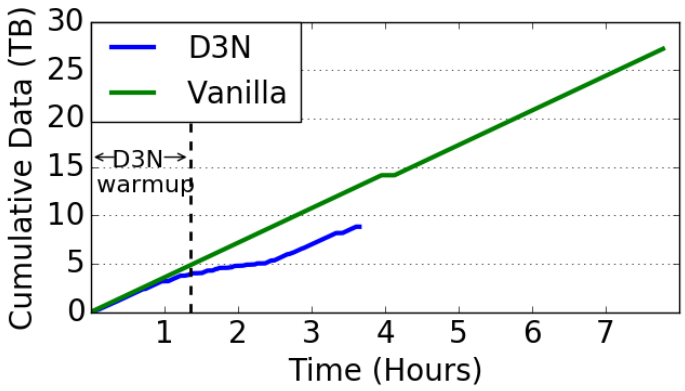


Figure 6·11: Cumulative (sampled) data lake transfers for Facebook workload D3N vs Vanilla, throttled (12 Gbit) network to data lake. For D3N the link is saturated during the warm-up phase due to cache misses, but demand is reduced during the measurement phase due to cache hits; Vanilla takes 3× as long, saturating the link throughput during the experiment.

of 50% to each cache layer. Aggregate bandwidth to the back-end storage cluster was 80 Gbit/s (one 40 Gbit NIC per server); additional experiments were performed with an aggregate bandwidth of 12 Gbit/s, by throttling each cache server-to-storage connection to 6 Gbit/s. Hadoop clients were emulated by a custom tool, generating HTTP requests (with a 512 MB block size) to mimic the S3 requests each mapper would have initiated, allowing all mappers to be emulated from two 36-core nodes with 40 Gbit NICs. Since the trace lacks client node information, requests were randomly assigned to each mapper with no locality, giving conservative results vs. real workloads with non-zero locality.

Figure 6·10 displays the trace completion time with and without D3N for the two network bandwidth configurations. The experiment was divided into two consecutive phases, warm-up and measurement. The warm-up phase consisted of the first 33% of the total trace; performance results are reported for the measurement phase alone as well as for the full workload run time (warm-up plus measurement). With an unrealistic 80 Gbit of bandwidth to the storage pool, performance improved by about 25%. With 6 Gbit from each cache server to the backend (a conservative emulation of

a shared 10 Gbit connection) the full workload and measurement-only times improved by 2.4× and 3× respectively.

With only 75% re-use, we see that the cache is still highly effective, greatly increasing storage system throughput and thus application performance. Throughput also compares favorably with the traditional alternative of manually copying hot datasets into a disk-based cluster-local HDFS system. Due to the huge speed disparity between NVMe and disk, it would take 60 to 80 disk spindles across this cluster to equal the throughput of the two dual-SSD cache servers.

Another benefit is the reduction in inter-cluster traffic. Figure 6·11 we see cumulative data transferred from the back-end storage (sampled using `pbench`) for the 12 Gbit/s experimental configuration. With the vanilla RGW, the Ceph link is nearly always saturated, with 23 Tb of observed data transferred after the warmup phase, while D3N transfers about 5 Tb, more than a 4× improvement (e.g. allowing 4 times as many analysis jobs to share the storage backend or bottleneck links).

6.5.2 Adaptability of D4N, D3N vs Alluxio to Realistic Workloads

To examine D4N's overall performance for workloads with more realistic distributions, we replay TR-FB2020-2 trace with results unaffected by computational overheads. We measure overall runtime, as well as the total volume of local hits, remote cache hits, and data fetched from the data lake due to complete misses. We compare D4N with D3N, All-AR(Alluxio always-replicate), and All-1R (Alluxio single-replica). In the case of D3N, we use the production (non-adaptive) version of D3N and do not enable the write-back cache (upstream community has not been willing to accept model of unreliable storage of intermediate data, therefore production version of D3N does not have a write-cache.).

Clients were emulated by a custom tool to generate S3 requests with a 128 MB block size, mimicking mappers and reducers with zero CPU overhead. This allowed

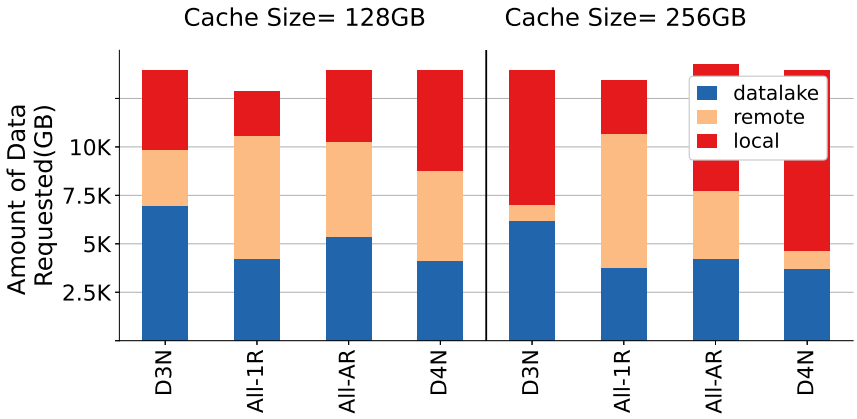


Figure 6·12: Local/remote cache and data lake accesses.

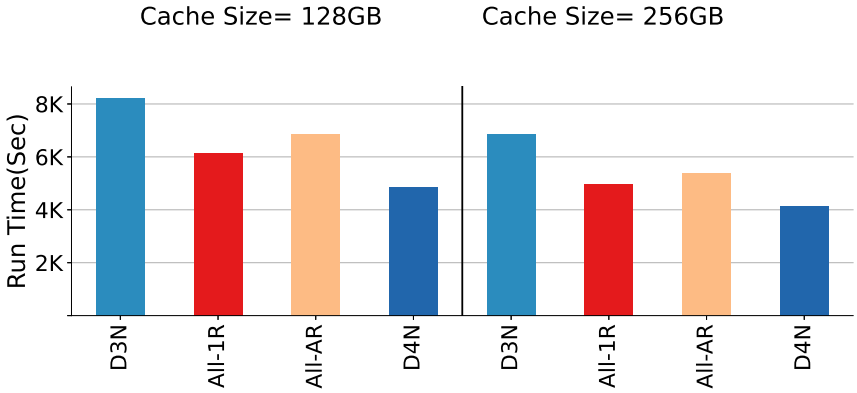


Figure 6·13: Run time for D4N, D3N, and Alluxio configurations, TR-FB2020-2 trace, no intermediate data. D4N maximizes local hits and minimizes data lake accesses, achieving the lowest runtimes.

all mappers and reducers to be emulated from only 24 client nodes, 6 per cache. Since the trace lacks locality information, requests were assign to mappers in round robin order. Aggregate cache capacity was scaled from 128GB to 512GB, i.e. 32GB to 128GB per cache server.

Our first experiment replays only the input reads and final output writes; results are seen in Figure 6·12 and Figure 6·13. Only data for 128GB and 256GB experiments are reported, as little difference was seen between strategies for the 512GB case. In each case, D4N accesses to the data lake (i.e. misses in both local and remote caches) are as low or lower than All-1R, while local hit volume is as high or higher than All-AR, and the resulting runtime is better than either.

The next experiment replays the entire trace including intermediate output, with an aggregate cache size of 640 GB. Once intermediate data is written, it is immediately read by the same compute node, and then deleted. In all cases with write caching—i.e. Alluxio and D4N—the temporary data is deleted before it would be written back to the data lake, and only the final output data (540GB) is written back.

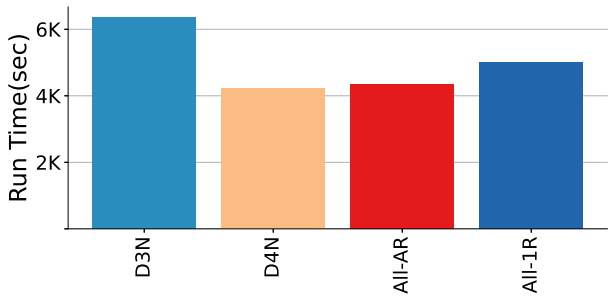


Figure 6·14: Run time of the trace with intermediate datasets. Total cache size is 640GB. D4N, All-1R and All-AR keeps the intermediate data in the cache. D3N and All-1R performance drops due to the extra traffic generated by the triple replication.

For Alluxio we use the full 640 GB for the combined read and write cache; D4N uses fixed partitioning, and was configured with 512GB and 128GB read and write cache respectively. Results are seen in Figure 6·14: D3N, with no write cache, has the slowest run-time, while D4N outperforms both Alluxio options. We attribute this to two factors: first, D4N uses erasure coding for its write-back tier, generating half as much rack-to-rack traffic as Alluxio, and second, D4N deletes data based on age, while Alluxio keeps data in cache until it is evicted via LRU. Results show that Alluxio-1R suffers from the internal traffic that is generated for accessing the single replica, and its run time is slower than Alluxio-AR. In both cases, data lake bandwidth was not a bottleneck for writing output data.

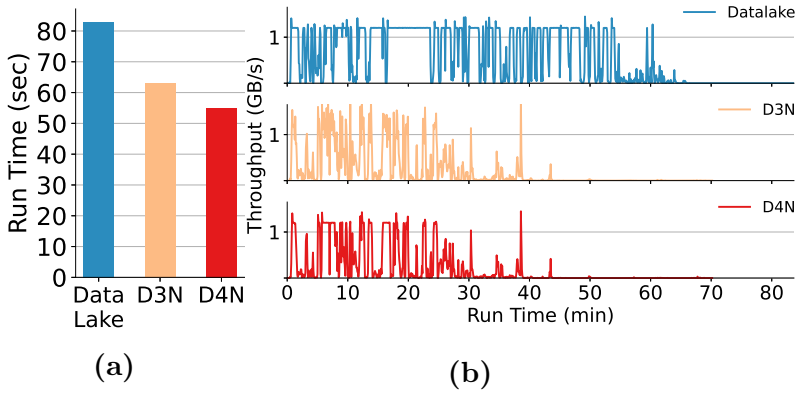


Figure 6·15: Run time and data lake accesses of production trace-based analytical workload for D4N, D3N and uncached datalake. D4N significantly reduces both run time data lake access.

6.6 Real Workload

The previous experiments either replay the real-world traces or used artificial workloads generated by what are essentially load testing tools; in real workloads performance is limited by computation as well I/O speed. Will these gains hold up under real, computation-based workloads, or have we merely accelerated I/O to the CPU-bound portions of a job, while leaving the I/O-bound portions unaffected? To investigate this we run a PIG/Hadoop-based TR-CP2020 workload, this time measuring run time and datalake traffic, for three configurations: D4N, D3N, and direct datalake access. Each cache node is configured with 512GB of cache space.

Results: We should note that the workload creates DAG-shaped query plan. Some jobs has multiple phases where they create intermediate results that becomes an input for the next job. In the experiment we observed that the workload creates burst of IO requests, and spend some time on computation and then issue IO requests again.

As seen in Figure 6·15a, D4N improves the performance by 31% and 12% against datalake and D3N respectively. D4N performans better because i) it provides a write-back cache which stores final output and also the intermediate data that has been

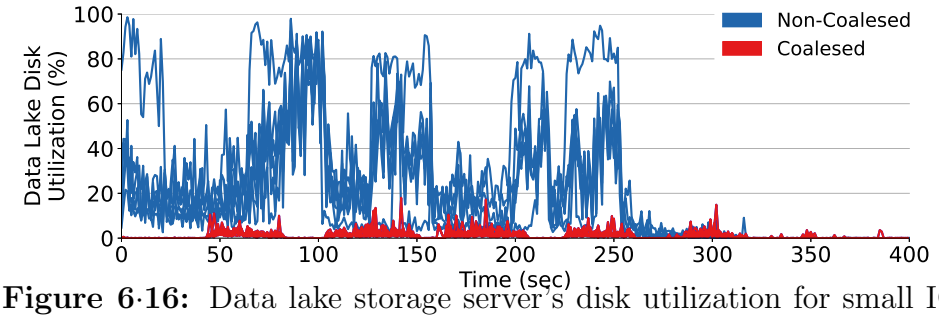


Figure 6·16: Data lake storage server's disk utilization for small IO requests. When small IO directly written to the data lake. The disk becomes a bottleneck. D4N' small object packing reduce the load on the disks significantly without impacting the performance.

generated between different jobs, and ii) D4N read data directly from datalake and unlike D3N it does not have an additional hop upon a cache miss.

Figure 6·15a shows one of the cache servers data lake network traffic which is captured by Linux-SAR monitoring tool. Results demonstrate that read and write-back cache reduce data lake accesses significantly.

6.7 D4N's Specialization

In this section, we evaluate the two specializations we have developed for D4N; small objects specialization via write coalescing and analytical frameworks specialization using Kariz [20], and we show how cache specialization can improve the performance of the system and workloads' runtime.

Small Objects Specialization We run an experiment to examine the effectiveness of small object packing with measuring disk utilization across the data lake storage servers. We use TR-IBM2019 that is dominated by small object sizes (4 MB and less), and replay this trace using `S3-benchmark`on a 48 node cluster where each node creates 32 concurrent requests.

Small writes are coalesced in 32 MB objects and are written to the data lake when they reach to the end of their lifetime in the cache, which is 1 minute in our

experiment. We throttle the transfer rate to the data lake to 1 GBps per cache node.

Figure 6·16 shows disk utilization of one of the data lake server. Load is depicted as disk utilization percentage, as reported by /proc/diskstats, across all disks in the data lake. When small requests are written directly to the data lake storage servers, we observe average disk utilization is between 40%-60%, and for some of the disks for a certain period we observe 100% disk utilization. Coalescing small objects into large objects reduces the load on the data lake storage disks. With coalesced writes, the data lake disk utilization is below 15% for most of the time and never exceed 20%. Coalescing writes doesn't impact the write throughput since the data is written into the write-tier. Read requests to small objects are issued as a "range-request", and we do not observe any performance degradation on the read performance.

We also observe that the aging process can create burst of I/O to the data lake. Increasing the aging window and/or transfer rate increases the load on the disks. Our results suggest that for the best performance the transfer rate and the window for aging operation should be dynamically adjusted based on the observed traffic to the data lake.

Analytical Frameworks Specialization In this experiment we evaluate the benefit of specializations to support wide range of access patterns of workloads. We deploy Kariz [20] to give hints to D4N based on the extracted DAG information of submitted analytical workloads.

We mimic a shared multi-cluster data center where one cluster is aggressively using the entire cache space and other clusters may starve. Caching specializations like Kariz can provide more efficient caching for these clusters that are not aggressively accessing the storage. To show this; we set up two clusters each with 12 compute nodes, both accessing the same cache server, which has a 512GB capacity. Cluster 1 runs a workload, which creates 20TB read requests with 1TB unique dataset and

uniform access distribution. Cluster 2 runs Hadoop/PIG framework along with Kariz which runs TR-CP2021 workload.

When Kariz gives hints(prefetch, pin) to the D4N, the run time of DAG based workload is improved by 20%. When the Kariz is disabled, Cluster2 has to read most of the data from the data lake since Cluster1's read demand is much larger than the available cache capacity which results in eviction of the Cluster 2's data sets from cache.

Chapter 7

Conclusions and Future Work

7.1 Conclusion

This dissertation focuses on extending immutable object-based data lake across a data center using *cooperative caching* techniques. We developed two data center scale cooperative cache architectures, D3N and D4N, designed to be part of the data lake itself rather than part of the computer clusters use it. We demonstrate that exploiting the immutability of object stores significantly reduces the complexity of cache design, allowing us to explore caching strategies that were not feasible for previous cooperative cache systems for a file or block-based storage. Both architectures explore cooperative caching techniques to extend the data lake by distributing the caches all around the data center and support data sharing across all workloads, frameworks, and clusters in a data center which access the same shared data lake. We show that extending the data lake to cache data near the point of access improves the locality and delivers high bandwidth to applications.

D3N shows that it is possible to implement a real working cooperative cache for immutable data lakes without requiring any modification to applications. It enables the integration of caches into an existing data lake, and D3N has been upstream into an existing Ceph as an experimental feature, allowing many data lakes around the world to deploy it. We show it is possible to implement a cache management algorithm to adjust local and global demand based on local information (measure latency and historical access patterns). Our experiments demonstrate that D3N's implementation

is highly efficient to saturate the dual NVMe SSDs and the 40 Gbit NIC, where a single SSD throughput is 5× faster than a 90 spindle Ceph cluster accessed without network bottleneck. D3N achieves significant performance improvements —up to 3× reduction in runtime and reduces the bisectional bandwidth demands by a factor of 4 for realistic data analytical workloads.

D4N architecture tradeoff the simplicity with sophisticated caching strategies. D4N architecture shows the importance of maintaining a global state, which is required to have a write tier and cache specialization per application. We demonstrate that it is possible to maintain a global state for object stores without introducing performance overheads due to the large block access of object stores. This allows us to explore sophisticated caching strategies that were not feasible for previous cooperative cache systems for a file or block-based storage. D4N exploits the composability of the S3-interface and re-uses the existing Ceph software to layer write tier on top of the data lake. Finally and most importantly, D4N demonstrates the possibility of designing a customizable storage system by enabling customization. It supports application-specific specializations on top of a general storage service, eliminating the need to fragment storage into separate pools in order to provide performance improvements for specific applications. We implemented a new intelligent caching algorithm, GWF, which automatically adjusts the global demand on the data center by comparing the global values of each cached block, and performs well in both high local demand and high global demand, and achieves substantial performance gains over D3N and a state of art cluster cache Alluxio. D4N's small object packing specialization reduces the server load on the data lake by 200% over the case of no write tier, and with analytic cluster-specific specialization, it improves the run time by 20%.

7.2 Future Work

One of the least explored areas of the D3N and D4N research agenda has been the fair sharing of cache resources. In both cache architectures, the cache resource allocation policies are first-come-first-serve. For instance, users who read data at long intervals may gain little or no benefit from the cache simply because their data is likely to be evicted. Another important future research direction is how to provide a partitioning mechanism for a read and a write cache, dynamically adjusting the size of each cache by sensing the characteristics of workloads. It would be crucial to investigate resource allocation strategies for both architectures' read/write cache and multiple users.

An obvious direction for future work is to develop new specializations to customize caching policies for more workloads. With knowledge of the access pattern, one can customize caching policies (e.g., replication, write-back, admission, prefetch, redundancy), enabling valuable cache resources to be efficiently used. For example, the replication policy of write tier (either replication count or erasure code length) can be tuned per application to match the application's availability requirements. In addition, future work should explore how to combine cache specializations with the composability of the S3 interface to implement new functionalities on top of D4N caches or the data lake and understand the potential and challenges of customizability and composability. Furthermore, it would be interesting to investigate how to base everything on immutable object stores and find mechanisms for efficiently implementing mutability, resiliency, and consistency at the most appropriate layers to support a wide range of applications.

Both D3N and D4N are designed for a single data center. Future research can extend both systems for hybrid clouds or geo-distributed data centers where data is replicated to caches across multiple data centers and stored in different data lakes. Can caches still exploit the immutability to efficiently move data between different

data centers or cloud regions in such use cases? Investigating caching policies and exploring how to address cache coherence and scalability issues in a geo-distributed data centers would be an important research direction.

Finally, "upstreaming" the code is critical for system research because otherwise, the promising research results go unnoticed and unused. Once the upstreaming is completed, the code will be supported by an external engineer team and can be deployed and used in real production systems. We are still in the process of "upstreaming" the D4N prototype which is a lengthy process of convincing the Ceph community, followed by code review, testing, and further development. We are still unclear when the upstream takes place; therefore, future work should develop strategies for making the "upstreaming" process more efficient and easier for researchers, such as developing unit testing strategies, automating the deployment and configurations, and providing documentation.

Bibliography

[1] Alluxio Amazon AWS S3. https://docs.alluxio.io/os/user/stable/en/ufs/S3.html#advanced-credentials-setup.

[2] Alluxio Security. https://docs.alluxio.io/os/user/stable/en/operation/Security.html.

[3] AWS Lambda. https://aws.amazon.com/lambda.

[4] CEPH Object gateway SWIFT API. http://docs.ceph.com/docs/master/radosgw/sswift/.

[5] cpp-redis: C++11 Lightweight Redis client. https://github.com/cpp-redis/cpp_redis.

[6] HDFS Storage Efficiency Using Tiered Storage. http://www.ebaytechblog.com/2015/01/12/hdfs-storage-efficiency-using-tiered-storage/.

[7] Improving Data Locality for Analytics Jobs on Kubernetes Using Alluxio. ImprovingDataLocalityforAnalyticsJobsonKubernetesUsingAlluxio.

[8] Inntroduction to Librados. http://docs.ceph.com/docs/giant/rados/api/librados-intro/.

[9] MinIO Optimizes Small Object Storage and Adds .tar Auto-Extraction. https://blog.min.io/minio-optimizes-small-objects.

[10] redis-pys: The Python interface to the Redis key-value store. https://github.com/redis/redis-py.

[11] store small object's data part into xattr. https://github.com/ceph/ceph/pull/29863.

[12] TensorFlow: Large-scale machine learning on heterogeneous systems.

[13] The Massachusetts Green High Performance Computeing Center (MGHPCC). https://www.mghpcc.org.

[14] Varnish HTTP Cache. https://varnish-cache.org/.

[15] Web widget nudges scientists to share their data. http://dataverse.org/.

[16] Databricks IO Cache. `https://docs.databricks.com/delta/optimizations/delta-cache.html`, 2021.

[17] Operation of Anycast Services. `https://tools.ietf.org/html/rfc4786`, 2021.

[18] Pig TPC-H Queries. `http://github.com/ssavvides/tpch-pig`, 2021.

[19] rgw: D3N Cache changes for Upstream 36266. `https://github.com/ceph/ceph/pull/36266`, 2021.

[20] Mania Abdi, Amin Mosayyebzadeh, Mohammad Hossein Hajkazemi, Emine Ugur Kaynar, Ata Turk, Larry Rudolph, Orran Krieger, and Peter Desnoyers. A Community Cache with Complete Information. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 323–340. USENIX Association, February 2021.

[21] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Apostol (Paul) Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. In *arXiv:1609.08675*, 2016.

[22] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*, Boston, MA, December 2002. USENIX Association.

[23] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: Lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 353–369, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, Francois Labelle, Nathan Coehlo, Xudong Shi, and Eric Schrock. Janus: Optimal Flash Provisioning for Cloud Storage Workloads. In *Proceedings of the USENIX Annual Technical Conference*, pages 91–102, 2560 Ninth Street, Suite 215, Berkeley, CA 94710, USA, 2013.

[25] Alluxio - Open Source Memory Speed Virtual Distributed Storage. `https://www.alluxio.org`, 2021.

[26] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, page 562–570, Washington, DC, USA, 1976. IEEE Computer Society Press.

[27] Inc. Amazon Web Services. Amazon Simple Storage Service (S3) — Cloud Storage — AWS. available at `aws.amazon.com/s3/`.

[28] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 267–280, San Jose, CA, 2012. USENIX.

[29] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Trans. Comput. Syst.*, 14(1):41–79, feb 1996.

[30] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: Scaling File Servers via Cooperative Caching. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 129–142, Berkeley, CA, USA, 2005. USENIX Association.

[31] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R. Butt. Taming the cloud object storage with mos. In *Proceedings of the 10th Parallel Data Storage Workshop*, PDSW '15, page 7–12, New York, NY, USA, 2015. Association for Computing Machinery.

[32] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating Content Management Techniques for Web Proxy Caches. *SIGMETRICS Perform. Eval. Rev.*, 27(4):3–11, March 2000.

[33] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał undefinedwitakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. Delta lake: High-performance acid table storage over cloud object stores. *Proc. VLDB Endow.*, 13(12):3411–3424, aug 2020.

[34] AWS Auto Scaling. `https://aws.amazon.com/autoscaling/`, 2021.

[35] Azure Auto Scaling. `https://azure.microsoft.com/en-us/features/autoscale/`, 2021.

[36] Azure Premium Storage, now generally available. https://azure.microsoft.com/en-us/blog/azure-premium-storage-now-generally-available-2/.

[37] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[38] Alexandros Batsakis and Randal Burns. Nfs-cd: write-enabled cooperative caching in nfs. *IEEE Transactions on Parallel and Distributed Systems*, 19(3):323–333, 2008.

[39] Doug Beaver, S Kumar, H C Li, J Sobel, and P Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. *OSDI*, 2010.

[40] S. Byan, J. Lentini, A. Madan, and L. Pabón. Mercury: Host-side flash caching for the data center. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, April 2012.

[41] CACHE TIERING. http://docs.ceph.com/docs/master/rados/operations/cache-tiering/.

[42] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), June 2008.

[43] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, 2016. USENIX Association.

[44] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache.

[45] CloudBD. https://www.cloudbd.io/, 2021.

[46] Frank Dabek, M. Frans Kaashoek, David R. Karger, Robert Tappan Morris, and Ion Stoica. Wide-area cooperative storage with CFS. *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

[47] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[48] EMC Corporation. EMC FAST Cache. http://www.emc.com/collateral/software/white-papers/h8046-clariion-celerra-unified-fast-cache-wp.pdf.

[49] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. It's Time to Revisit LRU vs. FIFO. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, July 2020.

[50] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 201–212, New York, NY, USA, 1995. Association for Computing Machinery.

[51] Google Auto Scaling. https://cloud.google.com/compute/docs/autoscaler, 2021.

[52] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. *SIGOPS Oper. Syst. Rev.*, 23(5):202–210, November 1989.

[53] Gabriel Haas, Michael Haubenschild, and Viktor Leis. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

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

[55] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: highly durable, decentralized storage despite massive correlated failures. In *NSDI*, 2005.

[56] Mohammad Hossein Hajkazemi, Vojtech Aschenbrenner, Mania Abdi, Emine Ugur Kaynar, Amin Mossayebzadeh, Orran Krieger, and Peter Desnoyers. Beating the i/o bottleneck: A case for log-structured virtual disks. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 628–643, New York, NY, USA, 2022. Association for Computing Machinery.

[57] Amazon S3 Storage Classes. https://aws.amazon.com/s3/storage-classes/.

[58] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Trans. Comput. Syst.*, 6(1):51–81, feb 1988.

[59] Amazon EMR. https://aws.amazon.com/emr/.

[60] IBM Easy Tier Function. https://www.ibm.com/support/knowledgecenter/STVLF4_7.8.0/spectrum.virtualize.780.doc/svc_easy_tier.html.

[61] IBM Cloud Object Storage File Access (FA). https://www.ibm.com/cloud/blog/object-storage-traces.

[62] Solid-State Drive Caching in the IBM XIV Storage System. http://www.redbooks.ibm.com/redpapers/pdfs/redp4842.pdf.

[63] ImageNet. https://www.image-net.org/, 2021.

[64] Virajith Jalaparti, Chris Douglas, Mainak Ghosh, Ashvin Agrawal, Avrilia Floratou, Srikanth Kandula, Ishai Menache, Joseph (Seffi) Naor, and Sriram Rao. Netco: Cache and I/O Management for Analytics over Disaggregated Stores. In *ACM Symposium on Cloud Computing (SOCC)*, October 2018. Best Paper Award.

[65] Song Jiang, F. Petrini, Xiaoning Ding, and Xiaodong Zhang. A locality-aware cooperative cache management protocol to improve network file system performance. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*, pages 42–42, 2006.

[66] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB*, 1994.

[67] Saurabh Kadekodi, Bin Fan, Adit Madan, Garth A. Gibson, and Gregory R. Ganger. A Case for Packing and Indexing in Cloud File Systems. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, Boston, MA, July 2018. USENIX Association.

[68] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[69] Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 33–45, Santa Clara, CA, February 2014. USENIX Association.

[70] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[71] M. R. Korupolu and M. Dahlin. Coordinated placement and replacement for large-scale distributed caches. *IEEE Transactions on Knowledge and Data Engineering*, 14(6):1317–1329, Nov 2002.

[72] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An Informed Storage Cache for Deep Learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 283–296, Santa Clara, CA, February 2020. USENIX Association.

[73] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, page 148–159, New York, NY, USA, 1990. Association for Computing Machinery.

[74] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. *Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks*. ACM SoCC'14, November 2014.

[75] Kunal Lillaney, Vasily Tarasov, David Pease, and Randal Burns. Agni: An Efficient Dual-access File System over Object Storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 390–402, Santa Cruz CA USA, November 2019. ACM.

[76] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[77] Data Lifecycle Management and Tiering. `https://min.io/product/automated-data-tiering-lifecycle-management`, 2021.

[78] MinIO Web Site. `https://docs.min.io/docs/minio-quickstart-guide.html`, 2021.

[79] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Imple-*

mentation (OSDI 18), pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[80] Amin Mosayyebzadeh, Apoorve Mohan, Sahil Tikale, Mania Abdi, Nabil Schear, Trammell Hudson, Charles Munson, Larry Rudolph, Gene Cooperman, Peter Desnoyers, and Orran Krieger. Supporting Security Sensitive Tenants in a Bare-Metal Cloud. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 587–602, Renton, WA, July 2019. USENIX Association.

[81] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic Skip Lists. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, pages 367–375, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[82] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's Warm BLOB Storage System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 383–398, Broomfield, CO, 2014. USENIX Association.

[83] MuthitacharoenAthicha, ChenBenjie, and MaziéresDavid. A low-bandwidth network file system. *Operating Systems Review*, 2001.

[84] Srivatsan Narasimhan, Sohum Sohoni, and Yiming Hu. A log-based write-back mechanism for cooperative caching. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 10–pp. IEEE, 2003.

[85] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat Datacenter Storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 1–15, Berkeley, CA, USA, 2012. USENIX Association.

[86] ObjectiveFS: Shared S3 File System for EC2 instances. `https://objectivefs.com/`, 2021.

[87] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1099–1110. ACM, 2008.

[88] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.*, 33(4):351–385, jun 1996.

[89] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231. USENIX Association, February 2021.

[90] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231. USENIX Association, February 2021.

[91] Tiago B. G. Perez et al. Reference-Distance Eviction and Prefetching for Cache Management in Spark. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, 2018.

[92] Christian Pinto, Yiannis Gkoufas, Andrea Reale, Seetharami Seelam, and Steven Eliuk. Hoard: A Distributed Data Caching System to Accelerate Deep Learning Training on the Cloud. *CoRR*, abs/1812.00669, 2018.

[93] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, February 2019. USENIX Association.

[94] Ceph Object Gateway. `http://docs.ceph.com/docs/master/radosgw/`, 2021.

[95] Raghu Ramakrishnan, Baskar Sridharan, John R. Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, Neil Sharman, Zee Xu, Youssef Barakat, Chris Douglas, Richard Draves, Shrikant S. Naidu, Shankar Shastry, Atul Sikaria, Simon Sun, and Ramarathnam Venkatesan. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 51–63, New York, NY, USA, 2017. ACM.

[96] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-cache: Load-balanced, Low-latency Cluster Caching with Online Erasure Coding. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 401–417, Berkeley, CA, USA, 2016. USENIX Association.

[97] Redis. `https://redis.io/`, 2021.

[98] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[99] Francisco Romero, Gohar Irfan Chaudhry, Iñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. SoCC'21, Seattle, Washington, USA, 2021.

[100] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *13th ACM symposium on Operating systems principles*, pages 1–15, Pacific Grove, California, United States, 1991. ACM.

[101] s3-benchmark: Performance test for comparison of AWS vs Wasabi S3 systems, 2021.

[102] S3A FileSystem Client.

[103] Prasenjit Sarkar and John H Hartman. Hint-based cooperative caching. *ACM Transactions on Computer Systems (TOCS)*, 18(4):387–419, 2000.

[104] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[105] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020.

[106] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[107] Two Sigma. `https://www.twosigma.com`.

[108] Simpy: Discrete event simulation for Python, 2021.

[109] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the 2015 ACM SIGCOMM Conference*, SIGCOMM '15, pages 183–197. ACM, 2015.

[110] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007.

[111] Storage Performance Development Kit | 01.org. `http://www.spdk.io`.

[112] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A Warehousing Solution over a Map-reduce Framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[113] Niraj H. Tolia, Michael A. Kozuch, Mahadev Satyanarayanan, Brad Karp, Thomas C. Bressoud, and Adrian Perrig. Opportunistic Use of Content Addressable Storage for Distributed File Systems. In *USENIX Annual Technical Conference, General Track*, 2003.

[114] Geoffrey M. Voelker, Eric J. Anderson, Tracy Kimbrel, Michael J. Feeley, Jeffrey S. Chase, Anna R. Karlin, and Henry M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '98/PERFORMANCE '98, page 33–43, New York, NY, USA, 1998. Association for Computing Machinery.

[115] Werner Vogels. File system usage in windows nt 4.0. *SIGOPS Oper. Syst. Rev.*, 33(5):93–109, dec 1999.

[116] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building An Elastic Query Engine on Disaggregated Storage . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 449–462, Santa Clara, CA, February 2020. USENIX Association.

[117] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 267–281, Santa Clara, CA, February 2020. USENIX Association.

[118] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[119] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[120] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. Rados: A scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, page 35–44, New York, NY, USA, 2007. Association for Computing Machinery.

[121] Mark Woods, Amit Shah, and Paul Updike. Intelligent Caching and NetApp Flash Cache. `http://community.netapp.com/t5/Tech-OnTap-Articles/Intelligent-Caching-and-NetApp-Flash-Cache/ta-p/85956`.

[122] G. Yadgar, M. Factor, and A. Schuster. Cooperative caching with return on investment. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–13, May 2013.

[123] Yizhe Yang, Qingni Shen, Wu Xin, Wenjun Qian, Yahui Yang, and Zhonghai Wu. Memory cache attacks on alluxio impede high performance computing. In *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pages 407–414, 2018.

[124] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled Ben Letaief. Lrc: Dependency-aware cache management for data analytics clusters. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017.

[125] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

CURRICULUM VITAE