University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Andrzej Jackowski

Adapting Distributed Storage with Deduplication to Cloud Use Cases

PhD dissertation
in COMPUTER SCIENCE

Supervisor:
dr hab. Konrad Iwanicki
Institute of Informatics
University of Warsaw

Auxiliary supervisor:
dr Cezary Dubnicki
9LivesData, LLC

Warsaw, September 2023

Author's declaration:
I hereby declare that this dissertation is my own work.

....
date signature

Supervisors' declaration:
The dissertation is ready to be reviewed

....
date signature

The dissertation is ready to be reviewed

....
date signature

Abstract

Deduplication is a widely adopted data reduction technique. It is commonly utilized in backup and archival storage systems to decrease space consumption and increase write throughput. Today, such systems have to confront the fact that the popularity of cloud computing has grown rapidly in recent years, and hence, to keep up with these technological trends, they have to be adapted to the cloud environment. However, employing deduplication in cloud computing poses multiple challenges. In this dissertation, we introduce novel solutions to three such significant problems emerging in distributed storage with deduplication for cloud-oriented backup and archival applications.

First, given that object storage is a leading interface for accessing cloud data, we propose ObjDedup, a layer that implements such an interface for backup systems with block-level deduplication, thereby enabling their interoperability with clouds. ObjDedup introduces global data structures and algorithms designed to handle considerable object metadata, which are produced in such solutions. In effect, it achieves 1.8–3.93x higher write throughput than object storage without in-line deduplication. Moreover, compared to object storage on top of file-based backup systems, it processes 5.26–11.34x more object put operations per time unit.

Our second solution is InftyDedup, a novel system that implements tiering to cloud with deduplication, which is a technique that enables moving deduplicated data to a cloud store lacking deduplication. Unlike existing solutions, InftyDedup maximizes scalability by utilizing cloud services not only for storage but also for computation. Following a distributed batch processing approach with dynamically assigned cloud computing resources, InftyDedup can deduplicate multi-petabyte backups from multiple sources at costs on the order of a few dollars. Moreover, by selecting between "hot" and "cold" cloud storage based on the characteristics of each data chunk, it further reduces the overall costs by up to 26%–44%.

Since ObjDedup and InftyDedup require high resilience and efficient resource utilization from the deduplication system, our third solution, Derrick, aims to maximize both. It is a data balancer designed to make its decisions quickly in case of failures, yet to be allowed to take extra time to find a nearly optimal data arrangement and a plan for reaching it when the device population changes in a planned fashion. Derrick provides better capacity utilization, reduced data movement, and improved performance compared to balancing algorithms in two other leading systems. Moreover, it can be easily adapted to meet custom placement requirements.

Apart from advancing the state of the art, our solutions have been or will be deployed in HYDRAstor, a commercial system utilized by many organizations. Therefore, our research actually improves real-world backup and archival storage products.

Contents

Chapter 1

Introduction

Deduplication is a data reduction technique for storage systems that prevents keeping the same data units more than once. To date, it has been widely adopted, especially in systems for backup and archival applications, which have to reliably and efficiently maintain large data volumes [94, 95, 234]. In particular, because consecutive backups typically share most of their data, deduplication can reduce the storage space they consume by over 90% [95, 285]. Considering that in enterprise environments, the copies of data needing protection that accumulates over the years often require much more space than the original data [85], effective deduplication is a highly desirable feature. It is thus often one of the core functionalities of so-called *purpose-built backup appliances* (PBBAs) (or *backup appliances* for short) [89, 137], that is, (distributed) storage systems dedicated primarily to the backup and archival market.

Since the worldwide data volume is expected to continue growing exponentially in the coming years [99], data storage techniques — including those related to deduplication — constantly evolve to meet the evolving market demands and harness the potential offered by new technologies. On the one hand, physical storage carriers, notably hard disk drives (HDDs) and solid state drives (SSDs), are improved to offer more capacity [133]. On the other hand, various paradigms and abstractions over such physical resources, which facilitate their effective utilization, gained popularity. In particular, given that in recent years, the adoption of cloud services has grown rapidly [208], a notable trend in the backup and archival market is the increasing use of solutions employing object storage services offered by public clouds [45].

However, apart from presenting new opportunities for distributed storage with deduplication, these novel technologies and paradigms also pose multiple challenges. For instance, according to a survey published in 2018 by Storage Networking Industry Association [265], adopting the cloud, lowering costs, and facilitating data migration are critical concerns regarding long-term data retention in large-scale storage systems. Similarly, high-impact open problems pertaining specifically to deduplication [273] include facilitating data migration, increasing reliability, and optimizing resource consumption. Gartner emphasizes the fact that many leading enterprise backup and recovery solutions lack sufficient cloud support [127]. The trends report published in 2023 by Veeam confirms that companies typically store data in both on-premise and cloud systems, and that the majority of companies have unmet data availability and protection needs in the era of such hybrid solutions [306]. Finally, the significance of improving PBBAs is reinforced by the expected growth of their market, which is projected at 68% in the next five years [257].

Considering the aforementioned and similar analyses, this dissertation investigates novel solutions for distributed storage systems with deduplication, especially ones aimed at the

backup and archival market, that would make such systems better suited for the cloud computing environment. More specifically, the dissertation tackles the following research problems motivated below:

1. How to provide an object storage interface in a backup appliance with deduplication, which may be internally implemented as a distributed system, so that the throughput achievable over the interface would be comparable to those over the classic interfaces supported by the appliance?

2. How to integrate a backup appliance with cloud computing services into a two-tier storage system that would allow for moving deduplicated data between the tiers without capacity constraints due to the appliance tier and with low maintenance costs due to the cloud tier?

3. How to manage the placement of data onto physical storage devices in a backup appliance so as to ensure high resource utilization and reliability irrespective of the scale that a distributed system constituting the appliance exhibits?

To start with, as object storage is arguably the most common storage abstraction in the cloud, finding a solution to Problem 1 would allow services and apps written to work with the cloud interface to integrate seamlessly with PBBA, thereby utilizing in cloud services the immense body of knowledge on deduplication. To address Problem 1, we present ObjDedup, a high-throughput object storage layer for a state-of-the-art distributed store with global block-level in-line deduplication. As backup is one of the primary use cases for such systems [329], ObjDedup aims to provide a high throughput for workloads generated by modern backup applications writing data to object stores. They tend to use relatively small objects (e.g., 1 MB [301]), thereby incurring a large overhead on metadata, which is difficult to handle even in storage systems without deduplication [67]. Consequently, for efficiently managing such large metadata volumes, ObjDedup introduces novel data structures and algorithms dedicated for immutable-block stores under the considered workloads.

Second, given the popularity of keeping data in the cloud and the fact that, unlike backup appliances, most cloud storage services do not implement deduplication, Problem 2 concerns an emerging scenario in which on-premise backup appliances with deduplication are integrated with cloud services to implement two-tier storage systems that combine the advantages of both tier types. Such a technique of moving data from on-premise systems to the cloud is often referred to as *tiering to cloud* or *cloud tiering*. For the considered applications of such a system, a key challenge is efficiently moving data from the on-premise tier to the cloud tier such that the data end up deduplicated in the cloud. Although the first solutions implementing tiering to cloud have been proposed recently, they are fundamentally limited by the resources of the on-premise tier and lack in cost optimization. We thus present a novel solution to Problem 2, InftyDedup, that addresses these shortcomings by leveraging not only object storage but also a few other cloud features, such as dynamic resource allocation. This approach allows InftyDedup to perform duplicate elimination entirely in the cloud and facilitates employing various special-purpose cloud storage classes for data maintenance cost reduction.

Finally, both ObjDedup and InftyDedup require high resilience and efficient resource utilization from backup appliances; otherwise, the performance of systems the appliances are part of, such as the two aforementioned ones, would be compromised in those respects. On the one hand, providing these features in a constantly evolving system is difficult. In particular, the performance of storage devices does not increase as fast as their capacity [133]. On the other hand, changing the size of the system constituting a backup appliance in most

cases is inevitable for two reasons. First, due to the worldwide data growth, the storage requirements of the system are growing each year. Second, when the system capabilities are extended, notably by adopting ObjDedup or InftyDedup, the system should be prepared to handle the additional load resulting from the new functionality. Therefore, to address Problem 3, we propose Derrick, a three-layer balancer for what we called self-managed continuous scalability. It is a novel method of data distribution, which ensures that various constraints, including those specific for deduplication [283], are met during system configuration changes.

The rest of the dissertation is organized as follows. Chapter 2 provides the necessary background on deduplication. Chapter 3 gives a survey of relevant existing and emerging technologies in distributed storage. Chapters 4, 5, and 6 present the aforementioned core contributions of the dissertation: respectively, ObjDedup, InftyDedup, and Derrick. Finally, Chapter 7 concludes and outlines possible future work.

The research on ObjDedup, described in Chapter 4, was previously published in *IEEE Transactions on Parallel and Distributed Systems* [146]. Likewise, InftyDedup, covered in Chapter 5, was originally presented at the *21st USENIX Conference on File and Storage Technologies (FAST '23)* [169]. The work on Derrick, described in Chapter 6, appeared in turn in *ACM Transactions on Storage* [145]. Also, some parts of Chapters 2 and 3 have been compiled from those three publications.

All three solutions have been implemented for the HYDRAstor distributed storage system [94, 225]. Some of them are already integrated into the product and have proved to be effective in real-world deployments, while others still wait to be released in the future.

Chapter 2

Deduplication in Data Storage

Before we explain the details of our techniques for adapting distributed storage with deduplication to cloud use cases, let us introduce essential details of deduplication. From the perspective of the *"Mathematical Theory of Communication"* [271], discrete information can be represented as a string of bits, and the occurrence probability of each group of bits might be different. *Entropy* is a measure of uncertainty in data (i.e., entropy is maximized if each symbol or symbol sequence is equally possible), whereas *redundancy* measures a relative difference between the actual entropy and its maximal possible value. Techniques to decrease redundancy (e.g., data compression to reduce data size) and to increase redundancy (e.g., error correction codes to improve system reliability) are commonly used.

Deduplication is a method of decreasing redundancy. It originates from systems that avoid writing files [54, 282] or blocks of data [220, 255] when a file or block with exactly the same fingerprint (i.e., a cryptographic hash of data, like SHA-1) is already stored in the system. As deduplication is often utilized with other algorithms for decreasing redundancy, such as *Lempel-Ziv '77* [255] or gzip [220], the term *deduplication* is typically used to distinguish techniques that avoid writing data based on fingerprints [202, 348] from other algorithms, such as the previously mentioned *Lempel-Ziv '77*, which are often denoted as *compression* [79]. There are many approaches to implementing deduplication, for instance, similarity signature can be computed [33] rather than a cryptographic hash of data, however, in our study, we focus on the methods that are dominating the commercial systems.

Empirical studies show that performing deduplication on blocks of data with limited sizes (e.g., 2–128 KB per block) yields a higher data reduction than deduplication of entire files [211, 285]. Therefore, in our work, we focus on such block-level deduplication. The fact that block-level deduplication achieves superior reductions matches the intuition that finding two identical sequences of bytes is easier if the sequences are shorter. However, blocks cannot be too small as various overheads increase when the block size decreases [202].

Implementation of block-level deduplication requires chunking data into blocks. The most straightforward approach is to simply chunk data into fixed-size blocks (e.g., 8 KB each) [255]. Such a method fails to detect many real-world cases of duplicates, for instance, between two almost identical data streams if one of the streams is shifted relative to the boundaries of the blocks. Therefore, content-based variable-length chunking methods [172, 220] are often employed instead to improve the achievable deduplication ratios.

To sum up, a typical system with block-level deduplication chunks data streams into blocks, computes the fingerprint for each block, and eventually makes a decision on whether the block should be kept. However, this is just the beginning of the deduplication process because the effective implementation of deduplication requires solving numerous non-trivial

problems. Apart from selecting methods for data chunking, calculation of fingerprints, and fingerprint indexing, deduplication introduces challenges regarding data organization and locality, resource utilization, security, reliability, and many more. Therefore, the rest of this chapter is organized in the following way. Section 2.1 surveys challenges posed by systems with deduplication and related solutions proposed in the literature. Section 2.2 reviews applications of deduplication. Section 2.3 describes HYDRAstor, a state-of-the-art commercial distributed storage system with deduplication that was a starting point for our research. Finally, Section 2.4 presents the model of a deduplication system that we assume in the remaining chapters.

2.1. Challenges in Systems with Data Deduplication

2.1.1. Data Chunking

The first step of deduplication is chunking, which divides a data stream (e.g., a file) into fine-grained blocks. Fixed-size chunking generates blocks of equal lengths, which, as mentioned earlier, fails to detect duplicates in shifted data. In some applications (e.g., deduplication of blocks comprising a disk image of a virtual machine) shifts do not occur, and hence, fixed-size chunking is viable [6]. Moreover, the simple idea of fixed-size chunking can be further extended. For instance, DSFSC [171] limits occurrences of undetected duplicates for shifted data by performing fixed-size chunking twice: once from the start of a stream and once from the end.

Nevertheless, substantial research attention has concentrated on variable-length chunking which improves deduplication for shifted data. LBFS [220] implements variable-length chunking by computing a Rabin fingerprint [256] for each 48-byte region of a stream and setting the block boundaries based on the values of the fingerprints. As calculating and comparing fingerprints introduces significant CPU overheads, there are several algorithms, such as Gear [330], AE [342], and FastCDC [331], that focus on improving chunking performance without a significant deterioration of duplicate elimination. Since a reduction of block size typically results in a better deduplication ratio but also increases the overhead on storing and managing block metadata, algorithms such as Bimodal CDC [172] and Anchor-Driven Subchunk [259] introduce techniques that increase the average size of a block without decreasing the deduplication ratio. Finally, data characteristics depend on a data format and a source application. For instance, *tar* archives mix their metadata with the actual data, which spoils deduplication [193]. Consequently, approaches like Application Aware Deduplication [102] and Format Aware Deduplication [193] improve chunking with methods specific to particular applications and formats.

2.1.2. Fingerprint Generation

After a data stream is chunked into blocks, each of the blocks is assigned its fingerprint. A typical *compare-by-hash* approach is employed: computing a cryptographic hash of a block's data, such as SHA-1 or SHA-256, and comparing the blocks solely based on their hashes [220, 255]. Such an approach can be seen as counterintuitive because of the theoretical possibility of a hash collision [125]. However, the probability of a collision for 20-byte and longer hashes is marginal, and therefore, compare-by-hash is considered safe [52].

Similarly to chunking, fingerprinting introduces a significant computational overhead that researchers have tried to minimize. A possible solution is to employ GPUs [119, 288] or FPGAs [4] to speed up the hash computations. There are also algorithms that do not require

specific hardware, for instance, DeWrite [351] computes only a lightweight hash of data instead of a cryptographic one and, in case of collisions, resorts to comparing the data.

2.1.3. Fingerprint Indexing

Assuming a 20-byte fingerprint per 8 KB block, each petabyte of data requires 2.5 TB of fingerprints. Therefore, in multi-petabyte systems, storing all fingerprints entirely in RAM is typically not possible, and hence search and comparison of fingerprints is non trivial. Fingerprints can be stored on HDDs and relying on data locality [190, 348] and Bloom filters [318, 295], one can try to prevent expensive random I/Os. Alternatively, fingerprints can be stored on SSDs [6, 86], which provide orders of magnitude more random I/Os per second.

2.1.4. Data Fragmentation

Information locality is important not only for an efficient organization of fingerprints but also for the organization of the actual data blocks on disks. In systems without deduplication, a file is typically stored with a high locality (i.e., sequential bytes of the file are located close to each other on disk or disks). In contrast, in a system with deduplication, two consecutive blocks may have a completely different location if one of them is a duplicate (so it is already stored somewhere) and the other block is non-duplicate (so it is written in the currently available space). Such a phenomenon is often referred to as *fragmentation*. It is a serious issue when reading deduplicated data from HDD-based systems, as such drives can provide a relatively small number of non-sequential read I/Os per second.

The fragmentation issue can be mitigated if selected blocks are kept in an in-memory cache [152, 244] or on SSDs [83, 192]. Alternatively, blocks can be kept on a disk in more than one order to trade the data reduction for locality [61, 189]. Finally, blocks can be rewritten to newer locations [151] and removed from the old locations to prevent additional storage consumption.

2.1.5. Data Removal

Removal of old data blocks from a system with deduplication is required not only to implement defragmentation but also to allow efficient resource utilization if the system user decides to delete some of the stored data. The problem with the implementation of data removal is that deletion of a file in a system with deduplication does not exactly mean that any of its blocks are removed, as each block can belong to more than one file. Even if the system confirms that a block is no longer referenced by any of the existing files, a dangerous race condition arises in which the block can be removed and written again simultaneously. Therefore, some systems with deduplication explicitly disallow removing data [189, 255], whereas others implement complex garbage-collection solutions based on mark-and-sweep [92] or reference counting [283].

2.1.6. Inline and Offline Deduplication

An important decision in systems with deduplication is whether duplicate elimination should be done before storing the data (inline deduplication) or after the data are stored in a non-deduplicated format (offline deduplication). On the one hand, inline deduplication decreases storage capacity usage the most and, for data with a high number of duplicates, achieves considerably higher throughput as the volume of disk and network operations is decreased.

On the other hand, offline deduplication is often applied in systems with low-latency expectations [174, 234] to provide some space savings in the longer term but with a marginal impact on the write latency.

2.1.7. Reliability

Since the goal of deduplication is redundancy reduction, data reliability is influenced. In particular, a loss of a single block may affect more than one file. Therefore, systems with deduplication employ techniques such as RAID, erasure coding, and replication to improve data reliability [94, 348]. Moreover, there are deduplication-specific techniques to increase reliability in such systems, for example, Per-File Parity [326], which increases the reliability of frequently referenced chunks through their selective replication.

Interestingly, there are scenarios when deduplication actually decreases the likelihood of data loss [186]. For instance, restoring redundancy of the most popular blocks with a higher priority largely increases their reliability [101].

2.1.8. Security

Deduplication also has important implications for data security. First of all, encryption of data can preclude its later deduplication, as the point of encryption is to make the data harder to identify. Therefore, encryption should be applied after deduplication is done, or a special encryption method compatible with deduplication must be used.

One of the most popular methods of combining encryption with deduplication is *convergent encryption* [55], which uses a hash derived from data as an encryption key. Therefore, multiple users can encrypt the data with the same key if they possess the same information to store. As convergent encryption has its flaws, there are numerous techniques to improve the method [3, 148, 160].

A different security threat arises in situations where deduplication is performed over data belonging to multiple users, possibly from different organizations that do not trust each other. For instance, there are side-channel attacks on systems with deduplication [32, 38]. A system with deduplication can leak information whether particular data have already been stored if writing a new instance of the data results in a lower latency in comparison to non-duplicates. Therefore, various techniques are proposed to mitigate such information leakages [250, 335].

2.1.9. Higher-Level Abstractions

Data management in systems with block-level deduplication is difficult for many reasons. First of all, a system with block-level deduplication keeps data as fine-grained blocks, whereas the user normally expects that the system provides a higher-level abstraction, such as POSIX files [138]. Consequently, complex layers are provided on top of such systems to organize blocks into higher-level data structures [92, 298].

Nevertheless, many operations typical for systems without deduplication are complicated when deduplication is introduced. For instance, a question about how much space a particular file consumes is ambiguous. Should the system return the size of the file before deduplication or the actual number of bytes consumed on a disk drive? How should the answer change for directories or other groups of files? Answering such questions not only requires an exact specification but also possibly computationally complex algorithms. Harnik et al. [123] describe a possible approach to approximate the space consumption calculation. The idea is further developed by GoSeed [221] and Kisous et al. [167] to answer more questions, like which files should be moved between deduplication systems to optimize capacity usage.

2.1.10. Resource Efficiency

To meet the user demand, a system with deduplication needs to provide high and consistent throughput for various workloads, simultaneously performing all other tasks resulting from the challenges described hitherto. To this end, dedicated techniques have been introduced to improve resource utilization for various workloads in systems with deduplication [6, 278, 280]. In particular, systems accessed by multiple different users require special techniques to improve resource utilization [179, 205].

2.1.11. Final Remarks

The challenges described hitherto are often discussed by researchers, but there are far more research topics related to deduplication. To give a few examples, DedupSearch [97] is a novel method of finding keywords in deduplicated data; Finesse [343] is an approach of mixing deduplication with delta compression to achieve even higher data reduction; APP-Dedupe [204] is a technique reducing the write amplification of SSD drives, as flash memory used in SSDs enforces rewriting some of the old data when writing. Finally, systems with deduplication take advantage of methods designed for other kinds of storage systems, which we will describe in Chapter 3. More examples of solutions related to data deduplication can be found in related surveys [201, 218, 248, 329].

2.2. Applications of Deduplication

The presented diversity of challenges arises largely because there are many applications of deduplication in the real world.

Data backup is a use case existing from the earliest years of deduplication-related research [255, 282]. Typical backup policies require storing multiple versions of data, for instance, daily backups for a week, weekly backups for a month, and monthly backups for a year [2, 93, 106]. As only a fraction of the data change day by day, the backup data reduction with deduplication is often higher than 90% [285]. Therefore, a significant part of the research concerns deduplication in the backup use case [102, 152, 172, 189, 192, 244, 288].

However, applications of deduplication are not only limited to backups. DupHunter [344] is a new architecture of a (Docker) container registry that leverages the predictability of user access patterns to implement highly efficient deduplication, which decreases both storage consumption and container image retrieval latency. Operating systems, such as the Windows Server [214], allow deduplication of users' data, just as ZFS [240] enables deduplication for systems from the UNIX family. Distributed storage systems, like Ceph [320], offer deduplication [234, 316] for primary storage use cases.

Methods of deduplication in primary storage often depend on the type of medium that stores the data. DeWrite [351] is an inline deduplication scheme that utilizes lightweight hashing to speed up non-volatile memory reads and writes, as well as the average non-volatile memory consumption. Similarly, CA-Dedupe [105] and CRFTL [340] implement deduplication in a flash translation layer of NAND flash memory to provide deduplication for a large number of devices, including SSD drives and smartphones. Yet other methods are used to implement deduplication in DRAM [314, 328].

To sum up, there are numerous practical applications of deduplication. In our research, we focus mostly on the backup use case, as we will explain in the rest of this chapter.

2.3. HYDRAstor

HYDRAstor [225] is a commercial scale-out storage system with deduplication for backup and archive produced by NEC [228]. Since we have adopted HYDRAstor as a platform for our research, in this section we introduce its most important features.

To start with, HYDRAstor supports a variety of system sizes: from small one-server Virtual Appliances [230] to multi-petabyte grids with thousands of hard drives [225]. The grid systems can be expanded with servers from multiple generations without disrupting the workloads [229]. The performance scales linearly with the number of servers and, in the largest configuration, HYDRAstor can write petabytes per hour.

Inline variable-length deduplication is one of the most important features of HYDRAstor [94]. Actually, HYDRAstor was the first commercial implementation of scalable global deduplication, that is, one that eliminates duplicates against data distributed across a multi-rack system. To maximize data reduction, HYDRAstor employs content-based chunking, further improved with techniques such as marker-filtering [227], combined with data compression.

HYDRAstor ensures high reliability and resiliency to failures. Erasure coding [40] splits blocks into fragments and generates redundant ones. Such fragments are distributed across multiple devices to provide high availability even in the event of failures. Self-management features, such as automatic data rebuilding after failures, additionally increase the reliability of the system [94]. Moreover, WAN-Optimized Replication [229] can be used to achieve recovery in case of major (datacenter-wide) disasters.

The architecture of HYDRAstor provides an API for accessing deduplicated blocks. Notably, a filesystem interface [298] is implemented on top of this API to provide local file access, as well as compatibility with the Network File System (NFS) and the Common Internet File System (CIFS) protocols. HYDRAstor's *Express I/O* [226] is another access protocol, which reduces overheads and maximizes data throughput. Moreover, *Deduped Transfer* delivers even higher throughput by performing parts of deduplication on external media servers. Finally, HYDRAstor's OpenStorage Optimized Synthetics and Accelerator allows integration with the popular Veritas NetBackup [312] application.

Independent of the interface over which blocks are accessed, the blocks in HYDRAstor are immutable. They are organized in directed acyclic graphs (DAGs), to form larger data collections like files [94] (see also Fig. 2.1). Besides data, each block can store references to other blocks. Special blocks called *searchable retention roots* are source vertices of the DAG, and blocks referenced from searchable retention roots together with their descendants are considered live. Searchable retention roots can be marked as deleted by writing an associated *deletion root*. Blocks without live references are eventually deleted by a garbage collection algorithm.

Concurrent Deletion [283] is a garbage collection algorithm implemented in HYDRAstor. It introduces the concept of *epochs* to distinguish which data are written before a deletion is started, and therefore to allow reading, writing, and removing data simultaneously, even in a multi-server configuration. Similarly to other garbage collection algorithms in systems with deduplication, it is executed periodically in background and can take over an hour to complete. Notably, the performance of Concurrent Deletion is scalable, i.e. the computation takes roughly the same time if both the number of servers and the size of data are doubled.

Since HYDRAstor uses HDDs for user data, two algorithms were proposed to mitigate the effect of fragmentation. The first is Content-based Rewriting (CBR) [151], which analyzes a level of fragmentation when data are written and rewrites selected blocks to new locations. The second is Limited Forward Knowledge cache (LFK) [152], which keeps selected blocks in

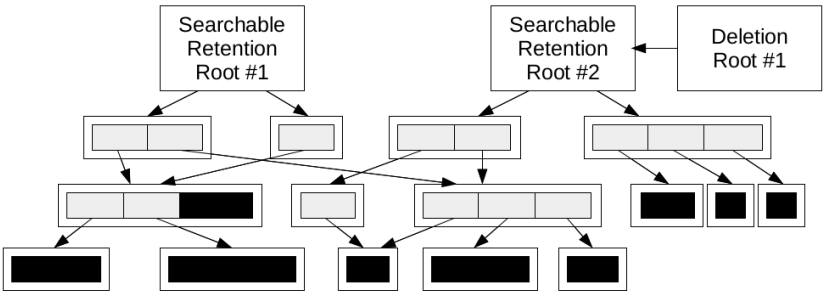


Figure 2.1: Block organization in HYDRAstor. A block can contain data (black) and/or references to other blocks (gray). Blocks with multiple incoming arrows are deduplicated. Descendants of *Searchable Retention Root #2* without references from *Searchable Retention Root #1* will be eventually deleted.

memory. According to an empirical evaluation [152], when both algorithms are combined, the restore speed can be comparable to reading non deduplicated data.

The portfolio of features offered by HYDRAstor is extensive and it includes many others not described here. In particular, there are important security and privacy functionalities, like Write-Once Read-Many (WORM) support, encryption, and data shredding [224]. Nevertheless, describing everything is beyond the scope of this dissertation.

2.4. Global System Assumptions

Systems with deduplication are a vast research area with numerous challenges and trade-offs. Therefore, in our research, we assume a particular system model. Our model is compatible with HYDRAstor, and therefore, we could implement our ideas in real-world systems deployed on several continents for many generations. However, the model is generally applicable to many other leading systems with deduplication, at least to some degree.

First of all, we assume a multi-server architecture with HDDs as the main storage medium. We do not assume any particular components like CPU models, RAM, or network, but we expect the hardware to meet the standards of average enterprise systems in the foreseeable future. For example, a system composed of ten servers with a 12-core CPU clocked at 2.40 GHz, 128 GB of RAM, multi-gigabit Ethernet, and 12x 6 TB HDDs meets our expectations. In our empirical evaluation, we used servers similar to this specification, and we describe our testbed for each experiment.

We assume that our system already implemented inline, block-level deduplication with multi-kilobyte blocks. We expect that chunking and fingerprinting algorithms are already optimized, and we do not go into their details. We also assume the blocks are organized into DAGs to form data collections such as files.

Our next assumption is that our system is designed primarily for storing backups, as this has a significant impact on the expected workloads. Accordingly, we assume that data will be written to the system mostly by backup applications, just as in real-world backup environments. Moreover, we expect the data to have characteristics that are typical for backups, for example, a high deduplication ratio between consecutive backups.

Regarding garbage collection, we do not assume any particular algorithm, but only that garbage collection is a long-running job that is executed periodically.

Finally, we assume that the system is highly reliable and stores data with erasure codes or similar techniques to ensure availability even in case of hardware failures.

Our system model overlaps significantly with many systems described in a survey on storage systems with deduplication by Paulo and Pereira [248].

Chapter 3

Relevant Work on Distributed Storage and Clouds

Distributed storage is a method of keeping and serving data in a system or a group of systems that consist of multiple devices, such as servers and data carriers. As distributing data across devices brings benefits such as increased reliability (a distributed system can be available even if some of its devices fail), and increased performance (combined devices can achieve higher throughput or more I/Os per second than a single device), distributed storage is widely used [81, 242, 243, 320].

In recent years, there has been a significant research attention towards distributed storage systems related to the *cloud* [58], which is a general term that describes resources, systems, and services available on demand. To explain the connection between clouds and distributed storage, and to provide background to our research, this chapter reviews important storage technologies as follows. Section 3.1 introduces the necessary historical context on storage systems. Section 3.2 surveys modern technologies of disks and other data carriers. Section 3.3 describes cloud computing and cloud storage and explains why the topic has been so important in research and industry in recent years. Section 3.4 discusses object storage, a modern data abstraction with skyrocketing popularity. Finally, Section 3.5 summarizes some of the most relevant recent publications in distributed storage and concludes.

3.1. Evolution of Storage Technologies

Data storage techniques constantly evolve to meet the market demand. In the last decades, storage system sizes have increased exponentially. In 1956, IBM 350 was released as one of the first hard disk drives [107]. Although IBM 350 physical dimensions dwarfed today's HDDs, its capacity was less than 4 MB. In the 1980s, disks such as IBM 3380 had roughly a 1000 times larger capacity than IBM 350. However, their performance (in terms of I/Os per second) did not increase between 1956 and 1980 as much as the capacity. Therefore, disks started to be grouped into arrays [245] to provide a higher number of operations per second.

The 1980s was also a time when techniques of coupling storage and network technologies were rapidly developed. Notably, the first distributed file systems were introduced [268, 294], including NFS [267], which continues to be developed and commonly deployed nowadays [124]. Further evolution of network technologies in the 1990s, such as the adoption of Fibre Channel [299], resulted in the introduction of new storage architectures, such as Storage Area Network [164], which is a specialized network that provides block-level access to disks. A different technological breakthrough of the 1990s was the introduction of the first solid-

state drive (SSD) [122], which offered orders of magnitude more random read operations per second than HDDs. However, the per-byte cost of SSDs was an order of magnitude higher than HDDs. Therefore, despite the superior performance, at the end of the twentieth century, massive distributed storage systems still utilized primarily HDDs and tape drives [65, 121]. Tape drives were especially appealing for storing backups [75], as tapes were several times cheaper than HDDs [116]. The situation changed in the first decade of the twenty-first century when the popularization of deduplication [94, 255, 348] made storing backups on HDDs economically justified.

At the time this dissertation is being written, HDDs are almost 1,000,000 times larger than in 1956 [322], and similarly, SSDs are almost 1000 times larger than in 1991 [269]. Just as in earlier decades, new types of data carriers are developed, and similarly, novel algorithms and system architectures that leverage the newest technology are invented.

3.2. Modern Data Carriers

For decades, computer systems have employed diverse data carriers [251], as different physical properties of devices offer different cost-performance trade-offs. A modern processor typically has L1 and L2 caches, which offer latencies below ten nanoseconds but have a very limited size (e.g., below 1 MB) [57]. Next, there are L3 caches and DRAM, which are an order of magnitude slower than L1 and L2 caches but are also much cheaper, and hence, modern servers can contain terabytes of RAM [1]. Caches and DRAM are typically volatile, which means data are lost when the device is disconnected from the power supply. That is why storing data for a longer time in caches and DRAM is not only expensive but also difficult. Therefore, only specialized solutions such as in-memory databases [63, 100], keep data solely in caches and DRAM.

Memory carriers are actively researched for the purpose of providing non-volatility by, for instance, employing carbon nanotubes instead of silica [104]. One of the promising non-volatile memory technologies called *3D XPoint*, was released for production in 2017 as Intel Optane [144]. Unfortunately, in 2022 Intel announced plans to cease future development of Optane to focus on the development of other products [143]. Nevertheless, the cost of 3D XPoint was still much higher than SSDs, not to mention other carriers like HDDs and tapes.

As tapes are still more affordable than HDDs, and HDDs are still less expensive than SSDs, whereas the performance relationship of these carriers is inversed, currently all three types of data carriers are applied [261]. These carriers are also actively developed. For instance, a modern LTO-9 tape format allows to store 18 TB of uncompressed data, but in the future LTO-14 is expected to provide 576 TB per tape [297] (one generation of LTO typically lasts for 2 to 3 years). Similarly, modern HDDs offer a capacity of up to 22 TB [322], but HDDs of 100 TB and more are already planned [219] as techniques to increase the areal density of platters are researched. One of the techniques available today to increase the density of HDDs is overlapping adjacent tracks during writing in shingled magnetic recording (SMR) drives [324]. SMRs typically have a 25% larger capacity than regular HDDs but writing tracks so close to each other requires write-amplification (i.e. rewriting data that was not recently modified), as adjacent tracks need to be often rewritten. Multi-terabyte SSDs are also available [266], and SSDs are expected to grow in the next years, not only due to the increased number of bits stored per cell [147] but also because new form factors such as EDSFF enable increasing storage density [166].

Looking at the roadmaps of tapes, HDDs, and SSDs, we can infer that all three data carriers will be developed and present on the market for at least a couple of years. The

challenge is, however, to utilize the data carrier most appropriate for a specific application and also to adapt systems to continuously improved hardware. At least until new types of data carriers replace the existing ones. For instance, DNA storage offers an order of magnitude higher areal density than any of the current data carriers, but at present, the technology is immature: the first automated DNA storage presented in 2019 [290] was capable of writing and reading data only in 5-byte cycles which take 21 hours [290]. Similarly, quartz glass [29] is a promising candidate for a storage medium but so far, glass-based storage is not commercially available. It is an open question whether new storage medium will dominate the market in the next decade [73]. What is certain, however, is that cloud storage, which offers services built on top of various storage devices, will have a large share of the market. Interestingly, in the case of public clouds, a customer often receives no information on what data carrier is employed for the implementation of the storage services. For instance, there is no public information on how AWS Glacier is built [80]. However, this does not prevent organizations from utilizing it successfully.

3.3. Cloud Computing and Storage

Cloud computing is a model of providing services and infrastructure in a highly scalable and dynamic manner [287]. The idea of cloud computing was introduced in the 1960s [103], but arguably, the first cloud computing services in today's sense were introduced in 2002 by Amazon [11]. In fact, the cloud started to become a market standard years later when companies like Microsoft [296] and Google [76] also began offering their clouds. Currently, the global public cloud market is expected to grow from $445 billion in 2022 to $988 billion in 2027 [258], and the majority of the market is held by Amazon, Microsoft, and Google [289], which are often referred to as hyperscalers [53]. However, the whole cloud market is even bigger and more versatile, because the public cloud is not the only available type of cloud. For instance, there are private clouds that are implemented for particular companies or institutions. Private clouds that are prepared for government are often called government clouds [187], as they require special features to provide security and privacy to implement systems such as electronic voting [349]. In turn, hybrid clouds are a mix of both private and public clouds [114].

The multiplicity of cloud services, their pricing model, and the potentiality to outsource IT work and become more flexible [327], convince companies to employ clouds. For instance, during the launch of Pokemon GO (which is a game for smartphones), the number of players, and therefore the data traffic exceeded expectations by an order of magnitude [198]. However, the implementation of the game server in the cloud facilitated scaling, and game developers were able to meet players' demands. Similarly, in scientific use cases such as high-performance computing (HPC), the cloud can offer easy access to a variety of hardware configurations [117]. Finally, clouds are also used by governments to provide the required resilience, security, and continuity in the face of emergencies [168]. Nevertheless, despite its advantages, the cloud model also brings several risks and threats. In the cloud, data is often kept and processed by a third party, which brings many questions regarding data confidentiality, integrity, and availability [10]. In turn, pay-as-you-go pricing introduces business risks, such as the possibility of excessive or uncontrolled spending [215].

Each cloud offers a variety of computing, network, and storage resources, as well as platforms and services built on top of the resources [39]. For instance, Amazon Web Services (AWS) customers can choose between virtual machines and containers available billed hourly [263], and many other services, such as the serverless lambdas (e.g., AWS Lambda

[246]) that completely hides physical resources as a customer is billed per function execution. Clouds typically also offer ready-to-use products, such as databases [277], key management services [8], and load balancers [279].

Cloud storage services are also diverse and provide varying interfaces, latency, and pricing. One extreme is network-attached SSDs providing a high performance for I/O-intensive workloads [44]. Another extreme, Glacier Deep Archive [43], allows for storing data for less than one dollar per terabyte per month but can require 12 hours to restore even a single byte. Some of the storage products, such as Elastic File System (EFS) [15] provide a storage interface (in the case of EFS, it is a classic filesystem interface) and dynamically scale the system based on customer demand. One of the most popular interfaces for that kind of use was the object storage interface, developed particularly for clouds, as described in detail in the next section. However, object storage is not the only storage abstraction designed for cloud usage. For instance, in recent years, there has been ongoing research on providing storage for serverless lambdas [64, 210].

3.4. Object Storage Interface

Object storage, such as Amazon S3 [12] or Microsoft Azure Blob Storage [126], has become a highly popular and versatile storage abstraction. It organizes unstructured data as objects that are grouped into buckets. Apart from the data themselves, each object normally comprises up to a few kilobytes of metadata, including a key identifying it within its bucket. These storage primitives can be accessed via an HTTP-based interface following REST principles: reading objects/buckets is done with HTTP GETs and HEADs, uploading with PUTs, deleting with DELETEs, and so on. Therefore, data can be easily accessed via REST API, even if an application using the interface and the object storage server are distant from each other as in a wide area network.

The demand for object storage abstraction is immense, as the simple structure and HTTP communication make it a good fit for cloud applications. In 2017, it was estimated that over 30% of data center capacity was in object stores [188]. In 2021, in turn, Amazon alone stored over 100 trillion objects in S3 [45]. Object storage interfaces are provided by hyperscalers [5, 213], other public clouds [36], on-premise enterprise storage systems [130], and open-source solutions [67, 142, 236]. What is important is that the on-premise (private cloud) solutions are just as popular as the public clouds. For instance, MinIO, which is one of the leading on-premise object storage, reported that a docker image with their object storage was downloaded over a billion times [217], and 75% of Fortune 100 companies run MinIO.

Likewise, object storage is utilized in diverse applications. For instance, video companies such as Netflix [9] and Dreamworks [231], and social media companies such as Pinterest [13], employ object storage to store videos. Similarly, object storage is employed to store game assets [206], or gamers' results [98]. Another application is keeping in object storage data sets for big data processing [212, 260]. Finally, popular use case example is storing backups [42, 88].

Object storage is implemented in systems that utilize various data carriers, including SSDs, HDDs, and tapes [180]. Backblaze, which is one of the popular object storage providers, reports that their data center has almost 100x more HDDs than SSDs [37]. To decrease storage costs even further, some cloud providers use SMRs [347]. Clouds often provide dedicated object storage services for data with a decreased reading frequency, such as Amazon Glacier [313] that offer large discounts for storing data but charge extra for restores. With that kind of service, object storage is an affordable option for rarely accessed data kept for a long time.

Despite having a few flavors, object storage interfaces are largely similar [109, 237]: an object storage interface is often described as S3-compatible or Swift-compatible, names originating from Amazon S3 and OpenStack Object Storage (Swift), respectively. In particular, MinIO [142], describes its interface as S3-compatible. RadosGW (Ceph Object Gateway) [67], which is in turn an object layer for the widely-adopted Ceph [67], supports both flavors. Finally, even OpenStack Object Storage [236], the original implementation of the Swift interface, now also incorporates middleware that emulates S3 [235]. Support for either of the interfaces in a backup appliance can thus be extended to other object storage interfaces.

3.5. Final Remarks

Distributed storage is a wide research area, with scientific publications also covering topics different than the cloud, effective use of particular data carriers, and deduplication. Each year, over a thousand publications on distributed storage appear [139], and whereas the summation of all of them is beyond the scope of this dissertation, we will summarize some of the most frequently researched topics that may help position our work.

First of all, the reliability of the storage system is crucial. Therefore, many publications analyze metrics such as failure rates in large-scale, production environments [120, 203, 333]. Based on the analysis of real-world failures, new techniques to improve reliability are introduced [155, 196]. Similarly, popular methods of increasing failure resilience, such as erasure codes, are revised [156, 185].

Large production installations are investigated not only in terms of their performance and general functionalities. For instance, Facebook published conclusions from the analysis of their exabyte scale Tectonic [243]. Similarly, Alibaba presented a modernized version of their exabyte scale Pangu [184]. Researchers from National Supercomputing Center in Wuxi the storage challenges on Sunway TaihuLight [149], which in 2019 was the third most powerful supercomputer in the world. Each of the publications outlines the problems that appear in practice in such systems and proposes novel approaches to improving the system performance.

One of the most common approaches in storage systems is caching selected data in faster-access layers. Despite the fact that caching techniques have been researched for decades [48, 252], new results regarding the usage of caches are published every year [195, 284]. The appearance of new data carriers is one of the reasons why new caching algorithms are necessary, as modern hardware, such as persistence memory, brings new challenges [325].

Similarly, file systems are adapted to support modern hardware such as SSDs [178, 346] and persistent memory [223, 323]. In fact, many other different aspects of filesystems are researched as well. For instance, new methods of journaling are still being investigated [165, 233]. Regardless of decades of filesystem research [207], there are still many open problems, even seemingly simple, like a proper handling of the case-sensitivity of file names [46]. Considering the popularity of the filesystem abstraction, it is unlikely that object storage will completely replace file systems in the foreseeable future.

Another interface that has received much attention recently is a key-value store [177] that simply allows querying, adding, and deleting keys associated with values. Key-value stores often utilize log-structured merge trees (LSMT) [41, 140, 175], which feature amortized $O(1)$ writes. An interesting improvement in key-value stores proposed recently [84, 183] is the usage of a so-called learned index [170], which employs machine learning to optimize the system's performance.

Finally, there is vast research in optimizing specific use cases of storage. Graph processing has many practical applications, and hence, many storage systems focus on providing low

latency and high throughput for operations on graphs [173, 334]. Similarly, storage systems are optimized for boosting machine learning applications [163, 194]. Yet another important problem is the optimization of storage for personal devices, such as smartphones [262, 350].

To summarize, this dissertation focuses on solving important problems in distributed storage systems with deduplication, especially for the use case of storing backups. However, the problem of efficiently storing data is a very broad one. Some of the issues it involves, such as keeping data resiliently, are general and applicable to most storage systems. Others, however, are very specific and suit only peculiar storage applications.

Chapter 4

ObjDedup: Backup Appliances with Deduplication as Object Stores

Object storage has recently become a widely adopted solution in the backup market. There are many properties that make this storage abstraction attractive for these applications. In particular, provided as a cloud service, object storage offers a convenient way of dependably keeping backups off-site, as its interfaces contain provisions for transferring large amounts of data via wide-area networks. They also display design concerns over security, including strong ransomware protection. Consequently, novel backup solutions utilizing cloud-hosted object stores as backends have appeared [42, 88]. Likewise, systems that internally back up their state (e.g., databases or analytic platforms [77, 293]) have added support for object storage. Finally, leading backup applications—originally targeting dedicated backup appliances as storage backends—have started integrating with Amazon S3 and similar object storage interfaces [301, 308].

In this light, there is a strong market incentive also to have dedicated backup appliances implement these interfaces. In particular, appliances that can significantly add much value as backends for object-storage-compatible backup applications are those supporting data deduplication. This is because backups inherently contain data that are repeating over time, thereby yielding high deduplication ratios [211]. Notably, an appliance offering global block-level in-line deduplication can reduce the storage footprint of data written independently by the different applications, even if backup applications internally implement their own form of deduplication. In combination with simplified regulatory compliance and higher control over data stored by on-premise machines, these compelling space savings and compatibility with cloud-based storage backends are the primary motivations behind adding object storage interfaces to backup appliances.

However, it is unclear how a backup appliance with deduplication should implement such an interface so as to offer adequate performance. Normally, for external applications, backup appliances provide dedicated data-transfer interfaces, such as Common Internet File System (CIFS), Virtual Tape Library (VTL), and others [254], which differ significantly from object storage interfaces. Although objects and buckets resemble files and directories from file systems, these primitives do differ. Likewise, object storage is designed for different access patterns than backup appliances. For instance, Amazon recommends relatively small objects: if an object is to exceed 100 MB, it should be uploaded in multiple parts for better throughput and recovery from network issues [12]. Accordingly, backup applications for object storage backends typically organize a backup into numerous small objects (1–64 MB) [78, 141, 308]. In contrast, leading backup applications for dedicated appliances use large files ($\geq$100 GBs) [315].

The bottom line is that backup appliances with deduplication may not be prepared to handle the volumes of metadata due to both the specific functionality and usage patterns of object storage interfaces.

In this chapter, we thus investigate the problem of efficiently implementing an object storage interface in state-of-the-art backup appliances with global block-level in-line deduplication. To this end, we take the following "pragmatic" approach. Since the considered appliances are a mature, complex, and highly optimized technology, we do not aim to redesign any of their internal functionality but only to add new functionality. Likewise, as a single appliance normally offers multiple backup interfaces at once, and data written via any of those are globally deduplicated (i.e., also against data written via others), when adding support for an object storage interface, we must preserve this behavior.

Our approach is further reinforced by the fact that many steps of data deduplication are independent of the particular interface, and a backup appliance is itself often a distributed storage system, ranging from a few to as many as thousands of machines. Such a system implements a number of features that work across all interfaces, such as ensuring the quality of service, preventing premature exhaustion of storage space, or controlling data resilience. Implementing these features as a single block-level storage engine that is shared by all exported interfaces is a common practice.

A consequence of this approach is that, similarly to the classic backup interfaces, an object storage interface should be provided as a layer over the block-level engine of a distributed storage system with deduplication. On the one hand, this decreases the number of design decisions required during our study, because effective solutions for many problems are already available. On the other hand, it poses novel problems because adapting some techniques that are popular in object storage without deduplication is not possible, and we needed to propose new dedicated solutions to achieve satisfactory performance.

Given this approach, the major contributions of the chapter are twofold. First, we present a *preliminary study* that aims to identify particular issues an implementation of an object storage interface for a backup appliance with global block-level in-line deduplication has to address. Based on data from 686 real-world deployments of our backup system, we extract statistical information characterizing their usage patterns. With this information, we analyze commonly used object interfaces to identify requirements, meeting which may be challenging in a system with global block-level in-line deduplication. Second, based on the study, we identify core algorithmic problems and propose our solutions to these problems, that is, *algorithms and data structures*, dubbed ObjDedup, which can be employed to provide object storage functionality efficiently as a layer on top of a block-level engine of a backup appliance. We also outline our implementation of this design for HYDRAstor and evaluate it experimentally. The evaluation indicates, among others, that the presented solutions can outperform the state of the art multiple times in terms of I/O operation throughput.

The rest of the chapter is organized as follows. Section 4.1 provides the necessary background and surveys related work. Section 4.2 contains our preliminary study based on real-world deployment data. Section 4.3 introduces the algorithmic core of our solution. Sections 4.4 and 4.5 discuss, respectively, the implementation and the experimental evaluation of the solution. Section 4.6 concludes.

4.1. Background and Related Work

As explained in Chapter 2, deduplication has multiple applications but it is particularly appealing for backup systems. In this section, we give a high-level outline of the operation of

such systems, emphasizing aspects that are the most relevant to the implementation of object storage.

4.1.1. Global In-Line Block-Level Deduplication

Our work focuses on a model referred to as *global in-line block-level deduplication*, as it is the state of the art for backup appliances (see Chapter 2). Although deduplication-related research focuses largely on improving the deduplication operations [6, 86] or reorganizing data to reduce fragmentation [61, 62, 151, 189], our research addresses an orthogonal problem: an effective implementation of object storage interfaces on top of a deduplication system that already has its block maintenance optimized. In general, state-of-the-art backup appliances solve numerous problems (e.g., block caching, ensuring the quality of service, balancing space utilization, or controlling data redundancy). Therefore, their block maintenance (including deduplication) is encapsulated into an engine that offers a block-level interface, allowing for reading and writing blocks as well as querying their presence. The higher layers, such as CIFS or VTL, are implemented on top of this block-level engine, treating it mainly as a black box, which is also the approach we follow in ObjDedup.

Consequently, the exact solutions the block-level engine employs to the aforementioned problems are largely abstracted out and should be irrelevant for the higher layers, including ObjDedup. However, the data must be eventually stored by the backup appliance in 2KB–128KB blocks using a well-defined data organization.

4.1.2. Deduplicated Data Organization

More specifically, the following data organization is commonly adopted in deduplication appliances and hence is also assumed in the design of ObjDedup.

All blocks are immutable. If block contents were allowed to change after a fingerprint computation, the contents of a block could be lost forever if it was deduplicated against another block that was modified later or, conversely, two blocks with ultimately different contents could have the same fingerprint. The fingerprint of a block is thus normally used as (an element of) the unique address of the block.

Blocks are organized into directed acyclic graphs (DAGs). Since a single block is typically too small to represent an entire data collection, such as a file or a directory, there must be means of grouping multiple related blocks. Therefore, blocks form DAGs as explained in Chapter 2. Since block data are immutable and since the address of a block contains a fingerprint computed over both the data and references constituting the block, **the references must be immutable as well**. As a result, changing a reference in some block deep in a DAG entails generating a new block with a new address, replacing references in all its ancestors, so that the change propagates up to the root blocks.

Blocks with no live references are eventually deleted to reclaim storage space. **Deleting blocks in a system with in-line deduplication requires considerable additional effort** to prevent situations where new blocks reference data that have been deleted. Typically, the system employs a multi-phase algorithm that follows a garbage collection technique such as mark-and-sweep or reference counting [92, 283].

4.1.3. Deduplication in Object Storage

Despite the popularity of object storage, there has been little work on how such support can be provided efficiently in a backup appliance with deduplication.

To start with, Cloud Tier [95] moves data written to a backup appliance (using an interface different than object storage) to cloud-based object stores. Such object stores do not provide deduplication, so the backup appliance deduplicates data before transferring them to the cloud. The solution is much different from ObjDedup, which extends the backup appliance itself with the object storage interface.

Several publications proposed adding deduplication to existing object storage systems, notably Swift [237] and Ceph [67].

Post-process deduplication approaches, like Ceph's deduplication [234] or LOFS [74], are very different from ObjDedup, which assumes in-line deduplication in the underlying block-level engine. In-line deduplication, if exploited well, can inherently offer superior write throughput and space savings for highly duplicated backup data [248].

In-line deduplication approaches proposed to date do not examine the problem of efficiently managing metadata due to supporting object storage interfaces. In particular, Wang et al. [316] focus on classic files in Ceph and explicitly mention support for Ceph Object Gateway as future work. Similarly, Khan et al. [161, 162] provide deduplication for Ceph's internal objects, which are different from object storage objects, supported in Ceph Object Gateway. In other words, rather than tackling the problems attacked by ObjDedup, that research explains how a variant of the black-box part of ObjDedup (i.e., the block-level engine) could be implemented in Ceph. Those ideas are further improved by CROCUS [119], which schedules deduplication-related operations onto CPUs and GPUs.

In contrast, the aforementioned Ceph Object Gateway requires bucket indexes [68] that are frequently accessed and modified. Therefore, they can incur a significant overhead if kept in a store with in-line deduplication. Yet, we are not aware of any relevant prior performance results for Ceph Object Gateway, and generally, the published results are insufficient to predict how the solutions would behave with large numbers of objects or small files, as generated by backup applications for object storage backends.

Finally, DedupeSwift [200] adds deduplication to Swift. In DedupeSwift, objects are stored as binary files, and metadata are stored in xattrs, so there is no dedicated metadata structure like in ObjDedup. DedupeSwift's throughput tops 10.54–25.51MB/s even with SSDs for deduplication caches, which is insufficient for a commercial backup appliance.

In conclusion, we are not aware of any in-depth analysis of the problems posed by a high-performance implementation of an object storage interface for a state-of-the-art backup appliance with global block-level in-line deduplication.

4.2. Preliminary Study

To gain more insight into the problems, we have conducted a study contrasting real-world usage patterns of backup appliances featuring global block-level in-line deduplication with the relevant properties of object storage interfaces.

4.2.1. Object Storage API Analysis

Certain features of object storage interfaces have become a market standard, and understanding what backup applications can expect is a part of our research. Because of space constraints, here we focus only on those features that are the most vital for the considered applications. When it comes to storage organization, a crucial property is that, apart from the data themselves, each object (and bucket) has associated metadata. The metadata of an object contains a key that uniquely identifies the object within its bucket, metainformation on the object's data (e.g., length, MD5 digest), and user-defined metadata. The total size

of the metadata can vary and—compared to the size of object data—can be significant. For instance, in Amazon S3, a key is up to 1 KB, and user-defined metadata are up to 2 KB [12].

Likewise, although the basic commands follow REST principles, object storage interfaces are extensive. For example, Amazon S3 currently has almost 100 commands, of which some have no counterparts in POSIX file systems. In particular, besides object management requests, such as PutObject, DeleteObject, ListObjects, there exist commands related to multi-part uploads, tiering, replication, security (e.g., encryption, ACLs, ownership control) and the like. Virtually all these commands access object metadata.

Another feature with far-reaching consequences is the usage of key prefixes. First, objects can be listed given a key prefix and a delimiter. In effect, even though the bucket-object hierarchy has just two levels, a deeper, directory-like structure of a classic file system is often recreated by organizing objects through their key prefixes. For example, listing objects with delimiter "`/`" and prefix "`mydir/`" is similar to calling "`ls`" in "`mydir`" of a file system. There are, however, some differences from classic file systems. A major one is that object listings are limited in size (in Amazon S3, to 1000 objects), which entails multiple invocations for prefixes with large numbers of objects. Second, prefixes are utilized for guaranteeing and scaling performance. For instance, Google Cloud Storage initially offers 1000–5000 requests per prefix per second. If the actual number surges dramatically for a prefix, some time may be needed for reorganization, during which the performance is lower [110]. Moreover, as object storage interfaces originally assumed wide-area networks, they promote moving large data in smaller parts. For objects larger than 5MB, multi-part upload (MPU) is recommended, with object transfer split into up to 10,000 parts. MPU state needs to be tracked by the backend because the parts can be provided in any order, and uploads are done in parallel and reattempted if required.

Finally, when operating with object storage interfaces, modern backup applications do make use of their features, notably wide-area-network- or security-oriented provisions. Even though not every application utilizes all commands of an interface, their deep understanding is necessary when developing support for object storage, especially since the market is constantly evolving and the demand is changing. For instance, in recent years, some backup applications have started using object locks as a protection mechanism aimed to prevent unintentional data updates and deletes [31, 222]. Another example is that some backup applications avoid MPUs (e.g., by uploading backups in objects below 5 MB [141]), but others do utilize this feature [56]. In either case, however, the typical object size in such applications is between 1 MB and 64 MB [78, 141, 308]. In other words, even if the data fed to a backup application for an object storage interface are the same as the data fed to a backup application for traditional backup interfaces, in the first case, the storage backend will receive data collections that are several orders of magnitude smaller than in the second.

4.2.2. Backup Data Pattern Analysis

To contrast these observations on object storage interfaces with the usage patterns of backup appliances with global block-level in-line deduplication, we analyze real-world deployments of such systems. Data are written to a backup appliance in various patterns in backup jobs [6], and the jobs are run periodically (e.g., once a day at a specific time) [304] based on backup life cycles and policies. Even a single backup application can write hundreds of jobs each week to back up diverse servers and business applications [27]. Similarly, deletion of data from each job is done based on a retention policy (e.g., after five days) [305], but, as mentioned previously, garbage collection under deduplication requires significant work and is thus executed sparingly, at most a few times a week. Therefore, we examine how a system

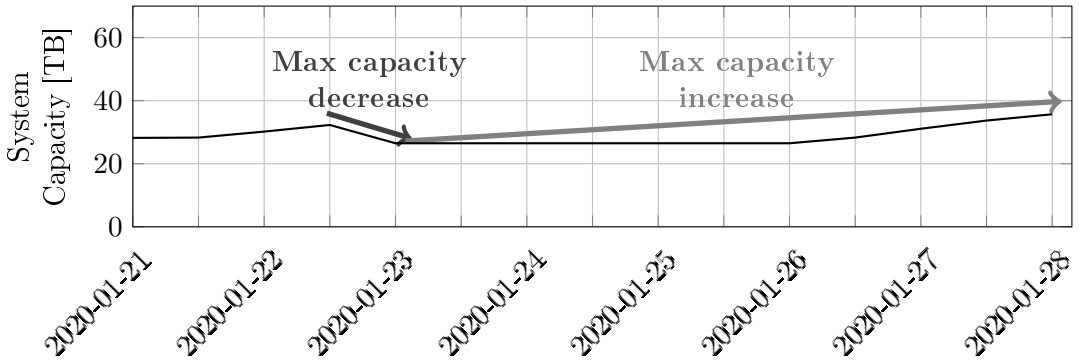


Figure 4.1: The evolution of capacity utilization in a representative sample.

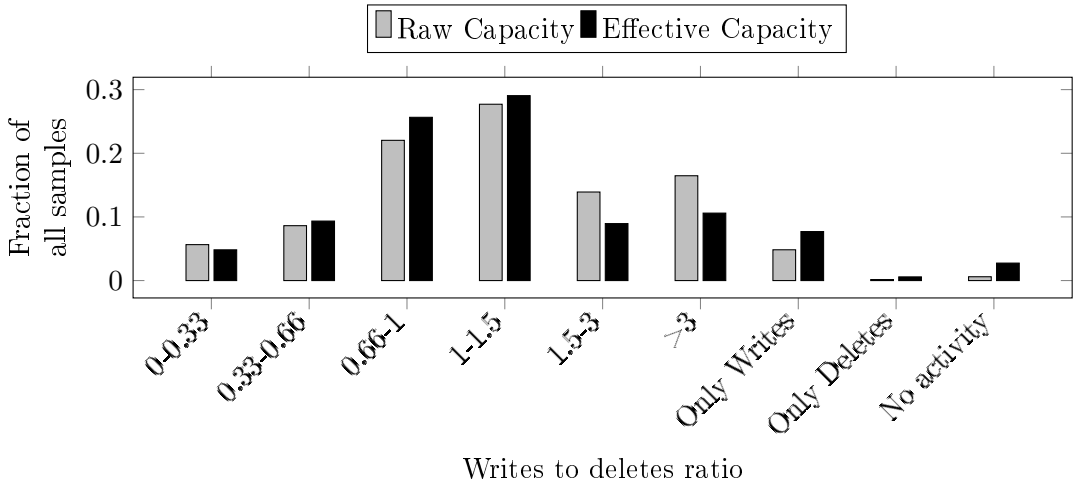


Figure 4.2: The ratio of weekly maximal increase and decrease of raw capacity (after deduplication) and effective capacity (before deduplication).

behaves based on information collected over a week, which we refer to as a *sample*. More specifically, we analyze 13,102 samples collected from 686 commercial deployments of our backup appliances and present those results that have had the most impact on the design of ObjDedup. While this is yet to be confirmed empirically when the solutions introduced in this chapter are massively adopted, we expect that object storage interfaces would not alter the observed patterns significantly, as the backup life cycles, policies, and data themselves are largely independent of a backup interface.

For every sample, we calculate the ratio of the maximal increase and decrease of capacity utilization (cf. Fig. 4.1). We examine changes in both: *raw* capacity (i.e., data physically stored after deduplication) and *effective* capacity (i.e., data written by backup application, before being deduplicated). The distribution of this value for all samples is plotted in Fig. 4.2. A majority of samples have their ratios above 1.0, that is, writes exceed deletes, which is expected given the continuous worldwide data growth [135]. Typically, the ratio is in the range 0.66–1.5, so every week similar amounts of data are added and deleted. Samples with no activity are virtually nonexistent, which implies that backups are indeed done regularly.

The magnitude of changes to raw and effective capacity utilization is shown in Fig. 4.3, separately for increases and decreases. It can be observed that capacity utilization in a system can change a lot in a week. The changes in effective capacity have even higher magnitudes

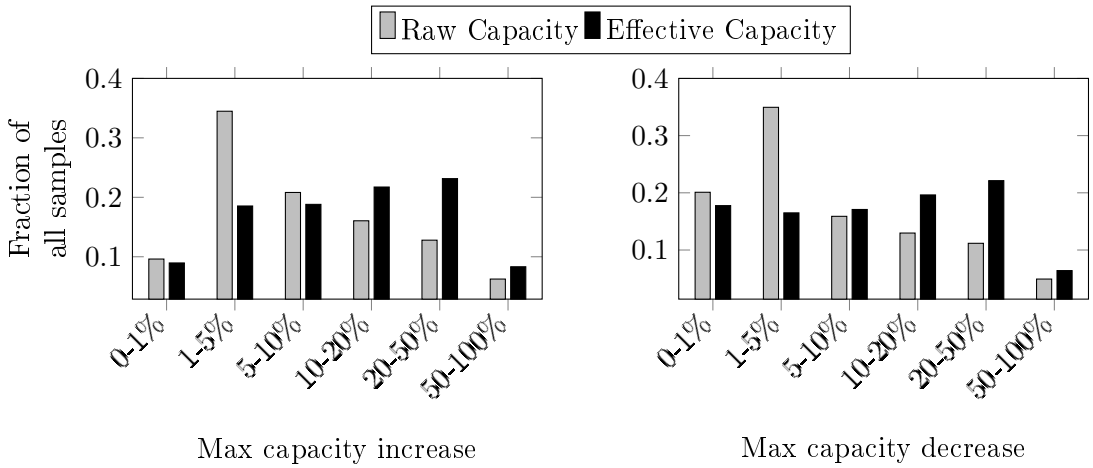


Figure 4.3: Maximal positive and negative capacity utilization changes within a week.

than in raw capacity. Given that effective capacity can be larger by an order of magnitude than raw capacity, this means that despite fairly stable capacity utilization (per Fig. 4.2), the data turnover is considerable: older backups are removed to store fresh ones.

Finally, Fig. 4.4 depicts the distribution of maximal capacity utilization in samples. It shows that although some fraction of free space usually remains in a system, in 16% of cases, the capacity utilization exceeds 80%. Therefore, considering the possibility of substantial ($\geq$20%) utilization increases (Fig. 4.3), the system indeed relies on efficient garbage collection.

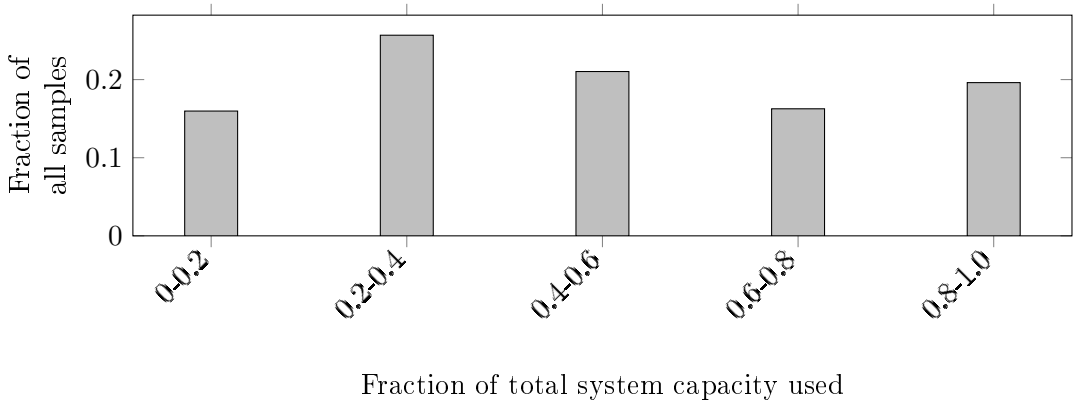


Figure 4.4: The weekly maximal utilization of system capacity.

4.2.3. Main Lessons Learned

The following major lessons can be drawn from our study.

Backup appliances and object stores have dissimilar characteristics. Backup appliances are optimized for write throughput, which is crucial given that their overwhelmingly dominant usage pattern is writing long data streams. Operating on individual data collections, in turn, hardly ever takes place, as even recovery typically concerns entire snapshots and is considered rather sporadic; the same applies to modifying metadata of existing collections. In contrast, object storage interfaces have been designed for flexibly organizing, efficiently accessing, and remotely managing large numbers of data items, so as to cover many use cases that may be encountered in the plethora of possible cloud-oriented applications. To this end,

object storage features an extensive API, rich system- and user-defined metadata enabling this API, and additional provisions for wide-area communication, scalability, security, and the like. It also requires respecting peculiar constraints regarding shaping the organization of the data and access traffic.

Dynamic and relatively large metadata of an object are problematic under immutable deduplicated blocks. For an object, the size of its metadata can be significant compared to the relatively small size of the data themselves, as recommended by object interfaces and respected by backup applications for object stores. In backup appliances, this phenomenon is likely to be aggravated since the data are normally repeating over time, and hence are often deduplicated. Furthermore, their high weekly replacement rate implies that many objects will be written and deleted every week. Each such operation on an object also requires at least one access to its metadata. In general, many object operations solely affect the metadata, updating them in some way. This is problematic given the block immutability in backup appliances with global in-line block-level deduplication. What is more, some object storage operations require tracking their progress by the backend. Such an operation generates metadata that are heavily accessed for a short time and are deleted afterward but need to be kept persistently in the backend to allow completing the operation even under transient failures. A prominent example is the aforementioned multi-part upload (MPU), which can produce thousands of metadata items for a single object.

Efficiency of metadata management is a fundamental problem on the scale of an entire backup appliance. This is due to the assumptions object storage interfaces make regarding the use of object key prefixes for collective operations, such as object listing, and for performance scaling. In particular, the listing feature implies that the metadata of all objects should be somehow indexed or sorted for efficiency. The potential solutions are further constrained by the fact that the delimiter of subsequent prefix parts is provided on demand and can thus be arbitrarily changed at runtime, even between requests. Guaranteeing performance and scalability, in turn, requires dynamically distributing the load on various objects between machines, for instance, based on the indexes. This implies that the algorithms for managing object metadata have to be able to work in a distributed fashion and handle partial failures.

Object metadata management solutions must not impair the performance of space reclamation. As revealed by our study, the high weekly data replacement rate in backup appliances already entails extensive use of block deletion and garbage collection. Supporting an object storage interface will likely increase the pressure on these mechanisms. This is because any update to metadata stored in immutable blocks typically invalidates these blocks as blocks with the new version of the metadata are written. Therefore, when addressing the previous problems, any implications on space reclamation must be carefully considered so that its efficiency is not impaired. In particular, adding an object storage interface to a backup appliance must not lead to situations in which blocks that are no longer necessary are not garbage-collected as soon as possible because of some dangling references, for instance, due to metadata indexing.

4.3. The Design of ObjDedup

In this section, we translate the conclusions from our preliminary study into algorithmic problems and present solutions to these problems, which we dubbed collectively ObjDedup.

4.3.1. Problem Statement

As explained previously, a backup appliance with global block-level in-line deduplication typically exports multiple well-established interfaces that are utilized by external backup applications, possibly at the same time. Internally, in turn, it is usually implemented as a distributed system that encapsulates the core functionality of deduplicated write-optimized fault-tolerant storage into a block-level engine, which is often a product of many years of development and fine tuning. The external interfaces are simply implemented as higher layers on top of this shared engine. We thus assume the object storage interface to be provided in the same manner. This assumption imposes a few constraints on our solutions, the major ones being:

1. The block-level engine must not be changed so as to avoid affecting the operation of the other interfaces exported by the appliance.

2. Likewise, extra hardware, such as additional machines or custom storage devices, must not be required from the appliance to support the new functionality.

3. The performance of the object storage interface, notably write throughput, space utilization, and fault tolerance, must be comparable to that of the classic interfaces.

Under these constraints, we consider the following overall design of ObjDedup. Objects and buckets are organized as other data collections (e.g., files and directories): into logical block trees within the block storage, with the root block representing a particular object or bucket and regular blocks holding the data of the object/bucket. In effect, the existing, highly-optimized pipeline can be utilized for writing object and bucket data, which allows for ensuring the same performance of these operations as for the other interfaces of the backup appliance. Object/bucket metadata are also kept in regular blocks within the block storage. Although an alternative design involving dedicated hardware for the metadata, like SSDs or non-volatile memories, could improve the performance of operations on the metadata, it would violate the previously formulated constraints. Moreover, storing the metadata within the block storage is essential for fault tolerance: if a machine responsible for a particular portion of the metadata fails, other machines can take over, as the block engine ensures that metadata are stored redundantly in the block storage and are available to all machines comprising the appliance. In contrast, what is different in the case of metadata compared to data is that because of the way object storage uses key prefixes in multiple operations and performance scaling, the metadata must be indexed and/or sorted by object/bucket key prefixes.

The central algorithmic problem that has to be solved can thus be formulated as follows:

> *How to efficiently organize object and bucket metadata by key prefixes and dynamically manage this organization by multiple processes given shared deduplicated write-optimized fault-tolerant immutable-block storage?*

Efficiency in this context has two facets.

First, the achievable throughput of the object storage interface must be comparable to that of the classic interfaces of the backup appliance, ensuring among others that while accesses to metadata are write-optimized, the performance of reads is not impaired. More specifically, as HDDs are assumed as the main storage medium, achieving a high write throughput is possible only if random disk I/Os are limited. Since blocks are immutable, updating metadata structures requires both reading some already stored blocks and writing new ones. Whereas the block-level engine batches writes, random reads may easily exhaust HDD capabilities.

Therefore, our asymptotic complexity goal for the number of reads necessary to update the metadata of an object/bucket is $O\big(log(n)\big)$, where n is the total number of objects and buckets in the store. We also want to ensure that in the case of updating the metadata of u objects sharing the same prefix, the complexity is $O\big(log_s(n) \cdot \frac{u}{s}\big)$, where s is the expected number of object metadata entries per block.

Second, the storage space of the appliance must be used efficiently as well. Not only does this mean that the storage overhead on metadata should be limited, preferably to $O\big(log(n)\big)$, but also, what is particularly important in a system with deduplication, that deleted blocks containing references to other blocks should be garbage-collectible without an unacceptably long delay. To be more specific, at any time, the number of blocks that store lifeless references should be smaller than a constant M and the constant itself should be small enough to ensure that in practice blocks can be updated within seconds or, at most, minutes.

Last but not least, our formulation of the problem entails that the management by multiple processes of the logical structure holding metadata in the block storage must be resilient to failures of these processes, so that the resulting solution can be made as fault-tolerant as the underlying block storage itself. This necessitates distributed algorithms.

4.3.2. Principal Ideas

To address the problem, we analyzed or experimented with multiple potential solutions: from database-oriented or file-system-oriented data structures and algorithms for write-once or erase-before-write storage drives to various fault-tolerant distributed indexes [34, 115, 191, 199, 339]. In short, the fact that the blocks in the assumed underlying storage are immutable and organized into DAGs limits the applicability of techniques that employ in-place updates, notably classic B-tree or many of its modern variations [115]. In our settings, these techniques would be inefficient because emulating each in-place update would require rewriting not only the updated block but also its every ancestor in the DAG. A particularly promising data structure for immutable-block storage was LSM-tree [199], which is widely adopted in distributed databases and offers an excellent amortized cost of insertions. However, to this end, it requires keeping deleted elements for indefinite periods, which is at odds with the need for prompt garbage-collection of deleted data. According to our preliminary study, keeping just a single deleted block with references can prevent reclaiming multi-gigabyte storage space. This can be very problematic, even if it happens just for few days.

All in all, we were unable to find an existing solution that would fit the assumed model of block storage with deduplication while at the same time being able to maintain the massive amounts of metadata required by object storage. Consequently, we have devised new data structures and algorithms dedicated for the considered scenarios.

More specifically, our solution involves two persistent data structures dubbed ObjectMetadataLog (OML) and ObjectMetadataTree (OMT). From the systems perspective, they are used to store object metadata and live references to object data in a write-optimized fashion: all metadata updates are first appended to an OML and only asynchronously (in the background) applied in batches to an OMT, which decreases the write latency and improves the throughput while at the same time ensuring efficient indexing by key prefixes. Both structures are kept in the deduplicated write-optimized fault-tolerant shared immutable-block storage, and their parts are also cached in memory. There is one instance of OMT in the storage and as many instances of OML as there are processes implementing the object storage interface. For scalability, this number of processes can be dynamically controlled to ensure, among others, an appropriate collective throughput and failure resilience. For presentation purposes, however, let us assume for a while that there is only one such process. We will drop

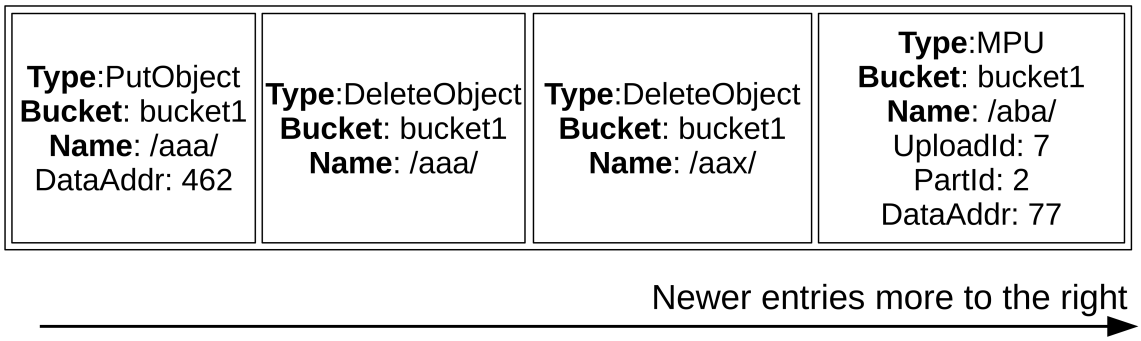


Figure 4.5: Sample contents of an OML.

this assumption shortly.

4.3.3. Object Metadata Log (OML)

The OML contains a sequence of *yet unapplied* metadata updates ordered by their time of arrival (cf. Fig. 4.5). In particular, it also keeps object removals because, for instance, combinations PUT-then-DELETE and DELETE-then-PUT differ in their outcome. A new operation modifying metadata is appended to the sequence, and the client is notified as soon as the append completes. To further improve the throughput of metadata writes, a burst of operations can be batched into a single write request to the block storage. Moreover, the sequence is kept small enough so as to be stored not only in the block storage but also in a memory buffer.

When the in-memory buffer is filled, it is swapped with a new buffer, and the operations it contains are applied to the OMT in the background. The application procedure has to be sufficiently fast so that the memory available for the buffers is not exceeded in the meantime. If that happened, processing client commands via the object storage interface would have to be paused, heavily impairing the overall write throughput. We will address this issue shortly.

When the operations from an in-memory buffer have been applied to the OMT, the buffer is ready to be reused, and the OML blocks corresponding to its contents are also deleted from the block storage (i.e., marked for garbage collection). Apart from potential memory constraints, this is another reason for keeping OML in-memory buffers small: in the block storage, the contents of such a buffer may include references to blocks that may be suitable for garbage collection (e.g., root blocks of deleted objects) and hence after an application to the OMT, they should be deleted as quickly as possible to reclaim storage space.

The careful reader may have noticed that the OML is never read from the block storage during regular operation described hitherto, as the in-memory buffers are sufficient. However, keeping the OML also in the block storage is necessary for fault tolerance. If a process running the object storage interface fails (e.g., its host machine crashes), its reincarnation (or another process) can continue after recovery without losing any updates to the metadata.

4.3.4. Object Metadata Tree (OMT)

The OMT complements the OML by organizing the object metadata in the block storage to enable efficient access, notably looking up and listing by key prefixes. Unlike the OML, the OMT is meant to be very large, as it keeps most metadata of the system—possibly for hundreds of millions of objects. The OMT resembles a B^+-*tree* and keeps a few types of

metadata utilized in object storage in its leaves. The most important ones are metadata of individual objects, MPU parts, and buckets (cf. Fig. 4.6). They are sorted by their type, key, and other content. Apart from storing the metadata, OMT leaves also have references to the roots of the logical block trees with actual object data.

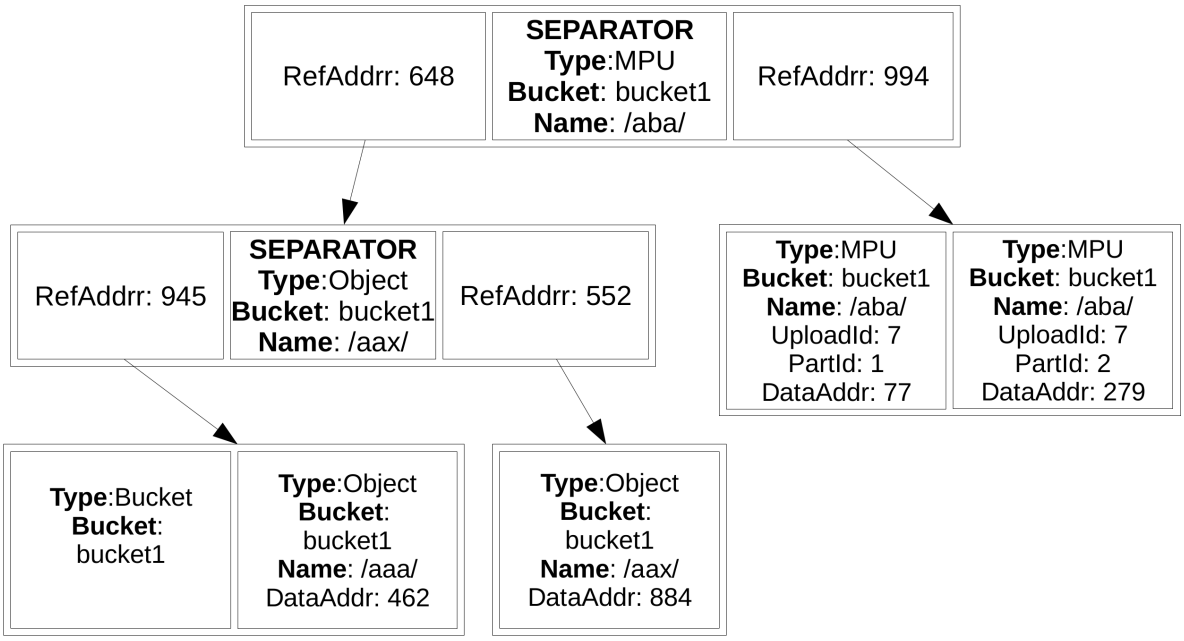


Figure 4.6: An example of an OMT.

The blocks that store internal nodes of the tree have references to the blocks with next-level nodes and separators that facilitate tree traversing. Each tree node, except for the rightmost ones, has its size between $\frac{1}{2}S$ and S elements, where S is a configuration parameter. To keep the tree balanced, all paths from the root node to the leaves, apart from the rightmost one, have the same length. OMT merges are the only operation that modifies the OMT and they keep both aforementioned invariants. All OMT operations are listed in Table 4.1.

Table 4.1: List of OMT operations.

	Description
OMT Merge	Applies changes from an OML to the OMT (described in Section 4.3.5)
OMT Distributed Merge	Applies changes from an OML to the OMT in a distributed manner and consists of two phases: SubOMT Generation (Section 4.3.7) and OMT Combining (Section 4.3.7).
OMT Lookup	Searches for an object in the OMT from its root node to a leaf (B-tree search).
OMT Prefetch	A special lookup version optimized for OMT merges (Section 4.3.6).

In contrast to the OML, the OMT is too large to fit in memory. However, a subset of its nodes is cached to improve performance. In particular, as our preliminary study shows, the list operations limit the number of returned objects, so caching internal nodes can significantly

accelerate consecutive listings. The size of such a cache is meant to be small: asymptotically proportional to the maximal size in the in-memory buffer of the OML.

4.3.5. Metadata Merge

Metadata merge is a background operation of applying to the OMT all changes from the OML. Since blocks holding the tree nodes are immutable, conceptually, the operation has to generate a new tree. However, rewriting all OMT nodes each time an in-memory buffer of the OML fills up would be an overkill. In particular, it would entail rewriting the metadata of every object and bucket in the system. Therefore, instead, the nodes from the old tree are reused whenever possible. In contrast, the blocks containing the overwritten nodes are eventually not reachable from any live blocks, and can thus be garbage-collected.

More specifically, a merge traverses the OMT in a depth-first search (DFS) manner. For each node, a decision is made if either the whole subtree of the node can be reused in the new version of the tree or some update exists in the OML that has to be applied in the subtree. If such an update exists, the subtree is traversed recursively down to locate the relevant leaf node. If, after the update, leaf size is not in $[\frac{1}{2}S, S]$, the node is split into two or the node next to the right is read so as to combine the two nodes into one or more nodes of valid sizes. In any case, all internal nodes on the path from the rewritten leaf to the root require rewriting as well, because the block addresses of the nodes deeper in the tree have changed. For a rewritten internal node with an invalid size, splitting or combination is done as for the leaf. If the size of the root node exceeds S, the node is split, and a new root node above is added: the tree grows by one level.

Special care is given to the MPU delete operation, whose single entry in the OML affects up to 10,000 parts of the deleted MPU in the OMT, and hence could potentially be costly. In such a case, only the two leaves containing the start and the end of the MPU range (and their OMT ancestors) need rewriting, while the nodes in between become suitable for space reclamation. This is because, after the OMT rewrite, their blocks are no longer referenced, directly or indirectly, from any live blocks, and hence will be garbage-collected eventually. In this way, instead of up to 10,000 nodes, the MPU delete affects only a number of nodes proportional to the height of the OMT.

4.3.6. Metadata Merge Prefetch

As mentioned previously, the pace at which the metadata merge operation can be done is crucial for the entire system's performance. A metadata merge that iterated through the OMT and issued a new read to the block storage each time a node intersected with an entry from the OML would last far too long and hence could lead to the aforementioned OML in-memory buffer exhaustion. This, in turn, would require pausing user operations, thereby severely deteriorating the overall backup throughput.

As a remedy, we thus propose a prefetch algorithm, referred to as *OMT Prefetch*, that reads from the block storage at most $2h(b-m)+3h\cdot m = h(2b+m)$ OMT nodes, where h is the OMT height, b is the length of the OML in-memory buffer (in entries), and m is the number of MPU delete entries in the buffer. The algorithm can be run in parallel for all entries of the OML in-memory buffer, so at most h sequential steps (i.e., causally-dependent reads) are required to prefetch all nodes. In other words, the work and span of the algorithm are respectively $h(2b+m)$ and h.

The prefetch distinguishes three types of OML operations: metadata inserts/updates, object deletes, and MPU deletes. For each type, a different set of nodes is prefetched. First,

a metadata insert or update can overfill a leaf node and may thus force splitting it and possibly its ancestors. Splitting a node does not require reading any other nodes, so for an insert/update operation in the OML, only those nodes are prefetched whose key ranges include the inserted/updated key. A metadata delete can in turn lead to combining or combining-and-splitting a leaf node, and possibly its ancestors, with the first succeeding nodes at the same levels. Therefore, only the nodes whose key ranges include the deleted key and their first right siblings are prefetched.[1] Finally, as explained previously, an MPU delete can lead to removing multiple nodes and updating no more than three nodes at each level. It is thus enough to prefetch the nodes that intersect with the keys representing the start and the end of the MPU parts, together with their ancestors, and the first right siblings of the nodes on the MPU end path (all of the nodes in between will be deleted).

4.3.7. Distributing Metadata Merge

The solution described hitherto assumes only a single process operating on the OML and OMT. However, the number of such processes must be scalable to handle more load and tolerate failures. A straightforward approach would be to partition the buckets among the multiple processes so each process would operate on a disjoint set of buckets and objects they contain. However, this may lead to overloading the processes responsible for popular buckets.[2]

We propose a solution in which, rather than only buckets, also individual objects are partitioned among the processes. Metadata operations for a given object are directed to the corresponding process. Each process appends updates to its objects and buckets into its private OML. However, the OMT is shared by all processes, which requires distributing the previously described metadata merge operation. Such a distributed merge proceeds in two phases (see Fig. 4.7). First, many disjoint OMTs, called SubOMTs, are generated by individual processes. Second, all these subtrees are combined into a single OMT, using a parallel algorithm.

SubOMT Generation Phase

To distribute work evenly, the space of OMT keys is divided into ranges, and each range is assigned to a different process. To achieve this, keys are first partitioned among all processes (e.g., based on their strong hashes), so that each OML is expected to have a similar size. Before each merge, a single, dynamically chosen process calculates boundaries for the ranges based on the contents of its OML. Then, the ranges are broadcast to the other processes, so that each process can generate its own SubOMT by merging relevant entries from *all* OMLs with the subset of OMT nodes that are within its assigned range. In this phase, each process reads all OMLs, but this happens simultaneously and, per previous explanations, the OMLs are small, and hence the block storage caches can be effectively used.

OMT Combining Phase

After the first phase, each of the resulting subtrees covers a disjoint contiguous key range. In the second phase, the subtrees are combined into the new global OMT with all keys. This

[1] With a small exception: the sibling of the leaf, which is not prefetched even though it may be needed to create a new leaf of proper size. The reason is that such a read can be done on demand later without affecting the critical path and the tree iteration, and we wanted the algorithm never to read more leaves than necessary.

[2] As a side note, in theory, rather than a bucket, a particular object could be popular and receive an excessive load. Our solution does not aim to address this simply because, in practice, we have not observed this phenomenon to be relevant to the backup use case.

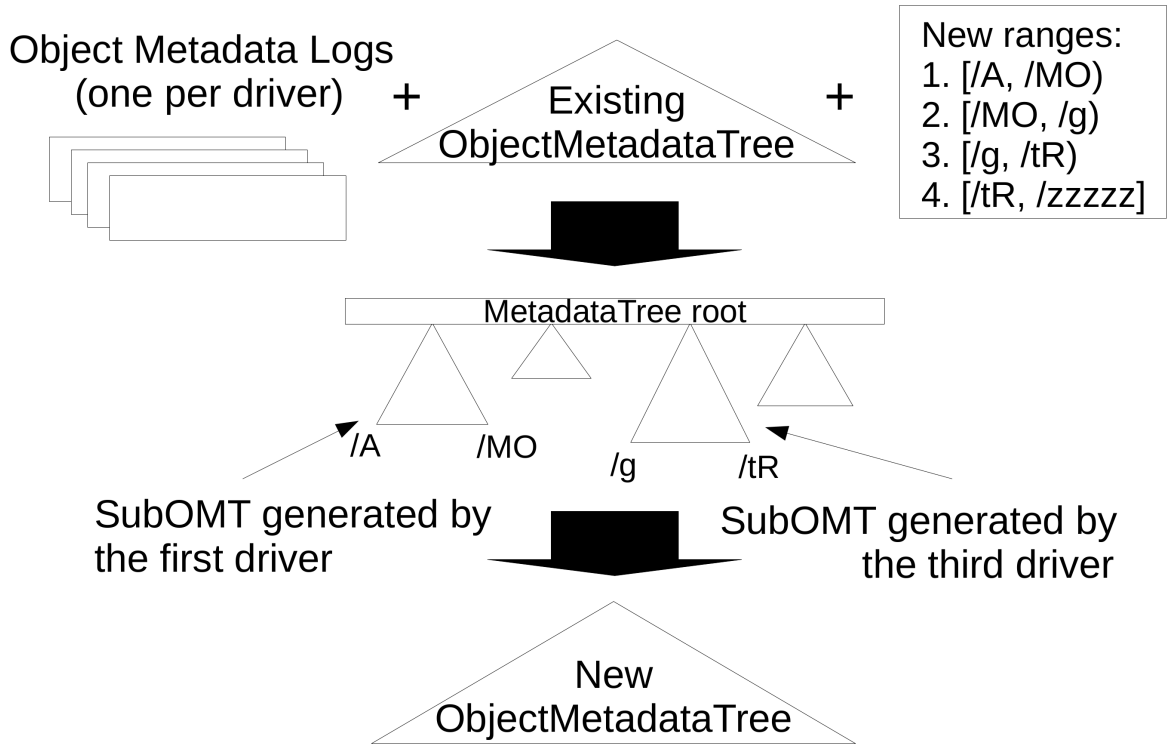


Figure 4.7: The phases of the distributed metadata merge.

process is not trivial, because the nodes on the rightmost path of each subtree can have their sizes below $\frac{1}{2}S$ and the heights of the subtrees may vary.

For two subtrees, we present how to generate a valid OMT by reading only the rightmost path of the first subtree and the leftmost path of the second (also see Fig. 4.8). First, read the rightmost path of the first subtree. According to the invariant, all other nodes in the subtree have correct sizes. Likewise, read the leftmost path of the second subtree. Starting from the leaf node in the leftmost path of the second subtree, add each entry from the node to the corresponding node from the rightmost path of the first subtree. If the size of the node is exceeded, create a new node and add a reference to it in a higher-level node; if the size of that node is exceeded too, repeat the process. The key observation is that if there are two nodes and at least one of them (the one from the second tree) has a valid size then one or two valid nodes can be created in a way that there are no leftovers. Ultimately, the only nodes with their sizes less than $\frac{1}{2}S$ are: the node that contains the entries from the root of the second subtree, the nodes on the rightmost path of the second subtree (it was spliced), and, if the first subtree was higher, the upper nodes on the rightmost path of the first subtree. Altogether, the combining reads only $h_1 + h_2$ nodes, where h_1 and h_2 are the subtree heights. The tree height is tiny, as discussed shortly, and the appropriate paths in different SubOMTs can be read in parallel from the block storage, so even for huge numbers of keys, one process is sufficient to combine thousands of SubOMTs quickly.

Remarks on Object Key Space Partitions

The presented solution spreads the load due to handling metadata between all processes, for instance, based on hashes of keys. Such an approach is efficient for those object storage interface commands that affect a single key (e.g., GET, PUT, DELETE), because one process

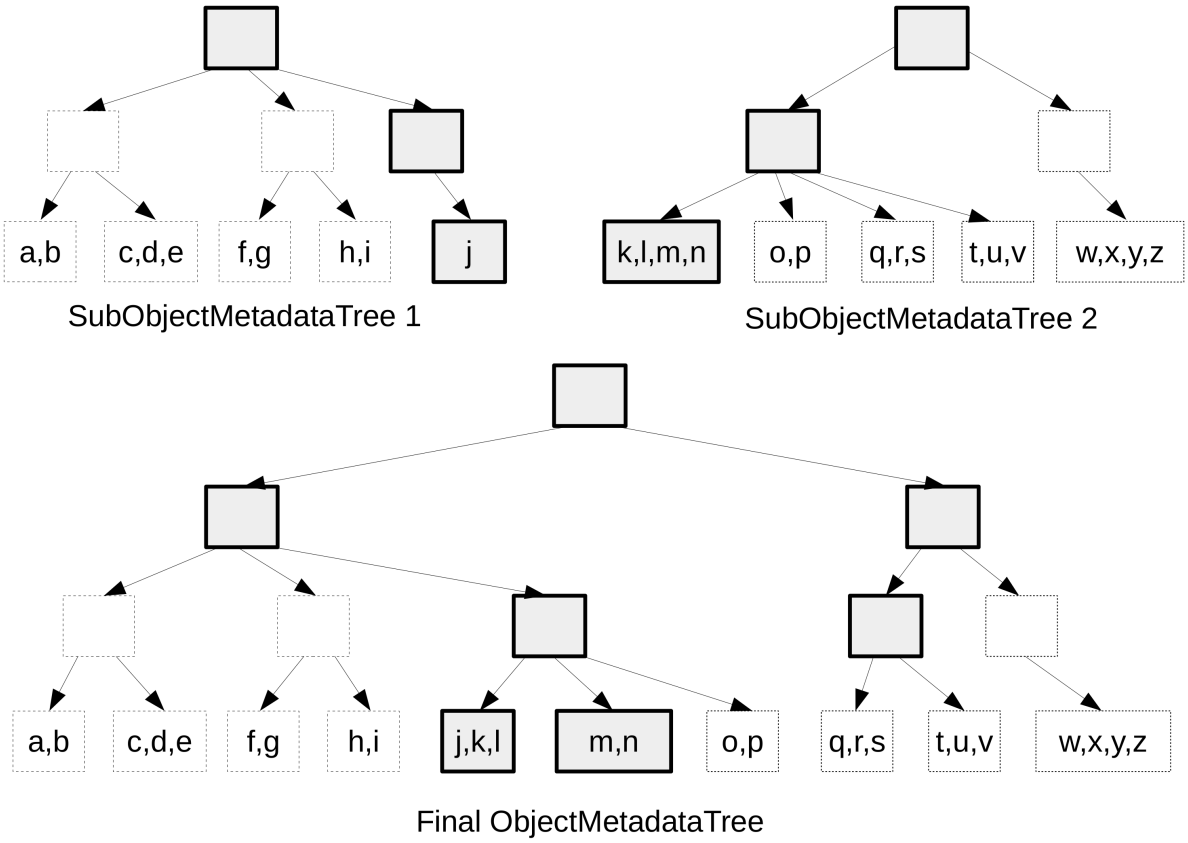


Figure 4.8: Combining two SubObjectMetadataTrees with S = 4. Only the gray nodes are read and written. The letters denote object keys.

can handle a given command invocation entirely. Interestingly, an MPU command invocation can also be handled by one process because the number of parts in such a request is limited (e.g., to 10,000). However, an object interface does contain commands that read information about many keys at once, like object listing. Efficient and consistent handling of such collective operations requires special attention.

Since the OMT is global and maintains the order of keys, retrieving from the OMT metadata, for example, for object listing, requires just a handful of reads, and hence they can be performed by one or multiple processes. Object storage interfaces limit the size of an object listing (e.g., to 1000 objects) and, in general, of the output of similar collective operations. In effect, many consecutive requests must be sent to generate a longer output. Nevertheless, such long outputs are also handled efficiently, as the block storage can cache the blocks corresponding to the repeating node paths in the OMT. The issue, however, is that the freshest metadata are not stored in the OMT but in the OMLs. Therefore, during a listing or a similar collective operation, the metadata from the OMT must be updated with the metadata from the OMLs of relevant processes. If the metadata are distributed by key hashes, virtually every OML must be contacted, which entails flooding all processes with requests.

To avoid such flooding, we propose to dynamically partition keys among processes. More specifically, the space of keys can be divided into ranges that are dynamically calculated based on the current load. If there are few or no requests, each process can be responsible for a similar number of keys. It reports its load to a distinguished process, so that if one process receives more requests than it can handle, the ranges can be recalculated. If necessary, multiple

processes can handle the same range, and in such a case, the requests within that range can be further distributed to selected processes based on the key hashes. With this approach, collective operations read metadata handled by multiple processes only when necessary, and only the relevant subset of the OMLs is affected.

Failover Handling

In the case of a process failure, another existing or new process can take over. This requires only restoring in-memory data from the block storage: the in-memory buffer of the OML of the failed process and the cache containing the OMT nodes with the operations from the buffer applied. If the failover is handled by some existing process instead of a new or recovered one, the process also needs to take over the metadata merge responsibilities for the failed process, so that an ongoing merge can be completed. In effect, it has additional keys to handle during the merge. This temporal imbalance is naturally corrected by a new key partitioning, which will take into account the change and distribute the work more evenly.

4.3.8. Final Remarks

The proposed solution can be used even in very large systems. The height of the OMT is limited by $h = \lceil log_s(n) \rceil$, where s is a branching degree (limited by constant S) and n is the total number of objects and buckets. Even for a massive 20,000-machine backup appliance, with each machine having 12 high-end 14-TB HDDs, an average object size of 10 MB, 20:1 deduplication ratio, and 5:1 compression, there can be $n =\sim 3.36*10^{13}$ objects. With $S = 64$, which is reasonable to keep the OMT nodes small when keys are large (i.e., 1 KB), the tree has at most 9 levels.

Further calculations confirm that keeping metadata in block storage rather than, for instance, in memory or on dedicated local SSDs of machines hosting the processes implementing the object storage interface is not only a design decision but actually a necessity in large systems. For example, in the previous system, each machine needs to store metadata of $1.68*10^9$ objects, so with 1 KB keys, they take 1.7 TB. If the objects are smaller (e.g., 1 MB) and have additional 2-KB user-defined metadata, the required capacity per machine sums up to over 50 TB. Assuming a typical hardware architecture of backup appliances, storing such a volume of metadata in RAM or on SSDs is infeasible today. Even if there are SSD disks in such appliances (which is not always the case), their capacity can already be used for other purposes. In other words, our algorithmic assumption is reinforced by the limits of today's technology.

4.4. Implementation

We have implemented ObjDedup in the aforementioned HYDRAstor system [94]. At the time of writing this dissertation, it was part of the product, delivering the object storage interface in the same way as the classic backup interfaces.

4.4.1. Overall Architecture

From a systems perspective, HYDRAstor consists of storage servers, which keep data on their disks, and a layer of access servers, which provide external access (cf. Fig. 4.9). The number of storage and access servers can vary depending on the capacity and performance targets, and the system can scale from one server to multi-rack installations.

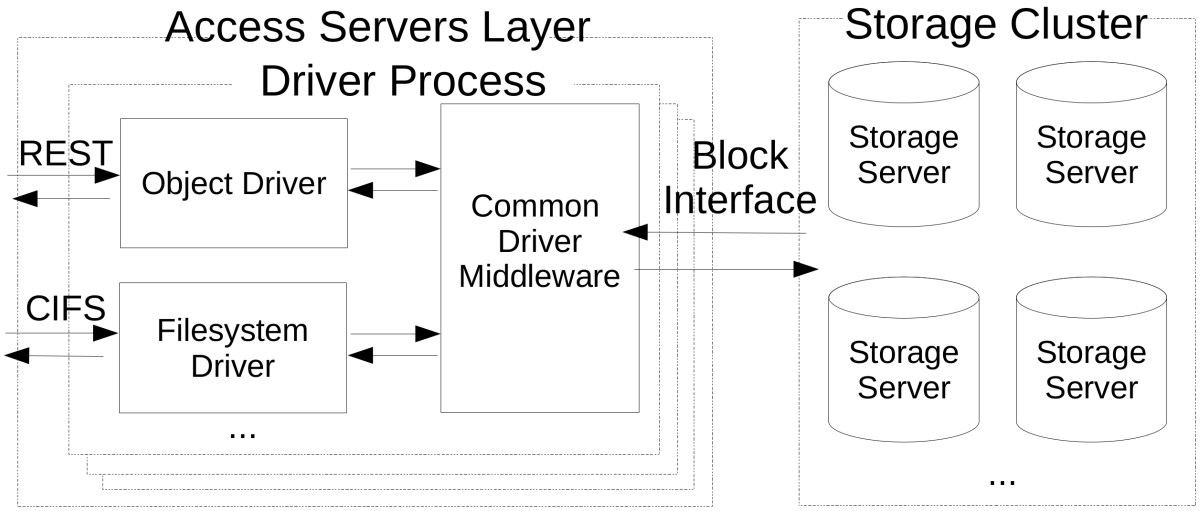


Figure 4.9: A high-level architecture of access and storage servers. One or multiple instances of driver processes communicate through a block interface with the storage cluster.

Storage servers maintain data by means of the block-level engine. In particular, they manage the distribution of blocks among storage media and memory caches, perform operations on the blocks, handle hardware failures, coordinate garbage collection, and the like. Storage servers comprise a storage cluster that exports a coherent block-level interface to access servers, with operations like writing a block or querying if a block is stored given (a hash of) its content.

Access servers, in turn, provide higher-level interfaces, like CIFS or, in our case, REST, on top of this block-level engine. These interfaces are implemented as drivers. Internally, they use common middleware that facilitates reusing functionality for accessing the block-level engine of the storage cluster. Especially the deduplication pipeline, including chunking and fingerprinting, is implemented in the middleware and coordinated by the access servers. Likewise, services for distributing computations, including optimized message routing (conceptually similar to MPI [158]) and locating servers, are provided by that middleware.

Overall, this architecture matches what we assumed previously for our algorithms. In this view, our work concerns the *object driver*, which implements the algorithms for OML and OMT to provide an object storage interface on top of the common driver middleware (cf. Fig. 4.9). Depending on the scale of the system as well as client-specific performance and fault tolerance requirements, instances of the object driver are hosted by one or more access servers.

4.4.2. Object Driver Architecture

The most outer layer of the object driver is HTTPServer, which receives REST commands (see Fig. 4.10). They are then processed by RequestHandler, which implements the logic of handling both data and metadata of objects and buckets, which ultimately end up in the storage cluster. For data-related operations, RequestHandler essentially uses the aforementioned common driver middleware, as data can be handled similarly to the other interfaces. For metadata-related operations, it also collaborates with ObjectMetadataLogHandler and InMemoryObjectMetadataStore. ObjectMetadataLogHandler maintains the in-memory buffer of the OML corresponding to the instance and is responsible for coordinating metadata merge

operations. InMemoryObjectMetadataStore manages the cache of the OMT with the updates from the in-memory buffer of the OML applied.

Again, this architecture is coherent with our algorithmic assumptions. HYDRAstor is write-optimized, and thus the object driver must follow the same principle. Writing data is inherently optimized by the storage cluster, and the control flow in the driver does not add any extra steps. For handling metadata, in turn, the object driver simply employs ObjDedup, which is also write-optimized by design.

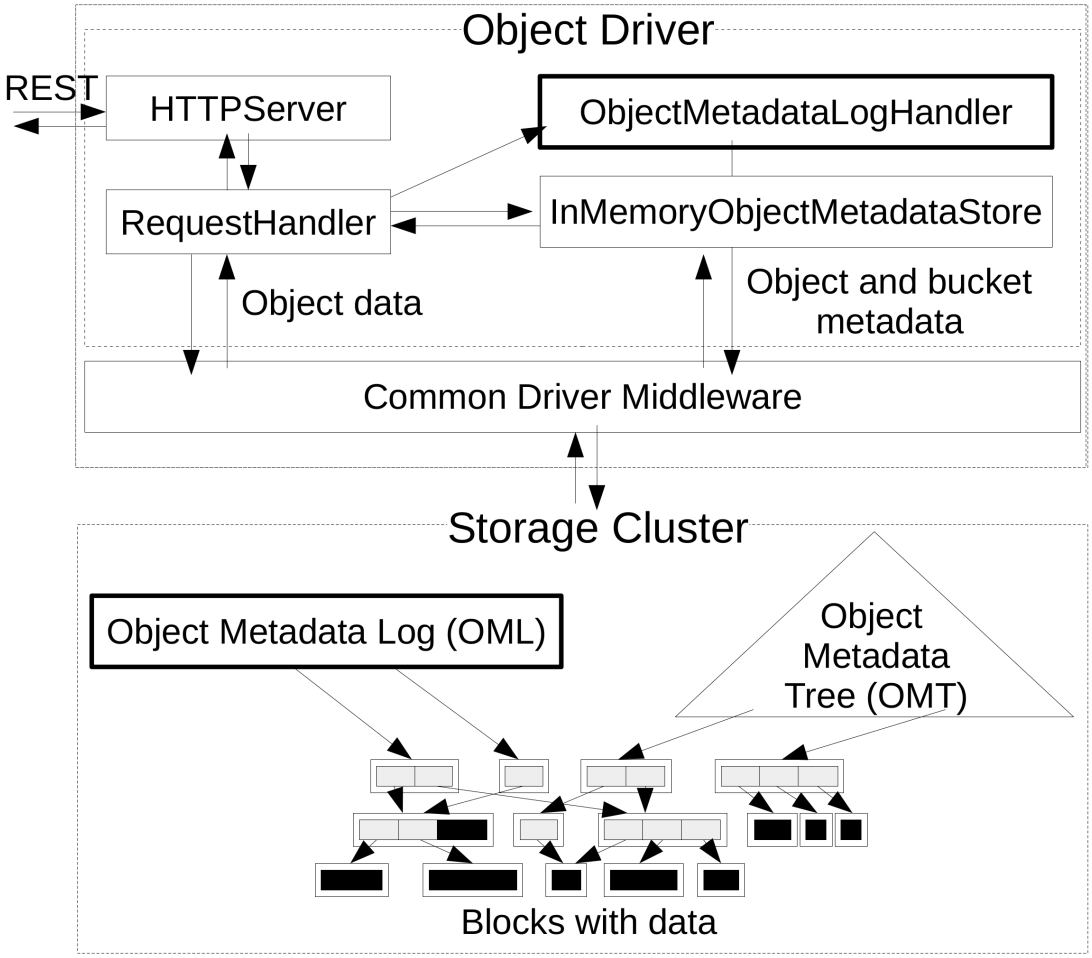


Figure 4.10: The architecture of the object driver.

4.4.3. Additional Issues

To reduce the latency of selected read and write operations, especially on small files or during more interactive sessions with the backup appliance, HYDRAstor features priority requests. They have a higher preference in queues, and their outcomes are reflected in storage media faster. However, they must not be abused, because their performance gains would not be observable in such a case. Following this approach, our object driver uses priority requests only for appends to the OML to further improve the client-perceived latency of metadata write operations. In contrast, all other requests to the storage cluster are regular (non-priority) ones. This is possible thanks to the fact that metadata merge is done in the background and utilizes OMT Prefetch, which anticipates and parallelizes future reads, thereby making

metadata merge fast even without priority requests. In the same way, OMT Prefetch improves the performance of collective operations, like object listing by the key prefix.

Such efficiency is also important for space reclamation. As a basis for garbage-collecting dead blocks, HYDRAstor utilizes reference counting. The space reclamation process is done in the background, in parallel to normal requests. Because of deduplication, parallel requests can increase or decrease block reference counts while seemingly dead blocks are being garbage-collected. The space reclamation algorithm must remain correct in the face of such concurrency, which is not trivial [283]. In particular, the algorithm operates in epochs and imposes restrictions on driver-kept block addresses. Notably, in epoch T, a driver must not keep addresses obtained before epoch $T - 1$. From the perspective of the object driver, if the deletion operation for an entity (e.g., an object, bucket, or MPU) is not applied from the OML to the OMT and deleted from the OML, a reference to the entity is live and garbage collection does not remove the entity. In practice, this means that the object driver should be able to apply changes from the OML to the OMT within a few minutes. This reinforces our previous claims about the need for the efficiency of metadata merge in ObjDedup.

Another side issue is the use of a dynamically selected, distinguished object driver instance in cases when multiple such instances operate in parallel. Essentially, this problem entails leader election. As HYDRAstor already implements leader election and automatic failover, we were able to reuse this functionality. In practice, however, any sensible at-most-one leader election algorithm could be used instead [276].

Finally, object storage interfaces are sizeable and change over time. One cannot simply ignore some of their functionality, because its use by backup applications evolves as well. Therefore, while devising the algorithmic solutions was already challenging, implementing them as the production-ready object driver without altering the block-level engine was equally demanding. In effect, however, as the requirements of ObjDedup on this engine are minimal, it can likely be implemented in other systems with interfaces allowing for writing immutable blocks in tree-like structures, which is a common feature in deduplication storage [248].

4.5. Experimental Evaluation

To evaluate our solutions, we have conducted numerous experiments using the implementation of ObjDedup for HYDRAstor. We present the most important results in three groups: Section 4.5.1 contains experiments that evaluate the main performance goals of ObjDedup in a distributed setup; Section 4.5.2 offers a comparison with the state of the art; and Section 4.5.3 encompasses a detailed evaluation of OMT performance in various scenarios, which aim to highlight possible limitations and bottlenecks.

The presented experiments emphasize write-related workloads. This is because, first, they are the most critical for a backup appliance and, second, other operations (e.g., object deletes or MPUs) incur largely similar or lower overheads on the performance of the system.

Most of the experiments were conducted on a testbed composed of 12 servers, which, for the following reasons, was sufficient to show how our solutions behave at scale. First, the data were kept in a default 9+3 erasure-code scheme. In effect, with 12 servers or more, each machine stored only one fragment of erasure-coded data, and that would not have changed if the system had grown further. Second, considering the maximal raw capacity of one HYDRAstor server, which is 168 TB, the total capacity of a 12-server system was over 2 PB. In other words, it was a fairly large installation considering deduplication, especially given that, in contrast to HYDRAstor, many popular backup appliances available on the market do not scale more [87, 253]. Finally, the experiments put a considerable pressure on our infrastructure:

altogether they took several weeks. Therefore, even in the scaling tests, we had to limit the maximal size of our testbed to 18 servers. Each of the servers comprised 2x Intel Xeon CPUs E5-2620 v3 2.40GHz, E5-2660 v3 2.60GHz, or E5-2430 2.20GHz, 96 GB of RAM, and 12x 7200-RPM SATA HDDs of 2–6 TB each.

4.5.1. Assessment of the Main Performance Goals

We start by assessing the main performance goals of ObjDedup. To this end, we evaluate its backup throughput, scalability, and overheads.

Backup Throughput

We test ObjDedup with varying object sizes and four different workloads: 100% non-duplicate writes, 100% duplicate writes, 90%:10% duplicate:non-duplicate writes, and reads. The first two workloads are extreme scenarios but still possible in practice (e.g., writing an initial backup without internal duplicates and writing the same backup twice). The 90%:10% duplicates:non-duplicates matches the expected average daily deduplication ratio [26]. Finally, the workload with reads is for reference. Each experiment involves a 2.4-TB data set, which is large enough to get reproducible results.

Figure 4.11 presents the throughput of ObjDedup normalized to the results of the HYDRAstor file system driver in the same configuration, which we use as a baseline. The object/file sizes varied from 8 MB, which is rather small even for object storage backup (we present results for yet smaller files in further experiments), to 1 GB, as increasing object size has a marginal impact on performance from some point. Both drivers achieve a comparable write performance for larger object/file sizes (over 128 MB). Moreover, since ObjDedup reuses the deduplication implementation (e.g., chunking, fingerprinting), it achieves the same deduplication ratios. With smaller object/file sizes, there are differences in write performance in favor of ObjDedup. The read throughput is comparable for all object sizes; the only noticeable difference is for 8 MB objects in favor of ObjDedup.

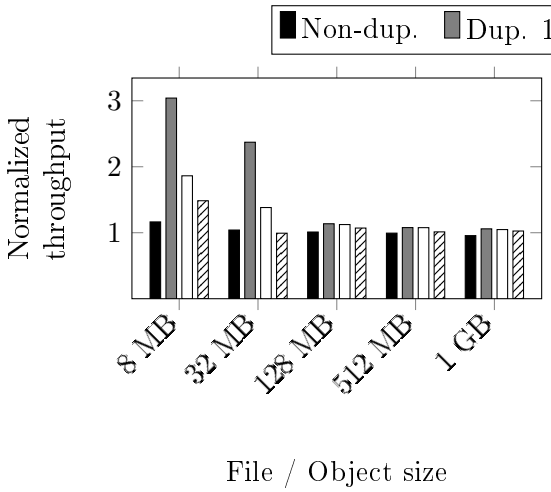


Figure 4.11: ObjDedup throughput normalized to results of the file system.

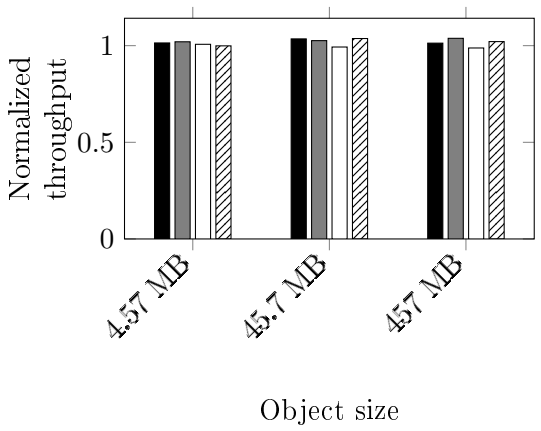


Figure 4.12: Throughput of writing with three different object sizes normalized to throughput of writing with the one (average) size.

Since ObjDedup is expected to handle simultaneous streams from multiple backup applications, the data in each of the streams can come in differently-sized objects. Nevertheless, as Fig. 4.12 shows, the throughput of writing data from three simultaneous streams with different object sizes (2/8/32 MB, 20/80/320 MB, 200/800/3200 MB) does not diverge significantly from the throughput of writing the same streams with a single object size (4.57/45.7/457 MB). The object size was selected as an average object size when three streams of equal size were written with three different object sizes (e.g., when three 1024 MB streams are written in 2/8/32 MB objects, the average object size is 4.57 MB).

Scalability

To demonstrate the scalability of ObjDedup, Fig. 4.13 depicts the throughput of writes in the three different write workloads (100% non-duplicates, 100% duplicates, 90%:10% duplicates to non-duplicates) and with three different object sizes (512 KB, 8 MB, 512 MB) that represents very small, medium and large objects (as presented in Fig. 4.11 increasing object size over 128 MB has a marginal impact on the performance). The presented values are normalized to the per-server throughput of 4-server HYDRAstor setup.

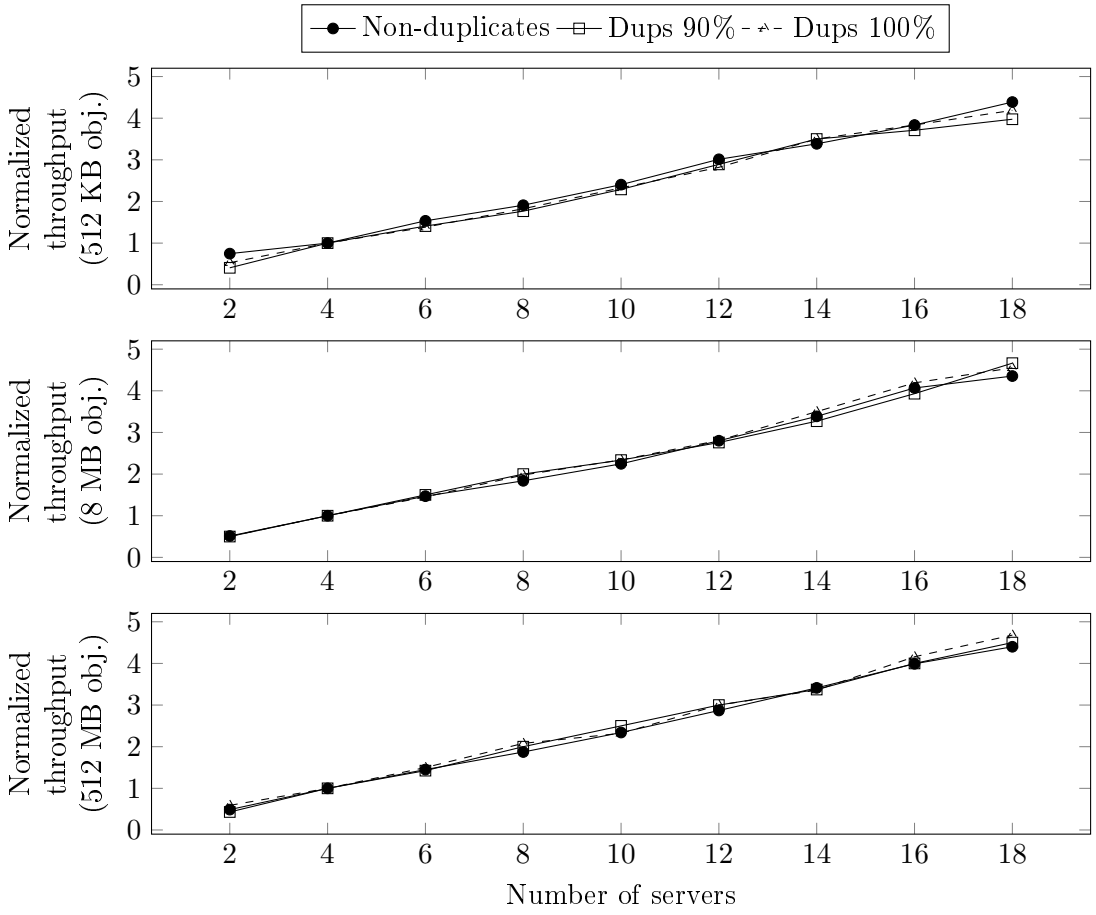


Figure 4.13: Scalability of ObjDedup (normalized to results for 4 servers).

On average, 18 servers were 4.41 times faster than four. The worst result (an average improvement of 4.18) was obtained with the smallest objects (512 KB), which reinforces the claim that efficiently handling small objects is not trivial. The smallest possible distributed

configuration (2 servers) is also included in the plot, but the results are a bit skewed, as such configuration uses far less network communication.

Overheads

We also measured and evaluated the overheads incurred by ObjDedup in terms of resource consumption (memory, storage, CPU), response latency, and scaling.

Memory consumption: The major cost is buffering data incoming through the HTTP server. The buffering is required to ensure that all components of the system have enough work to achieve a high level of parallelism. In our experiments, we use a 2-GB buffer per server, which is sufficiently large. The memory overhead of other data structures is in turn significantly smaller (e.g., metadata of 50k objects in an OML take less than 200 MB, even with 3 KB of metadata per object). Overall, none of our experiments consumed more than 3 GB of RAM per server.

Storage space consumption: Data blocks are referenced, and these references can take up a considerable amount of storage space. However, this happens with any interface, not just ObjDedup. Similarly, objects contain their metadata (up to 3 KB per object), but any object storage system must keep these. Therefore, the most important aspect is quantifying additional overheads in OMT and OML.

For each object, besides its metadata, an OMT leaf keeps 20 bytes of our internal metadata. Second, there are internal OMT blocks (storing object keys of up to 1 KB and separators), but with a branching degree $S = 64$, there are on average 48x fewer of them in the penultimate level of an OMT than leaves, and far fewer on all other levels combined. Finally, an OML keeps metadata of a limited number of objects (typically ∼50k), which is negligible in comparison to the millions of objects kept in the system. What is important, however, is that, during a merge, each object in an OML can cause a rewrite of a whole OMT leaf, and both versions of such a leaf need to be stored until the merge is finished. Therefore, for S=64, an OML can store as many as 3.2 M of object metadata copies. To sum up, with a realistic workload for backup data (e.g., object data being considerably larger than object keys) and multi-terabyte storage servers, the storage capacity overheads incurred by ObjDedup are far below 1% of the system capacity.

CPU consumption: If HTTPS is enabled, the majority of CPU load is due to encryption and decryption, as providing a multi-gigabyte throughput with cryptographic algorithms can require multiple cores. The CPU consumption of ObjDedup itself, in turn, highly depends on the workload. Typically, it does not exceed 3 cores per server, mostly handling HTTP requests and managing the OMT.

Latency: In the expected write-dominant backup workloads, the latency overhead is marginal and mostly comes from the fact that at the end of writing an object, an OML entry must be written in the block storage. However, there are two cases in which ObjDedup increases the latency considerably. First, when reading an object, multiple levels of the OMT are accessed, unless at least some of them are cached. In effect, the time of arrival of the first bytes of the data is proportional to the height of the OMT. For instance, if the OMT has 5 levels and a block read takes 100 ms because of the concurrent load, reading object data will take 500 ms. Second, a metadata merge can cause numerous additional I/Os per object if object keys are non-sequential, which we study in microbenchmarks (Section 4.5.3). In such cases, the storage cluster can become overwhelmed with requests and have several times longer response times, which ultimately increases the latency of all operations.

Scaling: As shown previously, the throughput of ObjDedup scales nearly linearly (as presented in Fig. 4.13), just as bare HYDRAstor. However, this is true as long as meta-

data handling does not become a bottleneck. We study such problematic scenarios in the microbenchmarks (Section 4.5.3).

Conclusions

To sum up, the main performance goals of ObjDedup were met. The achieved throughput of writing and reading backup with ObjDedup is comparable to the results achieved by filesystem driver for large objects/files and even better for smaller objects/files. In general, ObjDedup scales linearly, just as the underlying storage cluster. Finally, the overheads on ObjDedup are limited.

4.5.2. Comparison with Existing Solutions

In this section, we show how ObjDedup compares against the state of the art. As mentioned previously, we are not aware of any prior work that closely matches ours, and hence for comparison, we select systems that are close to our solutions in as many relevant aspects as possible. In Section 4.5.2, we thus compare ObjDedup to two state-of-the-art open-source object stores. In Section 4.5.2, in turn, we compare it to three file systems with deduplication.

Furthermore, performing the comparison was still challenging because of significant differences between ObjDedup and the reference systems. To avoid bias in favor of our solutions, the experiments evaluated ObjDedup in an unfavorable setup: in the first one, ObjDedup was the only object store that performed deduplication during non-duplicate writing, and in the second, the configuration was far from what ObjDedup was designed for.

Backup Throughput

In the first experiment, we used COSBench [345] to compare ObjDedup with other object stores. COSBench is a framework that enables performance testing of object storage but does not support writing duplicates. Therefore, we modified its code for that purpose. As the two reference systems, we selected Ceph with RadosGW and MinIO, which are state-of-the-art open-source object stores. We chose them despite their lack of in-line deduplication, because we did not find any alternative offering this feature that could be used in the experiments or had comparable results published.

In the experiment, we used one of our servers (2x Intel Xeon CPU E5-2620 v3 2.40GHz and 12x 4 TB SATA HDD). We decided upon the single-server setting, as it significantly simplified the setup and result analysis. Each of the three systems used 9+3 erasure codes, which give a high failure resilience with 12 disks and are broadly adopted (e.g., they are proposed in Ceph's documentation [66]). To store data internally, we used XFS for MinIO and BlueStore for RadosGW, which are recommended to achieve a high performance. The object sizes in the presented experiments vary from 8 KB to 2 GB, as decreasing them even further did not affect the number of operations per second and increasing them did not affect the throughput.

For non-duplicates, ObjDedup achieves a similar throughput as the others (Fig. 4.14). Despite the fact that deduplication consumes additional resources when writing non-duplicates, ObjDedup is either the fastest or the second fastest system. With duplicates, in turn, it has a 1.8–3.83x higher throughput than the others (up to 2790 MB/s), which is expected, because for duplicate writes in-line deduplication can overcome the limits of HDDs. Finally, in COSBench read tests (not plotted), the performance of ObjDedup is also comparable to MinIO and RadosGW. With small objects, it exceeds 740 GET/s. Its throughput with 32MB or larger objects is in turn close to 800 MB/s.

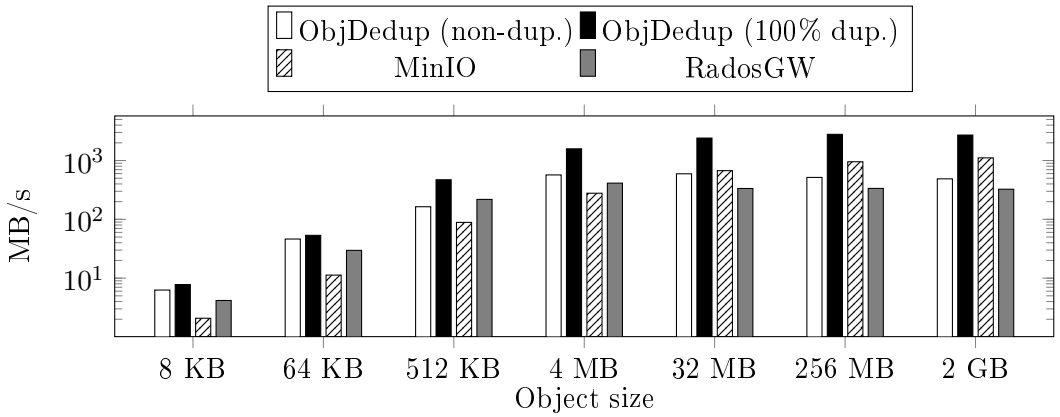


Figure 4.14: Write throughput (log. y-axis) with varying object sizes.

Files/Objects per Second

In the second experiment, we compared ObjDedup with existing deduplication solutions. Therefore, we selected three commercial file systems with in-line deduplication. We do not disclose their vendors, as our goal is to validate ObjDedup and not to compare the products.

Some of the file systems were designed for a mix of HDDs and faster storage devices (e.g., for a write-back journal). Therefore, to find the common denominator and make it possible to compare the solutions, we employed two testbeds: the first using SATA 1 TB 7200 RPM HDDs and the second using NVMe 1 TB SSDs for all kinds of storage. Both configurations were very different from what ObjDedup was tuned for, especially the full-SSD one, but facilitate result reproduction. In both of them, a single machine with Intel Core i7-7820X @ 3.60GHz and 64 GB RAM was used, as some of the file systems do not scale to more machines.

The presented experiments were conducted using MinIO's client, which can copy data to both file systems and object stores. Nevertheless, experiments with other tools gave similar results. We also conducted experiments with MinIO configured as a layer on top of each file system to provide object storage with in-line deduplication.

Initially, we intended to show results for objects and files of various sizes, similarly to the previous experiments. However, for objects below 100 MB, ObjDedup was typically one or two orders of magnitude faster, so we decided to investigate the phenomenon even further. Therefore, we limited the contents and file names to 32 bytes, despite the fact that ObjDedup was not designed to handle data in such small objects efficiently. In that way, we were able to measure the upper bound for operations (file creates or object PUTs) of each solution. The same directory was copied twice, so in the second run, all data were duplicates.

As shown in Fig. 4.15, ObjDedup can handle 4.6–8.35x more operations than the fastest of the file systems. In general, file systems with in-line deduplication are complex, so handling so many files per second is challenging for them. In contrast, ObjDedup applies updates to the OMT in batches, so they are highly efficient. Additionally, MinIO's client copies files to a temporary location and renames them afterward to prevent listings on inconsistent files, so two operations are needed per file. Compared with MinIO on top of a file system solution (FS+MinIO in Fig. 4.15), ObjDedup can handle 5.26–11.34x more PUT/s, which is justified as an additional layer introduces new overheads. To sum up, even with such a minimal object size, none of the file systems reaches a request rate comparable to ObjDedup.

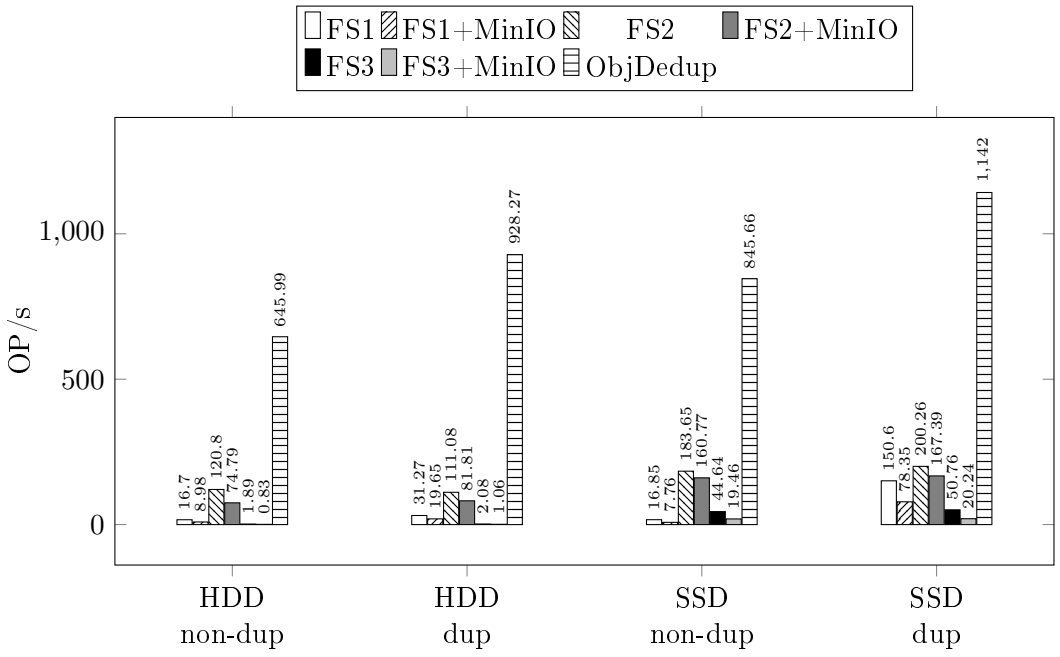


Figure 4.15: Number of operations (file copy / object put) per second.

Conclusions

ObjDedup is the first system of its kind and therefore comparison with existing solutions was challenging but we managed to conduct experiments with systems that are somehow similar. Compared to object storage implementations on top of file systems provided by state-of-the-art deduplication solutions, our solution can handle significantly more requests per second (5.26–11.34x). Moreover, ObjDedup offers a much higher throughput when writing duplicate data (1.8–3.93x) compared to leading object stores without in-line deduplication.

4.5.3. Microbenchmarks

OMT is our novel data structure essential for high performance. If a metadata merge takes too long, new requests cannot be handled, and the system throughput is decreased. For instance, if an OML stores up to 50k entries and a merge takes 100 seconds, the system cannot handle more than 500 PUT/s. Therefore, we evaluate the metadata merge and the distributed metadata merge in a series of experiments that show their performance under various circumstances, especially, how the pattern of the workload affects the performance of a merge.

The experiments were on-purpose conducted in a rather small configuration to emphasize the impact of the pattern and not system scaling. More specifically, the configuration involved two servers with 12x SATA 7200 RPM 6 TB HDDs each and 2x Intel Xeon CPU E5620 @ 2.40GHz or 2x Intel Xeon CPU E5620 @ 2.40GHz. The first server hosted both an object driver and a storage cluster server, and the second—just a storage server. Our testbed thus consisted of more than one server but only one object driver conducted merges (merge scalability is evaluated in further experiments).

To discuss the experiments, let us explain the object key patterns they utilized. In the *rand* pattern, all keys consisted of generated UUIDs, so object metadata were uniformly distributed across a large number of OMT leaves. In the *seq* pattern, objects were written to

1000 different prefixes (50 characters of a prefix followed by subsequent integer numbers), so consecutive objects belonged to the same or neighboring leaves. *Seq* thus approximated what we would expect from backup applications, while *rand* modeled a theoretical worst case. In addition, in some experiments—those with suffix *-delay*—block reads in the storage cluster were delayed by 150 ms to simulate a system overloaded with other tasks (e.g., due to drivers other than ObjDedup). Finally, unless stated otherwise, up to 50k entries were stored in an OML before a merge was initiated.

Prefetch Algorithms

First, we evaluate the impact of our metadata merge prefetching (OMT Prefetch). Without any prefetching, waiting for consecutive reads increases the merge time to tens of minutes, even when an OMT contains less than a million objects (Fig. 4.16). Therefore, prefetching is simply a necessity.

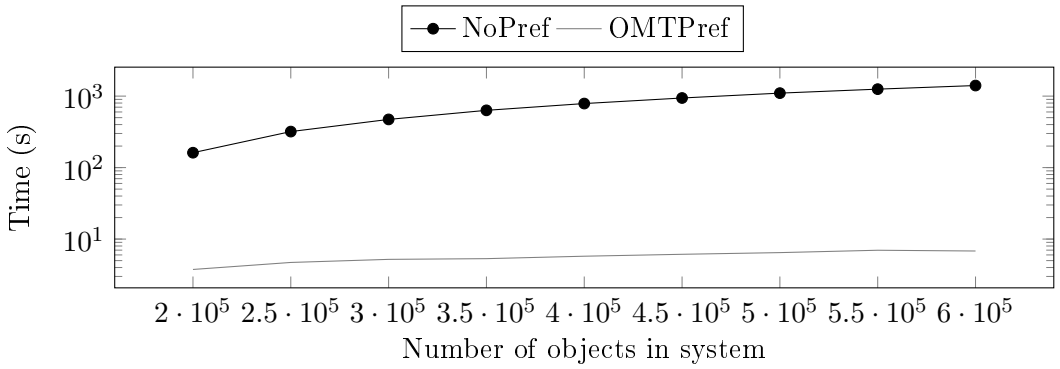


Figure 4.16: OMT merge with data written in *rand* pattern.

However, with a naive approach, referred to as the AllInt Prefetch, which simply prefetches all internal nodes of the OMT, the number of read nodes increases linearly in the total number of servers, even in the *seq* pattern (Fig. 4.17). In contrast, with the OMT Prefetch, it stabilizes around 10 seconds.

Since merges without the OMT Prefetch are slow, all experiments in the subsequent sections utilize it.

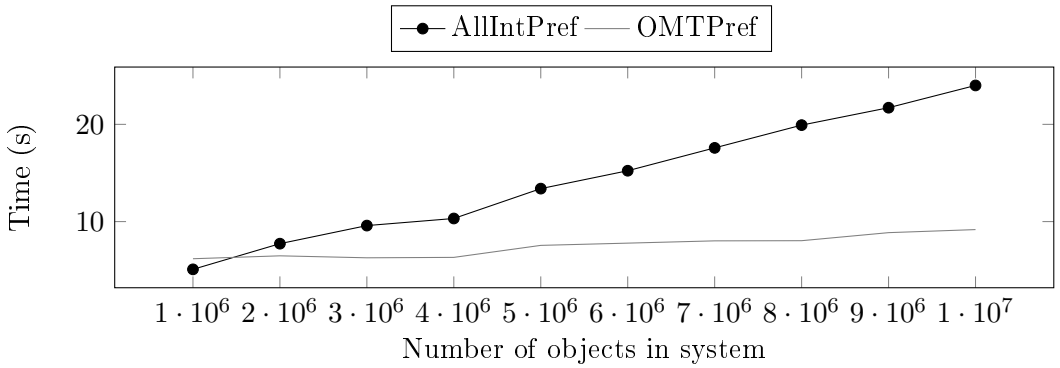


Figure 4.17: OMT merge with data written in *seq-delay* pattern.

Key and Metadata Sizes

As described in Section 4.2.1, an object key can consume 1 KB. Each OMT node stores full keys, so the key size affects the amount of information that is written and read during a merge. Object metadata, which consume additional kilobytes, are not kept in internal nodes but can be stored in leaves to decrease the number of I/Os for HEAD requests.

Long keys can increase the time of a merge up to 2x (cf. Fig. 4.18) if a bottleneck on the throughput of processed metadata arises. Moreover, if each OMT leaf stores additional 2 KB of uncompressible metadata, the time of a merge grows even further (*Key1KBmetadata2KB* in Fig. 4.18). In such a scenario, keeping object metadata together with object data should be considered. In practice, a block containing multiple keys can be compressed when the keys are similar (typically, long keys have a common prefix). If keys are large, but blocks holding them compress well, there is hardly any impact on the merge time (*CompressibleKey1KB* in Fig. 4.18).

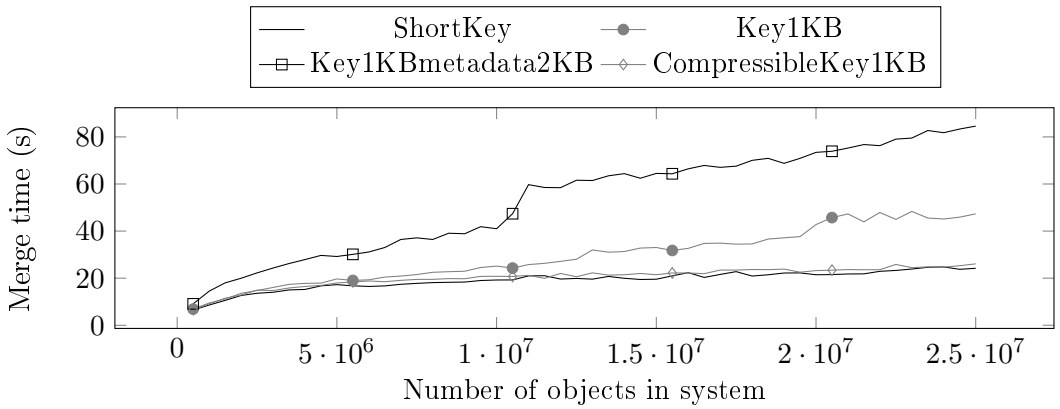


Figure 4.18: Merge time depending on key and user-defined metadata size.

Number of Objects

As shown in Fig. 4.19, irrespective of the total number of objects in the system, for the *seq* pattern, a merge takes at most 10–11 seconds, even with an artificial delay of 150 ms on the storage cluster emulating other load. This is because under this pattern the locality is high, so most of new objects are added in groups to new OMT leaves, and also few of the internal OMT nodes require rewriting.

The results are much different with the *rand* pattern, as the changes are distributed across the whole OMT. Therefore, the top levels of the OMT are almost completely rewritten, and each of the deeper levels requires up to 50K changes. In Fig. 4.19, the plot for the *rand* pattern looks almost like a linear function despite the fact that tree height is a logarithm of the objects number. Two phenomena contribute to this behavior. First, the number of internal nodes is small (about 10K for 25M objects), so most of them are rewritten when 50K randomly distributed objects are added. Second, the efficiency of caching in the storage cluster diminishes, because the larger the tree is, the less data locality.

To get more insight into how the bottleneck on reads from the storage cluster affects metadata merge for the *rand* pattern, we include results for two different types of erasure codes, that is, apart from the default 9+3, also 3+9. In *rand/9+3* and *rand-delay/9+3*, each read of a block needs 9 disk accesses. With such a volume of disk read I/Os, the merge time increases quickly for over 40M objects when the cache efficiency drops. In contrast, with

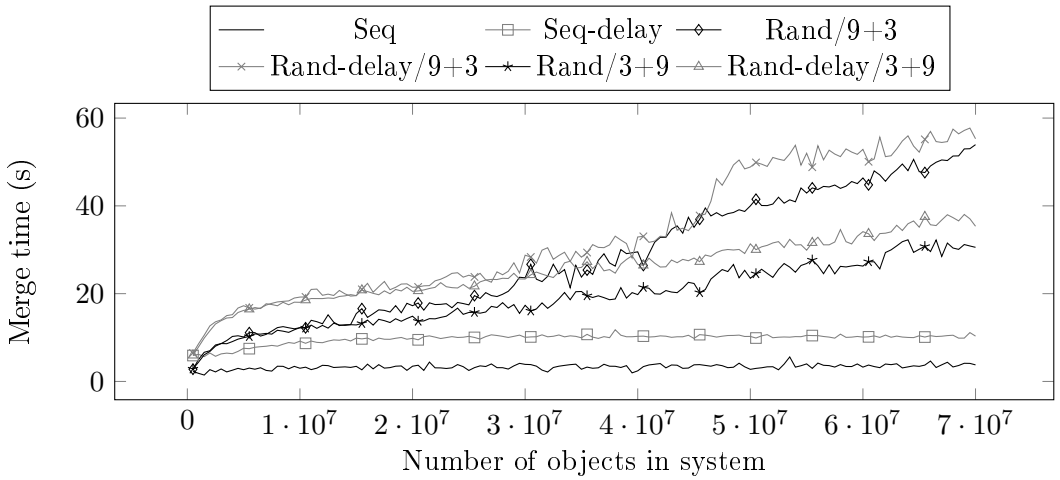


Figure 4.19: OMT merge with different key naming and EC schemes.

3+9 codes, each read needs 3 disk accesses, so the disks were not overloaded, and a merge can finish almost twice as fast. Note that there are techniques that allow for reducing disk accesses, such as adopting erasure codes requiring fewer I/Os during reads or simply caching a larger fraction of internal OMT nodes.

Distributed Metadata Merge

The last microbenchmark evaluates the distributed metadata merge in configurations of up to 16 servers hosting object drivers. In such a distributed setup, each object driver writes into its own OML, so the number of entries processed by each merge can be increased without changing the size of the OML per driver. In other words, an increased number of servers increases the number of objects processed per merge.

Figure 4.20 shows how the time of a merge changes for the *rand* pattern when the number of objects driver instances and storage servers (one per driver instance) increases with the number of entries per merge. Despite the driver coordination overhead due to the distribution, 16 servers are able to merge 800k objects faster than 8 servers merge 400k objects, 4 servers merge 200k objects, or 1 server merges 50k (using a non-distributed merge). This is because, with more objects in a single merge, the ratio of changed internal OMT nodes to leaf nodes decreases. In other words, the more changes to apply in a merge, the higher the probability that two or more objects have a common leaf or internal nodes. This characteristic enables handling the same load more than twice as fast when the number of servers is doubled.

On the other hand, the total number of objects in a system with more servers could likely be larger as well. In such a scenario, there is some loss resulting from the load distribution overhead and the increased subtree heights. For instance, for 30M objects in the system with 8 servers, a merge takes 45.335 s, while for 60M objects and 16 servers, it takes 50.951 s. Again, however, it is worth emphasizing that these results are for the *rand* pattern, which entails a lot of reads dispersed across virtually all parts of the OMT.

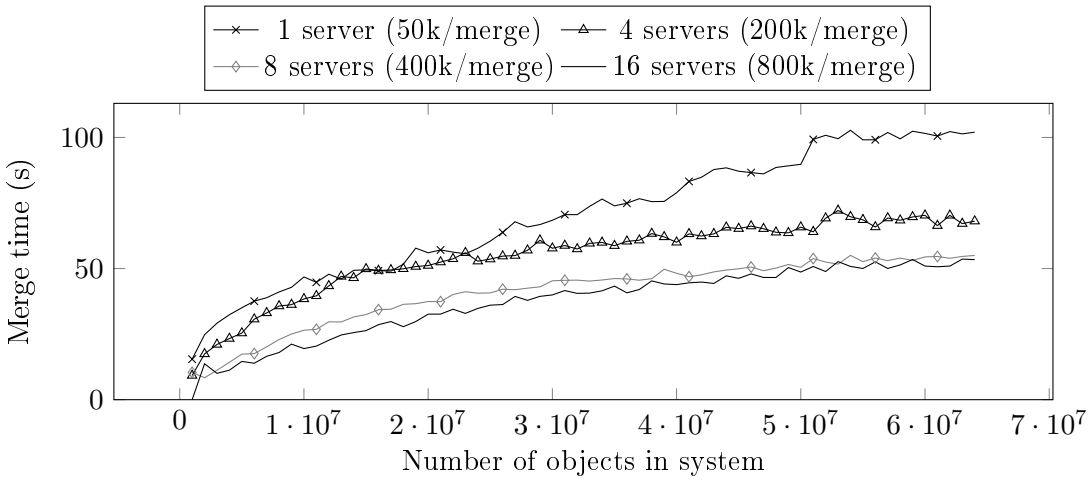


Figure 4.20: OMT distributed merge time with *random* keys.

A backup job, in contrast, typically affects only a consistent subset of the OMT (e.g., the keys have a common prefix). Therefore, we also analyze the *seq* pattern, scaling the number of prefixes for which data are written sequentially (1000 with one server, but 16,000 with 16 servers). As can be seen in Fig. 4.21, similarly to the previous *seq* experiments, the total number of objects has a marginal impact on the time of a merge. Moreover, as for the *rand* pattern, the merge time decreases with the number of servers. These are highly desirable behaviors for the considered backup applications.

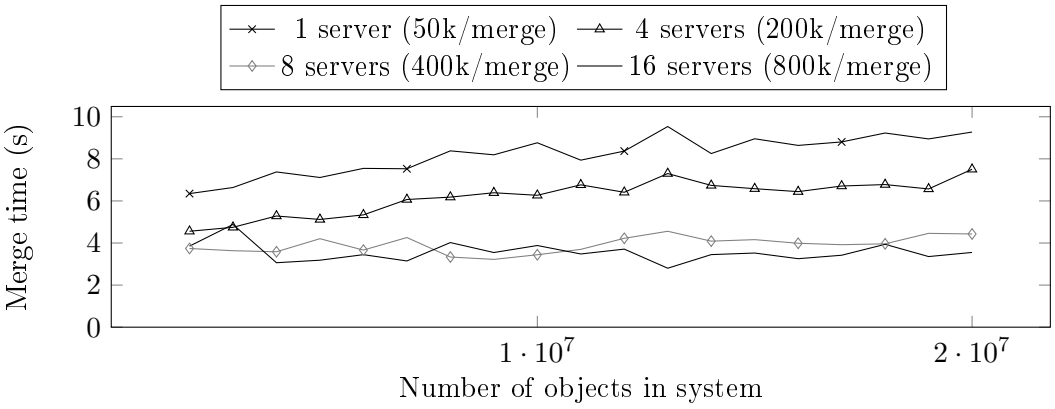


Figure 4.21: OMT distributed merge time with *seq* keys.

Conclusions

Our microbenchmark shows that our metadata structures consumes a small fraction of available resources for workloads with sequential prefixes, and also reasonably well with more malicious patterns. Even in workloads with randomized prefixes, our merging algorithms processes hundreds of operations per server per second. Moreover, as our evaluation shows, our distributed merging techniques allows effective scaling to handle thousands operations per second.

4.6. Conclusions

To sum up, there is a growing market demand for object storage interfaces in scale-out backup appliances with deduplication. Using empirical data from 686 real-world deployments of such commercial systems, we showed that a key challenge when aiming to provide support for such interfaces efficiently is the management of object metadata resulting from a different data organization and usage patterns of object storage. To address this problem, we proposed ObjDedup, a suite of distributed data structures and algorithms optimized to keep object metadata in immutable globally deduplicated block storage. We implemented ObjDedup as a layer on top of HYDRAstor and evaluated the implementation experimentally. The obtained results indicate that the performance of our solutions is comparable to that of the classic interfaces offered by HYDRAstor, despite the more challenging usage patterns. Moreover, our solutions can handle significantly more requests per second (5.26–11.34x) than object storage implementations on top of file systems provided by state-of-the-art deduplication solutions. Likewise, compared to leading object stores without in-line deduplication, it offers a much higher throughput when writing duplicate data (1.8–3.93x), not to mention the compelling storage cost reductions due to deduplication.

From a broader perspective, our preliminary study and evaluation of ObjDedup show that the trend in backup applications dedicated to object storage to write data as relatively small objects is problematic for traditional backup systems with deduplication. While ObjDedup addresses these challenges in a wide range of common configurations, we also demonstrated corner cases that are particularly hard to handle, such as extremely small objects, large keys that do not compress well, or keys without locality. As a result, we believe our study can also guide backup applications in how to adjust object writing patterns to maximize performance of storage with deduplication. Moreover, ObjDedup is an effective implementation for the object storage interface, and hence, it allows the utilization of backup appliances by cloud applications.

Chapter 5

InftyDedup: Effective Cloud Tiering with Deduplication

Managing the surging volumes of data that require protection or long-term retention increasingly necessitates novel backup strategies [30]. A popular approach is employing cloud-based solutions. For instance, according to Veeam, the number of organizations adopting cloud-powered data protection is expected to rise from 60% in 2020 to 79% in 2024 [303]. Similarly, in a survey by ESG, 72% of the participants confirmed using *tiering* techniques to move colder data (e.g., older backups and archives) from on-premise storage to the cloud [50].

In this context, deduplication can become effective and, as a result, is a core feature of several storage systems for on-premise backup applications [94, 234, 348]. In this light, for backup use cases, it is sensible to consider *cloud tiering with deduplication*, that is, moving data from a local tier (e.g., on-premise backup appliances such as HYDRAstor) to a cloud tier (e.g., a cloud object store like Amazon S3), so that ultimately the data kept in the cloud tier are deduplicated.

However, implementing cloud tiering with deduplication poses two major problems. First, state-of-the-art cloud storage systems provided by hyperscalers (e.g., Amazon, Google, and Microsoft) do not offer deduplication as a core functionality for their clients. Consequently, deduplication algorithms tailored for cloud tiering have to be developed. In the process, the extra tier should be treated not only as a challenge but also a potential opportunity for exploring novel deduplication paradigms dedicated for the cloud. Second, there is a large variety of available cloud storage service types, notably differing in pricing models. Initially, a lower storage cost implied a longer retrieval time (e.g., AWS Glacier [43]) but nowadays, systems like AWS Glacier Instant Retrieval [313] offer the same performance as other cloud storage services. The trade-off is that with a decreased per-byte monthly storage fee, the costs of data retrieval and the minimal data storage period are increased. Therefore, algorithms have to be devised to decide what type of service to use for which data, specifically considering the peculiarities due to deduplication.

As we discuss in more detail further in the chapter, despite some research progress, these two problems are largely open. In short, regarding the first problem, although a few backup applications [310, 247] and backend appliances [95, 281] with deduplication offer mechanisms for cloud tiering, they heavily rely on and are implemented mainly in the local tier. In effect, deduplication between different local tier systems is not supported for data stored in the cloud. Moreover, the entire process is fundamentally limited by the resources of the local tier. In other words, despite the possibilities offered by the hyperscalers, the actual scalability of the cloud tier in such solutions is severely limited, proportionally to what is offered by

the local tier. When it comes to the second problem, although the diversity of the service models offered by the hyperscalers can also be exploited in some solutions [232], this has to be configured manually or, at best, through policies depending on the ages of data collections. However, deduplication typically entails chunking data collections into smaller pieces that will hopefully be referenced multiple times, thereby possibly having different access patterns. This calls for finer-grained and more automated approaches to storage type selection.

In this chapter, we address both these problems, introducing solutions for scalable and cost-effective cloud tiering with deduplication. Accordingly, our contribution is twofold.

First, we present InftyDedup, a novel system for cloud tiering with deduplication. Like the existing tiering-to-cloud backup solutions, InftyDedup moves selected data from a local-tier system to the cloud, based on customer-specific backup policies. However, its operation aims to maximize scalability by exploiting cloud services—not only for storage but also for computation. Therefore, rather than relying on deduplication methods of on-premise solutions, InftyDedup deduplicates data using the cloud infrastructure. This is done periodically in batches before actually transferring data to the cloud, which, among others, enables dynamic allocation of cloud resources. Other functionalities, such as garbage collection of deleted data, are supported in the same way. We integrate InftyDedup with HYDRAstor [94], a commercial backup system with deduplication, and evaluate its performance in AWS, demonstrating that multiple petabytes can be deduplicated for a couple of dollars. Being highly independent of the local tier, InftyDedup overcomes the limitations of similar state-of-the-art technologies and offers unprecedented scalability. To the best of our knowledge, this is the first application of such solutions to backup systems.

The second contribution is an algorithm for decreasing the financial cost of storing deduplicated data in the cloud tier. It extends InftyDedup by allowing it to move deduplicated data blocks between cloud services dedicated to hot and cold storage. Whereas existing solutions do not address the problem at all or enable some optimizations at the level of data collections (e.g., backups or files), the fact that blocks are deduplicated between backups/files makes them a better unit for optimizations. In InftyDedup, the blocks are moved based on their metadata, notably deduplication reference counts and terse information provided by system administrators on their data collections. Our empirical evaluation of the algorithm shows that mixing storage types can reduce the total financial cost of cloud tiering with deduplication by up to 26–44%.

The rest of the chapter is organized as follows. Section 5.1 gives the background. Section 5.2 describes the overall architecture and specific algorithms comprising InftyDedup. Section 5.3 discusses the algorithm for exploiting cold cloud storage for cost minimization. Section 5.4 presents the experimental results. Finally, Section 5.5 concludes.

5.1. Background

To design InftyDedup, we assumed a typical implementation of deduplication system, described in Chapter 2. The data stream is chunked into small immutable blocks of size from 2 KB to 128 KB [286]; each block receives a fingerprint; the fingerprint is compared with other fingerprints in the system, and if it is unique, the block is written.

In case of tiering, a block can be removed after it has been migrated to another tier. However, reclaiming storage capacity in the presence of deduplication is nontrivial, as the system must ensure there are no other references to the removed block. Complex garbage-collecting algorithms are implemented [118, 283] which may process block metadata for hours.

In our research, we leverage the characteristics and lifecycle of backups to decrease the

total storage cost. Therefore, this section reviews the characteristics of backups, and cloud services, which are essential for the architecture of InftyDedup.

5.1.1. Lifecycle of Backups

Typically, backups are created and managed based on assigned retention policies [127]. From the perspective of our research, there are two essential constraints regarding the timing and life cycle of protected data.

On the one hand, the data should be up-to-date and available quickly in case of a disaster. For instance, Zerto reports [337] that their customers achieve Recovery Point Objectives (maximal length of period from which data is lost after a disaster) of seconds and Recovery Time Objectives (maximal time when data can be inaccessible after a disaster) of minutes. To achieve such ambitious objectives, recent data are kept as closely as possible to the infrastructure being recovered.

On the other hand, older versions of backups need to be stored for weeks, months, or even years [302]. As the objective points for older data differ, backups are often moved to less expensive storage after a specific time [311, 338]. Cloud is often chosen to keep the older backups for many reasons, including storing data in a different physical location. The pricing model of cloud storage is also appealing, but as described in the next section, many factors influence the total costs.

5.1.2. Cloud Storage

The market of cloud storage is mostly shared between three hyperscalers: Amazon Web Services, Microsoft Azure, and Google Cloud (as we described in Section 3.3). Therefore, in our considerations, we assume services offered by the three as a market standard.[1] The portfolio of hyperscalers comprises numerous storage and computing products: from databases, queues, and distributed filesystems to simple storage primitives, such as objects or blocks. Our goal is to minimize the storage cost of backups, so our research focuses on the most affordable products. The lowest price per stored gigabyte is offered by cold archival object stores, which are orders of magnitude cheaper than block devices, as shown in Tab. 5.1. However, many factors determine the total cost, including fees per request or I/O, charges for removing data before meeting the minimal storage duration, and data transfer costs. Accessing data in some types of the coldest storage takes additional time (e.g., 12 hours), but every hyperscaler offers cold storage with instant access [51, 113, 313].

	Amazon Web Services	Microsoft Azure	Google Cloud
Block Storage [$/GB]	0.08	0.15	0.04
Object Storage [$/GB]	0.021	0.0166	0.02
Archival Object Storage [$/GB]	0.004	0.01	0.004
Coldest Archival Object Storage [$/GB]	0.000 99	0.000 99	0.0012

Table 5.1: Sample monthly costs of storing blocks and objects in public clouds [2][17, 14, 111, 112, 216].

[1] However, there are numerous innovative services offered by other providers. For instance, the latest trend to decentralize the cloud [270, 272] can help to implement InftyDedup efficiently.

Uploading data to the cloud is usually free, whereas the cost of downloading data once a month can outweigh the cost of monthly data storage. In either case, network throughput to the cloud is a major concern. Hyperscalers offer connecting data centers to the cloud directly (e.g., with 100 GbE) [19, 25], but the availability of such networks is limited to specific regions. Alternatively, physical devices can be used for the movement of data [21], but it is rather for niche applications. Therefore, moving terabytes to the cloud can take up days.

5.1.3. Cloud Computing

The product portfolio of cloud computing services is also versatile, as summarized in Section 3.3. Among others, there are virtual machines (e.g., AWS EC2), containers (e.g., AWS ECS), and other services, such as event-driven function execution (e.g., AWS Lambda). The pricing model of computation services is typically based on the cost of the lower-level resources. For instance, ECS allows running containers on EC2 instances, so the cost of container execution depends on the amount and size of virtual machines which host the containers [16]. This billing model enables utilizing numerous servers (e.g., hundreds of servers) for short periods at a very low cost.

What is important for cost reduction, hyperscalers offer so-called *spot instances*, which are virtual machines with a discounted price of up to 90%. Spot instances can be interrupted by their cloud provider at any moment, but the computations interrupted within the first hour are free [23]. The exact price of a spot instance depends on multiple factors (e.g., the momentary demand), but historical data shows that achieving both a very low risk of termination and a significant cost reduction is possible [96]. Virtual machines (including spot instances) can have their local storage (e.g., SSD drives) that is less expensive than network-attached drives but has limited durability as the data are lost if the machine is destroyed or fails.

To minimize the costs of computations, we considered these cloud attributes in the architecture of InftyDedup.

5.1.4. Data Security in Cloud

A large number of publications explore security threats of deduplication in the cloud. Therefore, several methods of preventing particular attack types were proposed [49, 160, 182, 336]. Likewise, side channels leaking information from deduplication storage have been studied [32, 38]. Most threats arise from the situation in which a public cloud provider implements deduplication between users. InftyDedup is meant to be utilized by a single organization, and writing to InftyDedup requires accessing the local tier, so the situation is much different. Nevertheless, some organizations might find the deduplication side-channels as a threat within the organization, and adding security mechanisms to InftyDedup can be required. Moreover, the users of InftyDedup may not trust the cloud provider, so the local tier can encrypt data before storing them in the cloud. The structure of the data (information on block sizes and which blocks are referenced by which files) is still exposed to allow the computations, but the situation is similar in other tiering with deduplication solutions, as restoring blocks reveals the structure of files.

[2]The price of storage products depends on many factors, including region. Each cloud provides many products (e.g., each provider offers more than one cold object store). The prices between providers cannot be compared directly because the products differ. However, there are several categories of cloud storage products similar to the order of magnitude of the price. The table contains list prices as of 2023-01-01.

5.1.5. Cloud Tiering with Deduplication

DD Tier [95] is tiering with deduplication that performs its computations in the local tier, thereby imposing fundamental restrictions and limitations. First, deduplicating data between different local tier systems is impossible, as each system performs deduplication on its own. Furthermore, all or at least a large fraction of metadata is needed locally to operate. Therefore, metadata are stored in both tiers, which not only increases storage capacity usage but also forces downloading a large amount of metadata to recover even a single file. Moreover, the resources for metadata storage and processing of the local tier are limited. As locally stored metadata can consume hundreds of terabytes, the size of the cloud tier is limited (to 2x the size of the local tier). Alike, deduplication and garbage collection algorithms cannot overuse scarce local resources, especially RAM, CPUs, and disk I/Os. To this end, perfect hashing is used to decrease memory requirements below 3 bits per fingerprint. In effect, extending such solution with techniques similar to our storage type selection is very difficult.

DD Tier introduces a technique for estimating how much space will be freed from the local tier after moving data to the cloud, and in recent years, significant research attention has been paid to the problem of selecting files for efficient data removal and migration in systems with deduplication [123, 167, 221]. As long as such methods do not require storing additional metadata locally, they can be used with InftyDedup.

5.2. Architecture of InftyDedup

InftyDedup moves selected data from local-tier systems (i.e., on-premise backup appliances implemented as described in Chapter 2) to the cloud tier. The local tier is expected to have its own deduplication and to be hardware-failure resistant (e.g., by implementing erasure codes or RAID), as it persistently stores local data (e.g., data not selected for tiering). As shown in Fig. 5.1, the cloud tier stores deduplicated data with necessary persistent metadata, and occasionally executes highly optimized *batch algorithms*.

Before we describe the details of the structures and algorithms, we discuss our study of cloud characteristics (Section 5.2.1) and the assumptions we made based on them (Section 5.2.2). After that, we describe the structure of in-cloud data and metadata (Section 5.2.3), the model of communication between tiers (Section 5.2.4), and algorithms of deduplication (Section 5.2.5), garbage collection (Section 5.2.6), and file restore (Section 5.2.7).

5.2.1. Cloud Cost Considerations

We studied the pricing of public clouds to design InftyDedup in line with the current trends. First, we chose product types common for all vendors and compared the pricing models and capabilities of each product with other products of the same vendor. We did not compare pricing between vendors, as our goal was to design a cost-efficient architecture for any regular cloud, not choosing a particular vendor.

Keeping 1 PB of non-deduplicated data in a standard cloud object store costs between $\$16,600$ and $\$21,000$ per month, and between $\$4,000$ and $\$10,000$ for archival object storage with instant access. Therefore, the overall cost of storing data with deduplication, including additional storage for deduplication metadata and costs of computations, must be lower than that to bring any financial benefit.

Assuming a deduplication block size of 8 KB, a 10:1 deduplication, and 20 bytes per fingerprint, 1 PB of data requires 262 GB of fingerprints. If new backups of a similar size are

written each week, over 496 billion fingerprint existence queries to the cloud are needed each month.

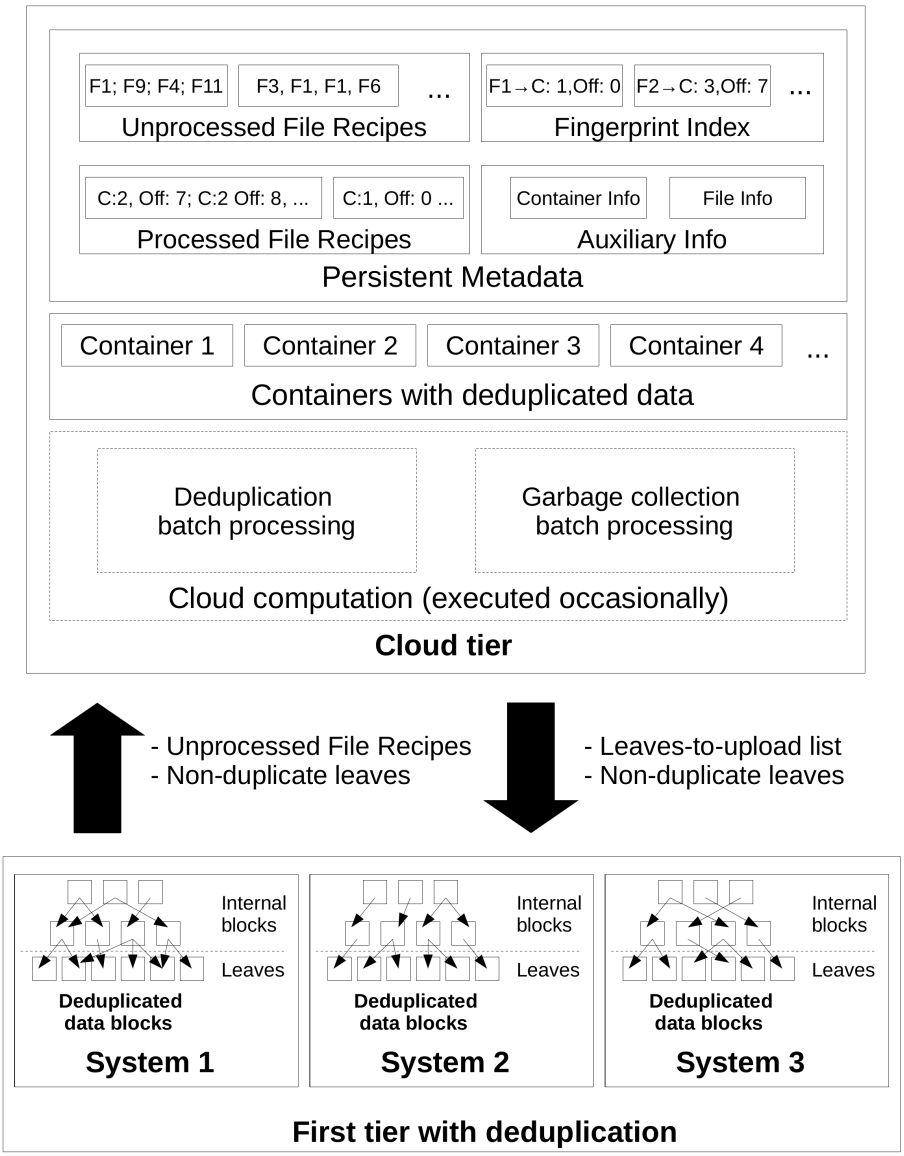


Figure 5.1: The architecture of InftyDedup.

Modern architectures of deduplication often keep the fingerprint index (or its parts) on SSDs [7, 86, 209]. Considering a naive approach in which each deduplication query requires a read I/O from an SSD drive, at least 190k I/Os per second are required to perform the necessary queries each month. To estimate the cost, let us consider AWS as an example. The monthly cost of EBS gp3 block storage which provides such an amount of I/Os per second is \$978, and EC2 instances (m5.large) capable of utilizing the I/Os cost \$3827. With a total cost of nearly \$5000 monthly for just handling deduplication queries, there is still room for a cost benefit from deduplication (depending on the deduplication ratio). Nevertheless, the price is significant compared to the cost of storage without deduplication.

These calculations led us to our conclusion that, despite the fact that SSDs provide a high number of random-read I/Os per second, relying on a random-read-intensive fingerprint

index *is not negligibly inexpensive* in the cloud environment. Although there are techniques that reduce the number of reads for traditional sequential workloads [348], their efficiency is decreased for modern non-sequential workloads, which need to be handled in addition to classic sequential workloads, as explained by Y.Allu et al. [6]. Similarly, the efficiency of methods that rely on data locality (like SISL [348]) decreases when data are highly fragmented.[3] Finally, these methods are often not prepared to update block information during deduplication, which is a necessary part of our algorithms for cold storage.

On the other hand, transferring data within the cloud is free of charge, and even the cheapest instance can transfer hundreds of gigabytes per hour [24]. Having the possibility of dynamically scaling resources between zero and hundreds of servers, processing the fingerprint index sequentially with a batch job can be more cost-effective than keeping the fingerprint index online 24/7 or relying on short-lived lambdas [20]. This is particularly true considering up to 10 times less expensive computation using the aforementioned spot instances. This key observation was used when designing the InftyDedup architecture based on assumptions explained in the next section.

5.2.2. Assumptions and Design Decisions

Our principal assumption is that *our cloud tiering deduplication must be processed outside the local tier* to overcome resource limitations and enable functionalities like deduplication between many local tier systems. Therefore, all metadata required for deduplication must be stored and processed outside the local tier.

As the network throughput between the tiers is limited, *data movement between the tiers should be minimal*. Therefore, only non-duplicate data must be uploaded to the cloud tier. When restoring data, it must be possible to download only the data absent at the local tier. However, for efficient disaster recovery, *quick and granular backup restores must be possible*, even when the local tier is unavailable.

The next central assumption is that *batch processing is preferred over streaming processing*. Therefore, the algorithms are executed occasionally (e.g., once a day or week for deduplication and even less frequently for garbage collection). There are multiple reasons for that. Firstly, as our cost analysis of public clouds shows, being prepared for data deduplication 24/7 is not negligibly cheap. Secondly, as explained in Section 5.1.1, backups are typically moved to the cloud after a specified period, so batch processing can be done without disrupting the data lifecycle. Finally, tiering to cloud with deduplication requires steps that take a significant amount of time: uploading data to the cloud, and running garbage collection in the local tier to reclaim space there. All in all, performing a costly deduplication query with each write brings few benefits in practice, and we decided to use the less expensive option of infrequent batch processing.

Garbage collection in the cloud tier must be cost-aware to ensure that data removal costs are not higher than keeping the data for a longer period. Similarly, storing frequently accessed data in cold cloud storage actually increases the costs, so the deduplication and garbage collection algorithms must be extendable with "intelligent" storage type selection.

Finally, our solution is meant to be suitable for a variety of cloud platforms and providers. Although in our description and evaluation we focus on the most popular hyperscalers, our architecture can be easily adapted to others. In particular, private clouds ensure privacy and compliance, so we verified our solution in our private cloud environment as well.

[3] Fragmentation also concerns the restore throughput [151, 189]. However, in the case of cloud storage, the read performance scales, and even with random 8 KB reads, the egress traffic cost is equal to the per-request fee.

5.2.3. Data and Metadata in Cloud

Based on the assumptions, we designed persistent structures of InftyDedup to be kept in a cloud object store as follows.

The largest structure contains blocks with deduplicated data grouped into *containers*. Selecting the size of containers depends on the cloud pricing, as writing and reading larger containers requires fewer requests but can increase rewriting costs when reclaiming space after garbage collection.

The largest metadata structure contains *file recipes*, which are effectively a list of per-block metadata as they appear in each file. If one block exists in a file multiple times, its metadata also occurs multiple times in its file recipe. There are two types of file recipes. Firstly, there are *unprocessed file recipes* (UFRs), which are provided by the local tier. UFRs contain the fingerprint of each block, as the local tier does not know the cloud location of the block. Later, during deduplication processing, each entry of UFR receives a cloud address of the block it references, so the file recipes are converted to *processed file recipes* (PFRs). A PFR can be a simple list of cloud addresses or have a tree structure to enable deduplication of its parts. In the latter case, fingerprints of the PFR blocks are added to the fingerprint index described next.

The second largest metadata structure is the Fingerprint Index (FingIdx) which contains a mapping from the deduplication fingerprint of each block to the cloud location. FingIdx is expected to be smaller than PFRs, as it contains only one entry per unique fingerprint. FingIdx is bucketed [291] rather than sorted, meaning the fingerprints are divided into thousands of buckets based on a hash function. Such a representation enables optimization of distributed FingIdx processing, as each bucket is small enough to fit into server memory.

There are also a few orders of magnitude smaller structures that keep information per file or container. The metadata structures are compressed to reduce space and network usage.

5.2.4. Communication between Tiers

The data exchange between the tiers is bidirectional but kept to a minimum, as the network connection between the tiers can easily become a bottleneck. Two types of information are sent from the local tier to the cloud (cf. Fig. 5.1). For each file selected for cloud tiering, the local tier system generates a UFR (a list of fingerprints of all blocks in the file). The UFR is later used as an input to batch deduplication, which generates in return a *blocks-to-upload list* that is, in fact, a list of containers. Each container comprises unique blocks that still need to be uploaded to the cloud tier. Based on the list, the local tier uploads the blocks to the cloud. During a file restore operation, blocks can be later downloaded from the cloud tier.

Therefore, the cloud tier has minimal requirements on the interface of the local tier. It is sufficient that the local tier is able to generate a UFR and later upload blocks based on the list of fingerprints. The local tier can be composed of multiple systems, provided that each system uses consistent chunking and fingerprinting.

5.2.5. Batch Deduplication

Batch deduplication (BatchDedup) is our distributed method of block deduplication in the cloud. It is expected to be run periodically, in harmony with the schedule of backups and garbage collection in local-tier systems. Each execution of BatchDedup is a distributed, fault-tolerant computation that ultimately changes persistent structures kept in the cloud object store. The computations are divided into steps, and each of the steps comprises smaller jobs that are parallelized and repeated in the event of failure. In our implementation, we used

YARN [300] to schedule jobs and HDFS [274] for reliable storage of temporary data. In effect, the jobs can be run on spot instances, as proposed in the AWS guide [22]. The state of computation is maintained by the YARN master node, which can be hosted on a non-spot instance to increase reliability, but even if the entire computation fails, the valid version of metadata always remains in the cloud object storage.

In short, BatchDedup takes UFRs as input, specifies new containers with blocks to be uploaded, waits until the local tier uploads the blocks, and updates persistent metadata. The UFRs are expected to be uploaded to the cloud before BatchDedup is started (partially uploaded UFRs do not take part in the process). The steps are as follows:

Step #1: UFR processing selects blocks that need to be uploaded to the cloud by comparing fingerprints from both UFRs and FingIdx. FingIdx and UFRs are bucketed based on fingerprints, and the buckets are distributed across multiple servers. After that, the fingerprints are compared in batches that are small enough to fit in memory.

Step #2: Container generation splits blocks selected in *Step #1* into containers to generate descriptions for the local tier. Each server processes a subset of blocks, and the blocks are distributed based on their original file (so blocks from the same file can be placed in the same container). The blocks are sorted by the order (offsets) in their original files,[4] as preserving the original order makes the latter step of uploading the container easier, and reduces the number of requests for garbage collection and data restores for non-fragmented data.

Step #3: PFR update is conducted after the first two steps, when the block location (its container and offset) is finally known for both new and old blocks. Based on that information, each newly written file receives its PFR.

Step #4: Block upload is initiated by the local tier systems. The local tier systems first download the descriptions of new containers (i.e., which blocks should be uploaded to what container). After that, each of the local tier systems uploads the actual data. When the uploads are successfully completed, the in-cloud metadata structures are updated to mark the new files as ready in the cloud.

The first two steps of BatchDedup are depicted in Fig. 5.2. Similar techniques are used to perform the remaining steps of BatchDedup and garbage collection at scale.

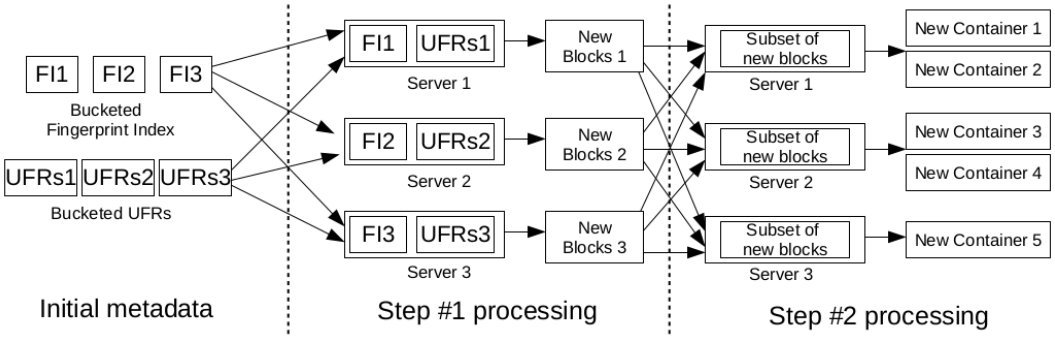


Figure 5.2: The first two steps of BatchDedup processed in a distributed manner.

BatchDedup processes FingIdx and all recently uploaded UFRs but does not touch any previously generated PFRs. As UFRs likely contain duplicates, in practice, the total size of UFRs is expected to be at least comparable to the size of the whole FingIdx, and with

[4] A block is expected to exist in multiple files or to be repeated within one file. In such a case, only the first appearance is stored in a container.

such an assumption, processing FingIdx does not dominate the asymptotic cost. Overall, the process is expected to take time: BatchDedup is executed periodically, the computations in Steps #1–#3 take from minutes to hours, and the block upload in Step #4 can even take days, depending on the data volume and network bandwidth. As Step #4 is inevitable in any cloud-tiering solution, the cloud tier alone is not suitable for providing very short RPOs. However, as backups are moved to the cloud typically after a specific time, the steps can be scheduled in periods that will not violate the timing constraints of the backup policy.

5.2.6. Batch Garbage Collection

Batch garbage collection (BatchGC in short) identifies blocks no longer referenced by any PFR and reclaims free space in the containers. BatchGC is expected to be executed periodically but less frequently than BatchDedup. Both algorithms modify the same metadata structures, so they cannot be executed simultaneously. However, file restores are possible at any moment.

PFRs keep the addresses of containers, so rewriting a container requires modifications of PFRs. The cost of processing PFRs is discouraging, as PFRs can be many times larger than FingIdx. However, garbage collection is done only occasionally, so even if it is a few times more expensive than BatchDedup, the overall cost of InftyDedup is not affected that much. Therefore, our primary goal is ensuring scalability, which enables meeting the time constraints of other garbage collection algorithms for deduplication storage [92, 283].

BatchGC comprises the following steps:

Step #1: File removal processes non-removed PFRs to find blocks that are still referenced by at least one file.

Step #2: Container verification checks how many blocks in each container are live. Based on one of the strategies (which we introduce shortly), a set of containers that will be removed or rewritten is selected.

Step #3: Metadata are updated based on the results of *Step #2*. More specifically, new metadata for modified containers are calculated. Some blocks may receive a new address, so new versions of FingIdx and PFRs are also needed.

Step #4: Containers are rewritten to actually reduce space usage. When all newly generated containers are written, the metadata computed in *Step #3* take effect, and old containers are deleted.

Immediate removal of unreferenced data is not always optimal, as rewriting a container in the cloud has a significant cost. Therefore, we investigated three strategies to decide whether a container should be rewritten:

GC-Strategy #1: Reclaim only empty containers. As in most cloud services, sending a request to remove an entire container is free, the strategy brings a cost reduction (as less capacity needs to be stored) with no additional cost. However, the strategy does not remove containers in which only a fraction of data has been deleted.

GC-Strategy #2: Reclaim containers if the rewrite pays for itself after T days. To determine whether rewriting a container will bring a cost-benefit, the following ratio can be calculated for each container:

$$x = \frac{COST_{rewrite}}{T_{days} * CAPACITY_{to\_be\_reclaimed} * COST_{byte\_per\_day}} \tag{5.1}$$

Only if $x < 1.0$, rewriting a container is less expensive than storing deleted data from the container for T_{days}. However, picking the proper value of T_{days} is nontrivial. For instance, if T_{days} is the time left until the next BatchGC, the containers are rewritten only if it brings financial benefit before the next chance to remove any data. In many cases, such T_{days} value is

too small and will prevent rewriting a container, although rewriting the container would bring a financial benefit in the long run. On the other hand, a large T_{days} value implies frequent rewriting, which can lead to exceeding *Strategy #1* costs.

GC-Strategy #3: Reclaim containers based on file expiration dates. *GC-Strategy #2* can be improved if files contain information about their expiration date (denoted EXP_{time}). Such information can be provided by the local tier systems in UFRs, if the EXP_{time} results from the backup configuration. Therefore, for each container, T_{days} can be calculated as the maximal EXP_{time} of its blocks (aligned up to the BatchGC schedule). EXP_{time} is expected to increase in time,[5] as new files with later EXP_{time} are stored. However, even with rising EXP_{time}, the cost never exceeds *GC-Strategy #1*, as a non-empty container is rewritten only when it is beneficial.

5.2.7. File Restore

The cloud metadata format supports straightforward file restores. Each file has its own object, with the key based on the local tier system identifier and file path. Therefore, the object storage interface features such as ACLs and per-prefix listings can be used for convenient file management. Based on the PFR, which stores the container address and data offset, the file can be read without accessing the local tier systems. As PFRs are updated during BatchGC, the movement of data between containers during GC does not spoil the reads.

However, egress traffic is a major cost, so restores can be additionally integrated with the local tier for cost reduction. For blocks available locally, the download from the cloud can be omitted. Blocks absent locally can be optionally stored in the local tier system after downloading, as some workloads require reading data again in the near future (e.g., restoring multiple similar VMs). Implementing such local-tier assisted reads requires storing fingerprints in PFRs, which increases the metadata size, but the fingerprints can be easily added and removed from PFRs on-demand in batch algorithms.

5.3. Cold Storage Utilization

To reduce the cost of storing data in the cloud, InftyDedup can be extended with an algorithm that selects whether a block should be stored in hot or cold cloud storage. We aimed to use cold storage services offering different pricing models than other cloud storage products but comparable durability and latency [113, 313]; otherwise, the movement of data to cold storage would negatively affect the recovery time.[6] Therefore, we focused on colder storage which offers a reduced price of storing data but increases the price of restores and imposes a minimal storage period (e.g., 90 days). To utilize the storage effectively, we rely on two additional pieces of information provided with each file (in UFRs):

1. **Current expiration date**, as in *GC-Strategy #3*.
2. **Rough, expected frequency of file restore.**

As explained earlier, the expiration time is typically known. The restore frequency is unknown in advance. However, assessing the read frequency of a file is a common practice for data kept in the cloud. For instance, Amazon explicitly recommends different storage classes for data accessed "once per quarter" and "1-2 times per year" [18]. In the specific case of backups, assessing the restore frequency should be possible, as a study of numerous

[5] Theoretically, EXP_{time} can decrease if someone deletes a file before the expiration date. We find such a case rather marginal. In particular, enabling WORM protection [309] prevents such removals.

[6] Our algorithms can also work with the coldest storage services, which lengthens the retrieval process. However, in such a case, additional information is needed to specify the allowed retrieval time of each file.

backup jobs [26] suggests that backup domains fall into three categories: those with very frequent restores, sporadic restores, and virtually no restores. Moreover, particular backup policies influence the restore frequency [241], and an upper bound on the restores can be calculated based on restore service-level agreements (SLAs). Finally, modern backup software already implements tools that allow viewing historical data on the restore frequency of selected resources [307].

The persistent data and metadata structures are organized as shown in Fig. 5.3.

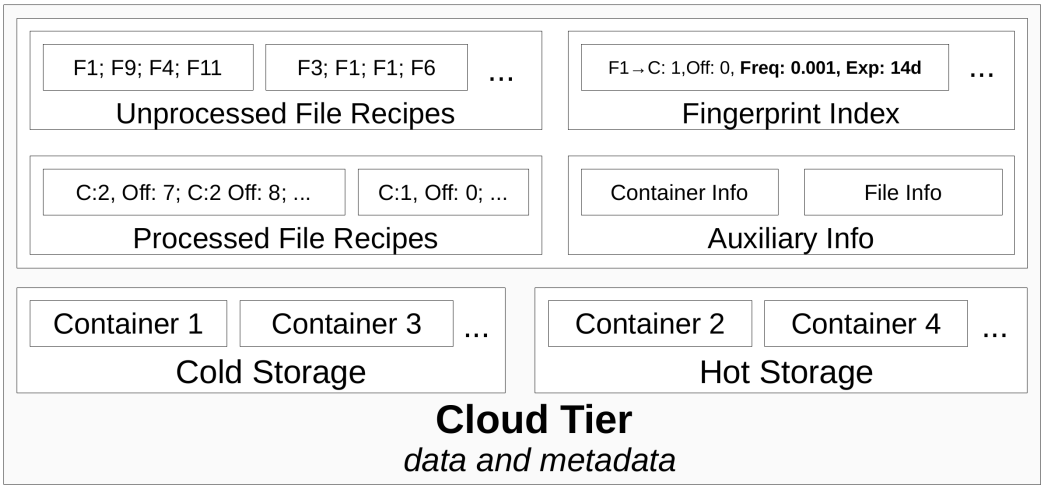


Figure 5.3: Architecture of data and metadata with two types of data storage (hot and cold). Fingerprint Index is extended.

The process of container writing during BatchDedup and BatchGC is extended to store each block in an appropriate cloud storage type, as shown in Fig. 5.4.

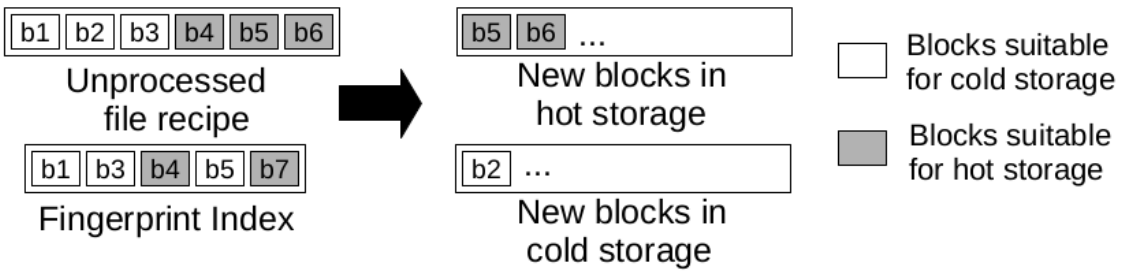


Figure 5.4: Writing blocks to more than one storage type. Although it is already available in colder storage, block *b5* is written to hotter storage if it brings a cost benefit (e.g., due to expected frequent restores of *b5*).

Each block is stored in a storage type for which the following formula has lower value:

$$t{=}COST_{insert}{+}(COST_{B/day}{+}COST_{restore}{*}FREQ_{restore}){*}EXP_{time} \tag{5.2}$$

In the formula, $COST_{insert}$ depends on cloud pricing, as well as the sizes of the block and its container, as the amortized cost of data insertion is included. $COST_{B/day}$ describes the storage cost of the block. $COST_{restore}$ depends on the data locality, as many blocks can be read with one request, so the upper bound for the $COST_{restore}$ can be calculated as *one*

request per block or assessed with a heuristic. $FREQ_{restore}$ and EXP_{time} are inherited from files referencing block and are stored with each block in FingIdx.

However, further adjustments to $FREQ_{restore}$ and EXP_{time} are required. This is because the first decision about the storage type must be taken when the block is stored for the first time and $FREQ_{restore}$ and EXP_{time} are underestimated, as more references to the block will likely come in the future. For instance, a block can be initially stored in cold storage but later it receives more references (and its $FREQ_{restore}$ increases). Vice versa, data with a short EXP_{time} can be kept in hot storage, although a reference with a larger EXP_{time} may come soon.

Therefore, both $FREQ_{restore}$ and EXP_{time} should be heuristically modified. A heuristic that worked well in our experiments relies on block reference counts. First, we select a number R of expected references for each block (e.g., a hard-coded value 5 or a value calculated from the system state). Then, we modify $FREQ_{restore}$ and EXP_{time} for blocks that have not reached the expected number based on the formula (e.g., we multiply it by $R - r$, where r is the actual number of references). In the end, $FREQ_{restore}$ and EXP_{time} for newly written blocks are more similar to their future values.

In justified cases, a block can be stored in multiple storage types (e.g., when a block stored in cold storage receives a reference with a high $FREQ_{restore}$), but BatchGC will eventually remove the unnecessary copies. Similarly, BatchGC can move a block from one type of storage to another (e.g., when a reference with high restore frequency has been deleted). Generally, during BatchGC, a formula for calculating whether a container should be rewritten considers the potential cost reduction caused by a change of the storage type. A decision on whether rewriting a particular container is profitable must be made for the whole container because rewriting the container also introduces costs. Nevertheless, blocks from one container can be moved to containers in various storage classes (see Fig. 5.5).

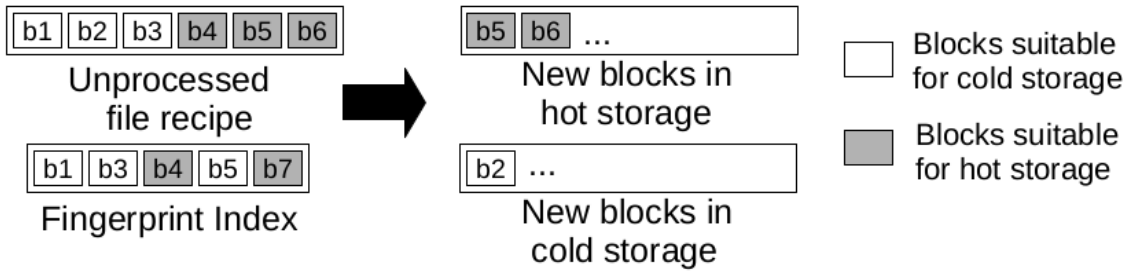


Figure 5.5: Rewriting containers to multiple types of storage.

5.4. Evaluation

We present our experimental evaluation of InftyDedup in two parts. First, we evaluate the performance and cost of our implementation executed in a public cloud. Second, we evaluate our strategies for garbage collection and storage type selection under various workloads.

5.4.1. Performance Evaluation

To evaluate the performance, we implemented InftyDedup using Apache Hive [60], which we selected as a possible approach to provide portability between different public and private clouds. We present results of the evaluation of our batch algorithms, as uploading containers

and restoring data are straightforward object storage operations in which the bottleneck is expected mostly on the network to the cloud (even a naive implementation can saturate 1 GbE network with uploads and restores using a single core).

Our batch algorithms differ greatly from the state-of-the-art tiering to cloud with deduplication techniques. Therefore, a fair comparison with existing solutions is virtually impossible. Instead, we present the results using publicly available hardware. The evaluation was conducted in AWS using m5d.xlarge instances with 4x vCPU, 16 GB of RAM, and 1x 150 GB NVMe (which costs less than network attached EBS). We aimed to use the smallest possible instances (to maximize horizontal scaling). However, in our workloads, the technological stack of Apache Hive was inefficiently utilizing the limited memory of the smallest instances.

The presented experiments used synthetic data with the following characteristics. Each file contained approximately 51 GB (as backup files typically have tens of gigabytes or more [315]) chunked into blocks of approximately 64 KB (the target block size of the deduplication system for which we prepared InftyDedup). The contents of the files are described in each experiment. We present results with synthetically generated data, as our algorithms mostly distribute the data (e.g., based on fingerprints) and later sort the data in small portions, so the exact characteristic of the data (e.g., the initial order of blocks) does not affect the performance much.

Batch Deduplication Processing

We evaluated BatchDedup in configurations varying in size. Each experiment comprised two steps. In the first (initial) step, a large number of files without duplicates is processed to resemble a situation in which new backups are uploaded to the cloud. In the second (incremental) step, a dataset 3x smaller than the initial backup is uploaded (as typically incremental backups are smaller than their corresponding full backups [26]), where 90% of the blocks are duplicates (which matches the expected average daily deduplication ratio [26]). The smallest configuration (8 instances) uploads 3072 files in the first step and 1024 in the second step. In larger configurations, the amount of data to be processed is scaled linearly with the system size. Therefore, the smallest experiment processed metadata of 208 TB data and the largest one of 1.66 PB. In all configurations, the first step takes between 1h53m and 2h10m (see Fig. 5.6), and the second step takes up to 30m.

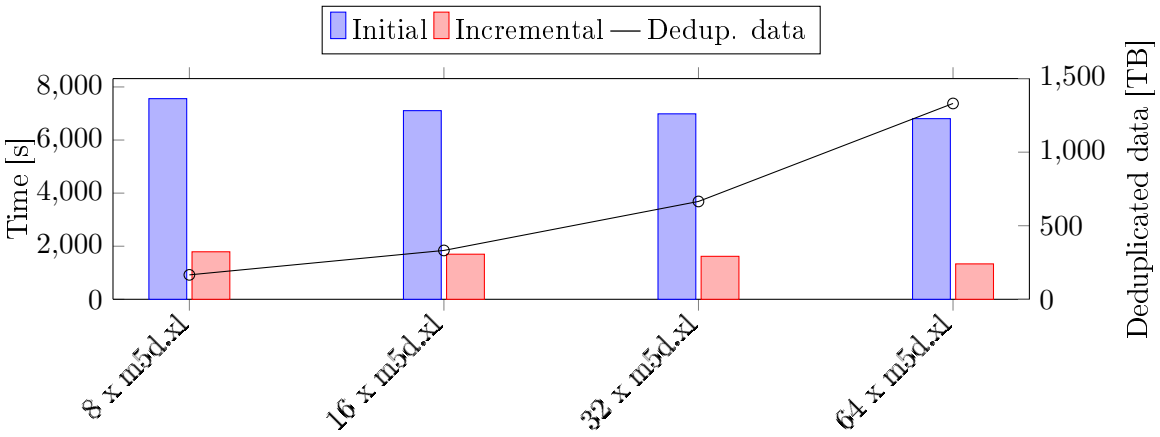


Figure 5.6: BatchDedup performance. The line and right y-axis show size of deduplicated data.

Overall, the performance scales close to linearly, as two times larger experiment with twice

as many machines achieve similar computing time. We analyzed the resource utilization, and the main bottleneck is the CPU, as most of the time its usage is above 95%. The network and the local SSDs are underutilized, with the peak per-server usage of respectively 350 MB/s of network bandwidth and 6% of disk utilization. We expect the computations can be further optimized, but the computations are already marginal compared to the costs of data storage. For instance, in the experiment with 32 instances, the second stage eliminates 191 TB of duplicates and costs below $1, which is less than 0.1% of monthly savings on storage. Similarly, the costs of accessing in-cloud metadata during processing are marginal, as both steps require roughly 250K GETs ($0.1) and 20K PUTs ($0.1), and the transfer fees within one availability zone are free.

We also conducted a different experiment with multiple steps of incremental uploads in one configuration (8 instances). As shown in Fig. 5.7, the incremental uploads are much shorter, as only a small fraction of new data is added to FindIdx. Each of the incremental steps take similar amount of time but later steps are slightly longer, because there are more unique blocks in FingIdx.

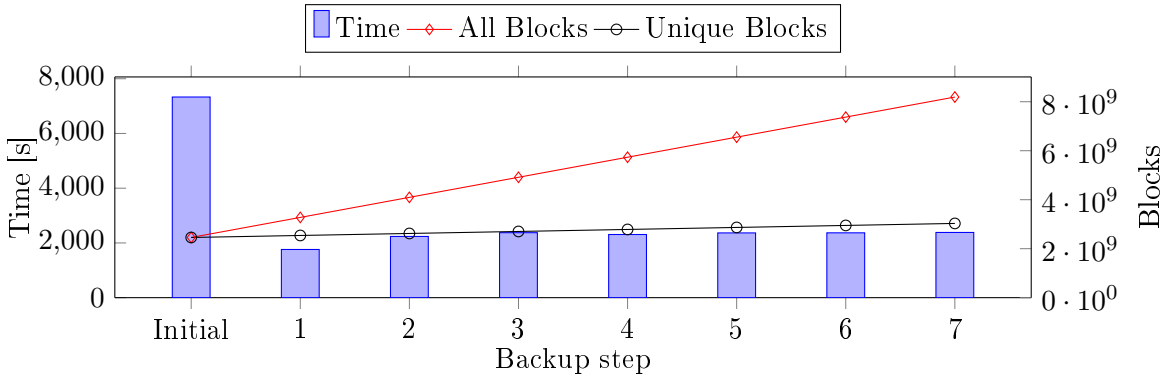


Figure 5.7: BatchDedup with growing data. The lines and right y-axis present the number of blocks before deduplication (all) and after deduplication (unique).

Batch Garbage Collection Processing

First, we evaluated BatchGC by removing a fraction of data uploaded in the first experiment from Section 5.4.1. Specifically, we removed the data uploaded in the first step to resemble removing the oldest backup. The processing took between 61 and 65 minutes (not plotted). BatchGC, unlike BatchDedup, reads all PFRs, so we also verify that the processing time increases close to linearly with the size of both FingIdx and PFRs. The experiment shown in Fig. 5.8 had multiple incremental steps, and in BatchGC we removed data from the first incremental step.

The results confirm that, for data with many duplicates, BatchGC is more expensive than BatchDedup. However, BatchGC is expected to be executed less frequently, so both algorithms will have comparable amortized execution costs.

5.4.2. Evaluation of the Strategies

We evaluated how our garbage collection and storage type selection strategies behave in numerous workload simulations. The strategies optimize the costs of storing data for months and years, so we could not conduct these experiments in the public cloud, as it would take too

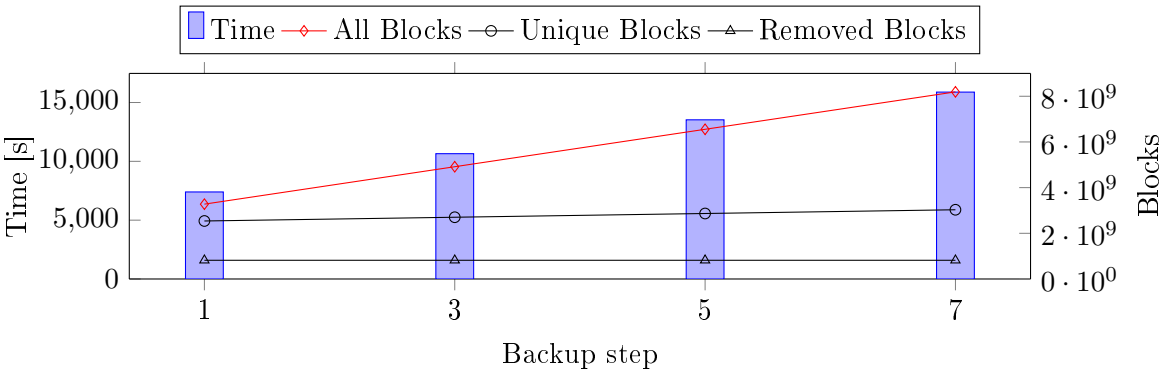


Figure 5.8: BatchGC performance. After 1-7 incremental steps, data from one incremental step was deleted (removed blocks).

long. Instead, we ran some initial experiments to confirm our understanding of the pricing model and features of the cloud, and based on the results, we implemented a simulator. The simulator calculates costs based on the pricing of cloud storage, requests, transfer, and other factors, like the minimal storage duration.

Each experiment was conducted in many configurations of workload characteristics and system parameters. We present aggregated (minimal, maximal, and average) results, with values normalized to the result with the minimal cost.

Workload Characteristics

Our simulator allowed specifying the following factors to evaluate various backup workloads:

Data source was selected from the following two sets. First, we generated synthetic workloads in which a given fraction of data was *modified* and *deleted* each day. Both types of modifications were applied in variable length *stream-contexts* (of size from 1 to 1024 blocks), so a given number of consecutive blocks was modified at once. The introduction of the stream-contexts was necessary, as data modified in small contexts is more fragmented, so the number of requests required to read is increased. Secondly, FSL traces [292] were used, as they are real-world datasets that contain information on how the data of multiple users change over years.

Retention policy specifies how long each file (backup) is stored. We analyzed guidelines related to retention policies [2, 93, 106] to generate realistic policies. Typically, each type of backup is stored for a longer time than its backup period (e.g., weekly backups are kept for four weeks). In our experiments, daily backups are kept for one week, weekly backups are kept for a month, monthly backups are kept for a year, and yearly backups are kept for five years. Based on that, we came up with three different policies: *keepAll* policy in which all types of backups are stored in the cloud, *dailyExcluded* in which daily backups are excluded (so only backups stored for at least a month are kept in the cloud), and *dailyOnly* in which only daily backups are kept in the cloud. The garbage collection was, in turn, executed every 7, 30 or 90 days. In all experiments, the simulation covered a period of 5 years.

Read patterns remarkably affect the total cost of ownership of data in the cloud. Unlike for writing data, we found no collected read traces for backup data. Similarly, there are no precise guidelines that describe typical backup read patterns. Therefore, we adopted a model in which each file is read with a given probability and verified the full spectrum of potential values.

Evaluation of Garbage Collection Strategies

To evaluate how the proposed garbage collection strategies perform in different workloads, we conducted experiments with the pricing model of *AWS S3 Standard* as *hot* storage and *Glacier Instant Retrieval* as *cold* storage.[7] Experiments in which the storage types are mixed based on our strategy are denoted as *mixed*. Garbage collection strategies are denoted as follows: Strategy #1, which removes a container only when it is empty, is denoted as *onlyEmpty*; *less{25; 50; 75; 99}* denotes Strategy #2 with the T_{days} parameter such that the behavior is equivalent to reclaiming space when less than *25 / 50 / 75 / 99* percent of container capacity is used by live data; and Strategy #3 is denoted as *costBased*.

First, we evaluated a case in which there were no reads. As shown in Fig. 5.9, *onlyEmpty* strategy achieved the worst results. For *cold* and *mixed* storage, *costBased* strategy gave significantly better results than others (on average 1.4%-23%), whereas for hot storage (where the rewrite cost is marginal) it gave similar results to *less99*.

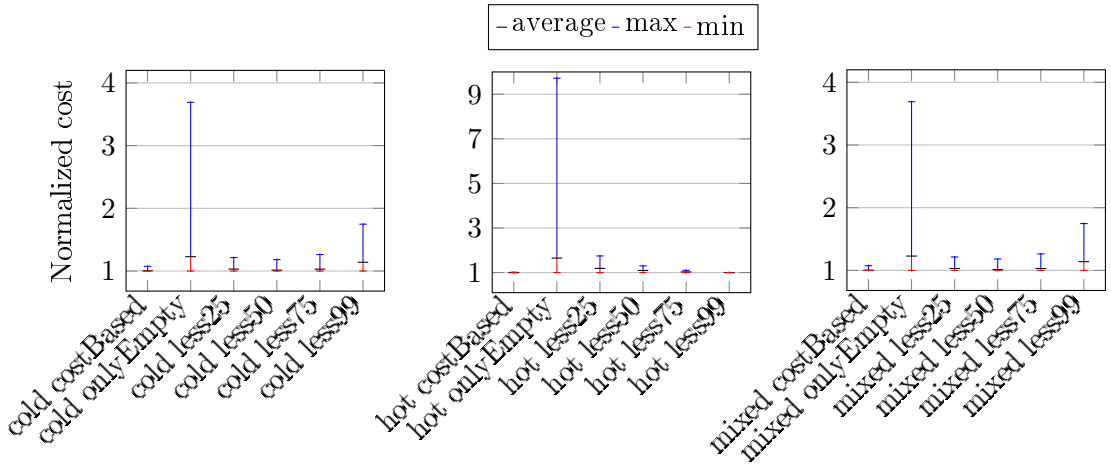


Figure 5.9: Garbage collection with different strategies.

In the next set of experiments, which included reads (with patterns explained in Section 5.4.2) and *mixed* storage, there are more differences between the strategies (Fig. 5.10). On average, *costBased* strategy is only 2.2% better, but comparing the worst cases, the difference is 24%. The analysis of the number of containers that are rewritten, deleted empty, or remain live at the end of the experiment, confirms that *onlyEmpty* has the largest number of containers that are live (Fig. 5.11).

[7] At the moment of writing this dissertation, cold storage had 4x/25x more expensive PUT/GET requests, 5.25x times less expensive storage costs, the minimum storage duration was 90 days, and an additional per-gigabyte retrieval cost for cold storage was equal to the fee for 3000 GET requests.

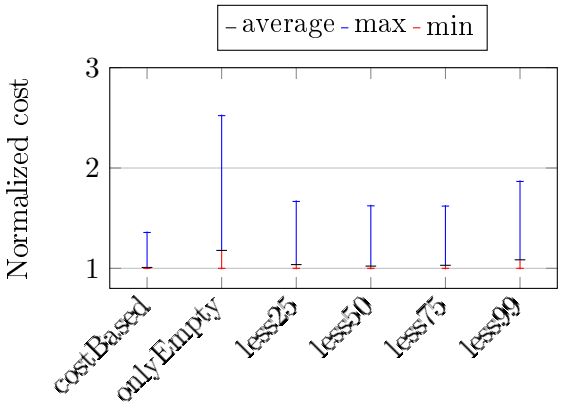


Figure 5.10: Garbage collection strategies with reads.

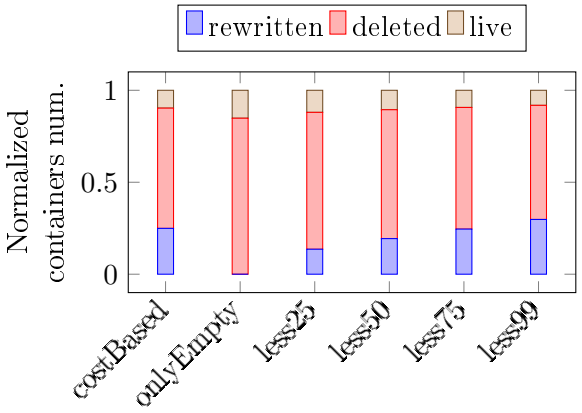


Figure 5.11: Breakdown of containers number.

The analysis of garbage collection strategies led to the question of how container sizes affect the costs, as smaller containers increase the probability of removing an entire container but also increase the number of PUT requests needed to store data initially or during container rewriting. As shown in Fig. 5.12, for *costBased* strategy, the lowest average cost is with 16 MB containers (4 MB and 64 MB are respectively 4.5% and 2% more expensive). The smallest, 1 MB containers were the most expensive, even with the *onlyEmpty* strategy, because of the cost of initial container generation (Fig. 5.13). Especially in *cold* storage, the cost of PUT requests is high (up to 40% of all costs with 1 MB containers).

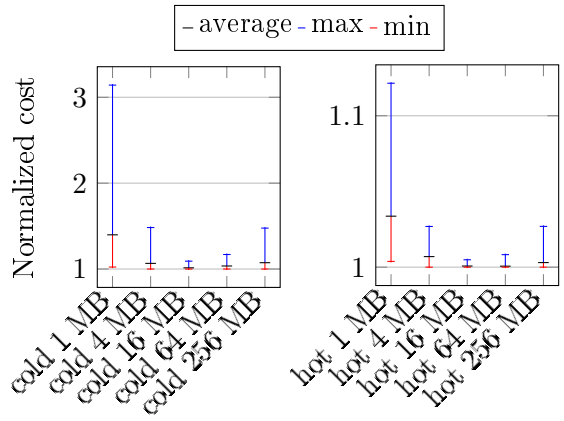


Figure 5.12: Garbage collection with varying container sizes.

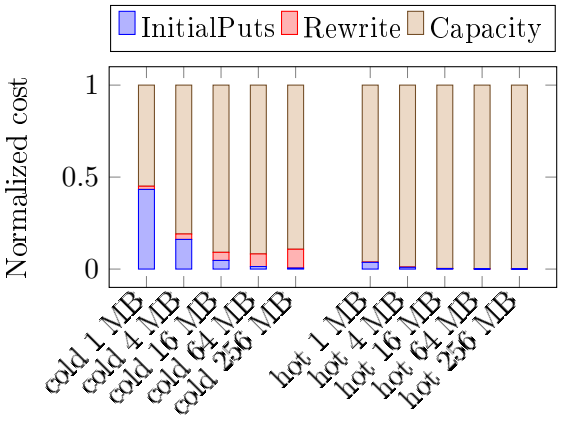


Figure 5.13: Cost breakdown with different container sizes.

Storage Type Selection

We evaluated our storage type selection strategies in workloads with varying read frequencies. For each experiment, there were 4 synthetically generated sets of files, and each set has a different read frequency: once a month, once a year, once a year with 1% probability, and once a year with 0.1% probability. All 4 sets were written together, just as in a storage system that keeps files with varying read frequencies. The experiments were conducted in series, and in each series, the read frequency was scaled by a factor from 0.001 to 10. Therefore, cases in which reads are virtually nonexistent, cases in which reads dominate the total cost, and cases

in-between were evaluated. A real-world ratio between backup and recovery jobs is typically 100 : 1 [28] but varies depending on the system [26]. In our experiments, the ratio of backups to recoveries for scale factor 0.01 is 70–700:1 (mean =216:1) depending on the retention policy. Therefore, we expect results with scale factors 0.01 and 0.1 to reflect a typical use case.

As shown in Fig. 5.14, on average the *mixed* strategy gives 55% cost savings compared to *cold* if there are many reads and 70% compared to *hot* if there are hardly any reads. The breakdown of newly created containers (Fig. 5.15) confirms that data ends up in cold storage when there are hardly any reads and in hot storage reads are frequent. The cost breakdown (Fig. 5.16) confirms that the *mixed* strategy balances the high storage cost in *hot* and the expensive reads in *cold*.

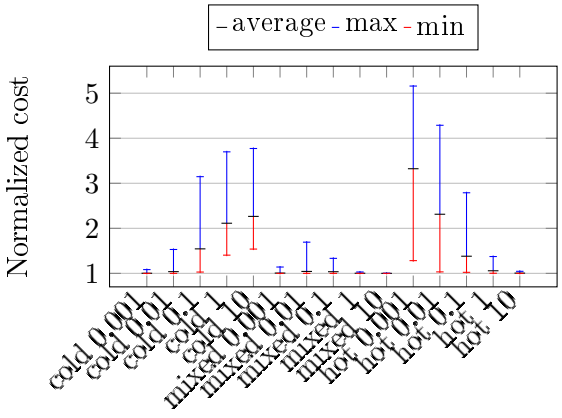


Figure 5.14: Storage type selection depending on the read frequency.

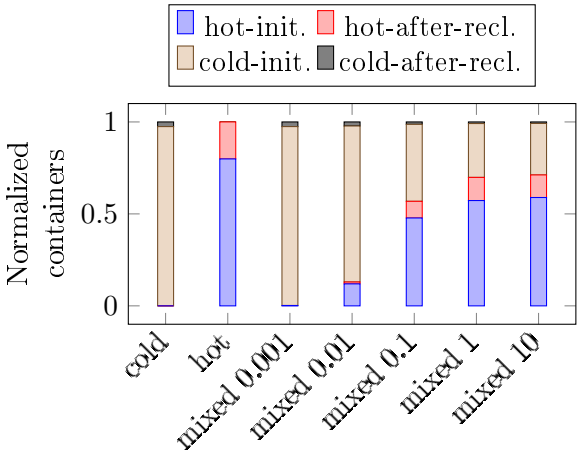


Figure 5.15: Containers created initially and after reclamation.

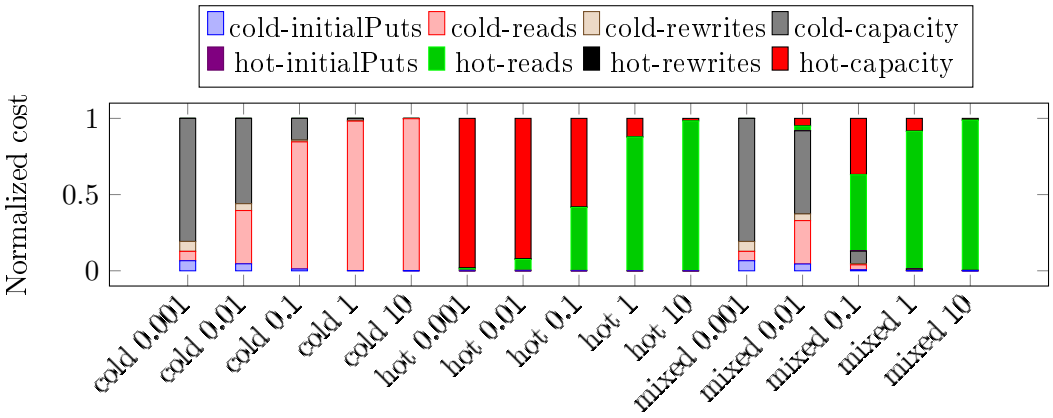


Figure 5.16: Cost breakdown with varying read frequencies.

We also evaluated how the changes in the predicted reference counts affect the cost. Fig. 5.17 presents the normalized cost, depending on the selection of the expected reference number. Without predicting that more references will come in the future, the cost is higher on average by 11% (worst case 289%) compared to predicting 5-10 references, so we confirmed that predicting the number of references brings a significant cost reduction. The results with 3-10 references are very similar, so the slight inaccuracies in the expected number of references

do not change the results much.

The mixed strategy depends on the expected frequency of reads, which may be incorrectly assessed. We conducted experiments with a significant prediction error (the value was underestimated and overestimated ten times). Even with such a large estimation error, the results are close to perfect (Fig. 5.18). Therefore, in all remaining experiments, we assumed perfect estimation to facilitate studying other experimental parameters.

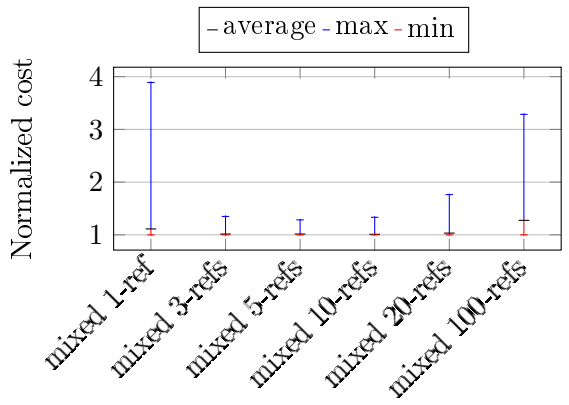


Figure 5.17: Cost of storing data depending on the expected number of references.

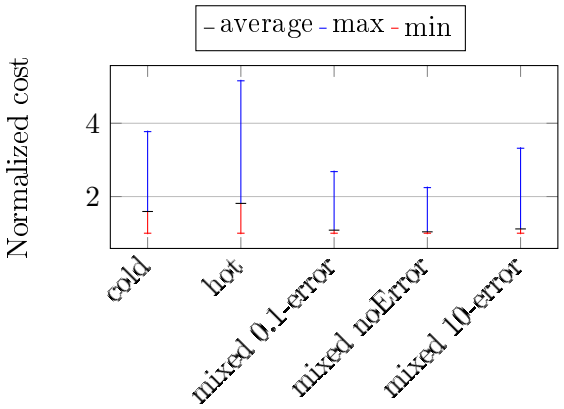


Figure 5.18: Cost of storing data depending on the error of frequency prediction.

Different Public Clouds

To confirm that our strategies are generally applicable to public clouds other than AWS, we repeated most of the experiments with the pricing models of Google Cloud and Microsoft Azure. As our evaluation shows, mixing cold and hot storage reduces the costs for all three major providers (Fig. 5.19). The noticeable differences in gain between the cloud providers follow from the different ratios of costs, especially the cost of storing data and egress traffic. On average, keeping data only in hot storage is 61% more expensive, and keeping data only in cold storage is 30% more expensive than using the mixed strategy.

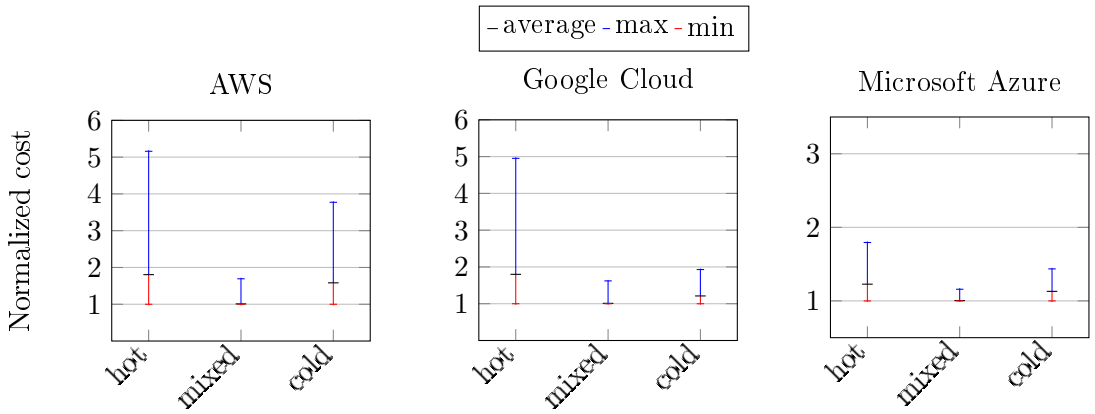


Figure 5.19: Storage type selection in different public clouds.

FSL Traces

Finally, we verified our strategies using Filesystem and Storage Labs (FSL) traces [292]. Specifically, we used all data available with 64 KB chunking in the *homes snapshots* dataset. The traces contain metadata of files chunked during writing, but they have no information about the read pattern. Therefore, for each user, we verified how our storage type selection works with a varying number of reads (restoring each backup with a frequency from 0.0001 to 1 time a month). As shown in Fig. 5.20, for the extreme read frequencies, the mixed strategy keeps almost all the data in the less expensive of the two storage types. However, if the number of reads is in between, the mixed strategy works better than keeping data in a single type of storage, because depending on the data characteristics, a different decision should be made for each block. In particular, the characterization of the reference count of each block is important, as frequently referenced blocks are accessed more often. Therefore, mixing storage types can outperform keeping the data in one storage type, decreasing the cost by 26%–44%. This result shows that even when the restore frequency of each file is known in advance, relying on selecting one storage type can be significantly more expensive than using our mixed strategy.

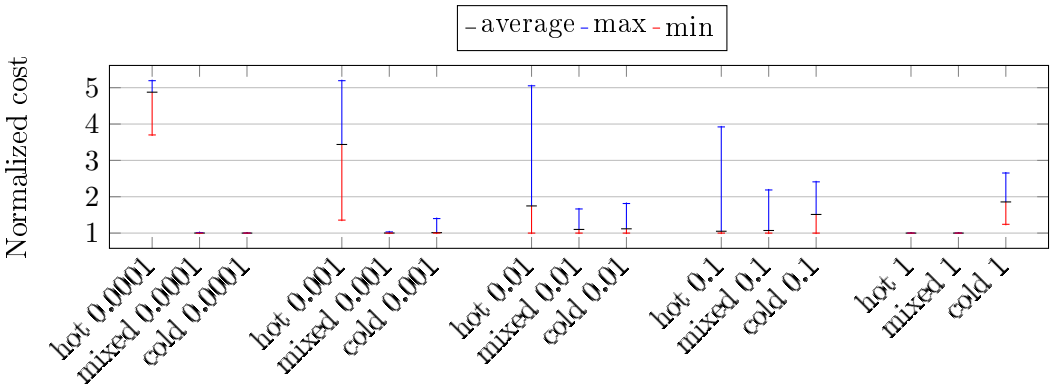


Figure 5.20: Total costs in experiments with FSL traces.

5.5. Conclusions

We presented InftyDedup, a novel, cloud-native approach to tiering to cloud for a storage system with deduplication. Compared to the state of the art, our architecture does not impose any limit on the size of the cloud tier and supports deduplication from multiple local tier systems. We implemented InftyDedup for a commercial storage system (HYDRAstor) and evaluated it in a public cloud (AWS). The evaluation confirmed the desired scalability of deduplication: our batch algorithms, designed to reduce cloud costs and harness dynamic resource allocation, were able to process metadata of multi-petabyte data collections for a couple of dollars.

To further decrease the cost of cloud storage, we proposed an extension to InftyDedup which moves blocks between hot and cold cloud stores based on their anticipated access patterns. Its evaluation with real-world traces showed that our deduplication-specific heuristic for adjusting the expected read frequency, which takes into account block reference counts, decreased the costs on average by 11%, and the overall solution achieved 26%–44% reductions. The algorithm requires minimal input from a system administrator and was demonstrated to retain its cost benefits even when the administrator's estimations were inexact.

Chapter 6

Derrick: Balancer for Resilient and Efficient Distributed Storage

The management of physical data placement across devices is a fundamental problem that virtually any distributed system has to address. Especially distributed storage systems, which are normally responsible for maintaining data for other tiers, have to deal with the many intricacies of this problem. In particular, to decrease the risk of data loss and shorten failure handling, such systems have to replicate or erasure-code data chunks and disperse them across different physical devices. To optimize capacity utilization, they have to balance the amount of data between the devices while also accounting for the underlying network characteristic so that the cost of keeping the redundant chunks in sync is acceptable. When new devices are added, or existing ones fail, the systems have to move or reconstruct data chunks, ideally in a way that minimally affects the performance of the core functionality. These are just a few common examples of data placement requirements, and many applications also have their own specific ones.

For these reasons, as we clarify in the next section, the problem of physical data placement in storage systems has received considerable research attention. A lot of that attention was focused on large-scale cloud-oriented storage systems, namely data-center-wide systems with thousands of machines or global geo-distributed systems that can be orders of magnitude larger. In contrast, relatively little work was dedicated to *on-premise scale-out storage systems*, such as HYDRAstor [94], Ceph [320], and Swift [128]. On the one hand, given the data deluge in today's digital societies, a market for such solutions is thriving (IDC expects a 4.7% compound annual growth rate in External OEM Storage [136]). On the other hand, however, managing data placement in such systems poses unique challenges, which cannot be effectively addressed solely by adopting solutions developed for public clouds.

More specifically, these challenges stem mainly from the life cycle of on-premise scale-out storage systems. Once deployed, such a system is controlled by the owning client. Consequently, it should hardly require human intervention, instead being largely self-managed. Furthermore, to accommodate the ever-accumulating data, the systems are typically expanded, often repeatedly, which has two key implications. First, an average system comprises multiple generations of hardware that inevitably offer different performance characteristics and capacities. Nevertheless, despite this heterogeneity, the system is expected to ensure high utilization of all available resources. Second, it is not uncommon for a single system to grow even by orders of magnitude. Supporting tiny configurations is thus as important as large ones to provide flexible scaling. For instance, Scality [130] reduced the minimal system size to three servers to solve the challenges of small and medium-sized enterprises. Again, the

performance overhead due to the growth must not be observable; on the contrary, the performance scaling should be as close to linear as possible to justify the expansion costs. In short, on-premise scale-out storage systems are expected to offer what we have dubbed *self-managed continuous scalability*: a single system must be capable of autonomously maintaining high resource utilization even when it is expanded by a few orders of magnitude with heterogeneous hardware.

While self-managed continuous scalability may seem like a natural requirement, it is hard to meet in practice. This observation is consistent with a common computer systems design principle, referred to as the incommensurate scaling rule, which states that changing a parameter of a system by a factor of ten usually requires a new design [264]. In the context of data placement, when formalized, many issues are NP-hard problems [108, 131]. Consequently, algorithms for large-scale deployments are inherently heuristics that rely on probabilistic or asymptotic properties (holding only for sufficiently large systems) while at the same time emphasizing different aspects, notably fault tolerance. In contrast, small-scale deployments sometimes make it possible to efficiently search the entire solution space to find optimal placements. In other words, algorithms employed for managing data placement do vary depending on the scale. Furthermore, as we elaborate shortly, they are heavily affected by practical issues, notably conflicting requirements with respect to placement, hardware heterogeneity, and traffic considerations, to name just a few. Controlling data placement in systems that ensure self-managed continuous scalability thus indeed requires special solutions.

The relevance of these issues is reinforced by a recent report by Gartner [242], which argues that scalability and flexibility (e.g., handling device additions without disrupting other operations) with simultaneous cost reductions are the current challenges in on-premise storage. New technologies, such as energy-assisted magnetic recordings HDDs [249], bring new possibilities, but keeping up with the newest hardware requires significant software changes. According to Coughlin Associates [82], the vast majority of capacity still is—and will be in the foreseeable future—shipped in HDDs. The problem is that the performance of HDDs does not improve as fast as their capacity,[1] resulting in fewer than 10 IOs per second per terabyte in modern drives. High performance and flexibility are very hard to achieve with such a limited value of I/Os, so attentive data placement across devices and thrifty data movements are necessary when aiming at continuous self-managed scalability.

This chapter presents Derrick, an algorithm for managing data placement. We implemented Derrick in our commercial backup and archival storage system, HYDRAstor, and verified it in production. A key observation behind Derrick is that data arrangement has different requirements and timing constraints depending on particular events that trigger its changes. Therefore, aiming at self-managed continuous scalability precludes a one-size-fits-all approach. To better explain the trade-offs and prioritizations due to conflicting needs, we analyze common requirements on data placement in on-premise scale-out storage systems. The study is based on: empirical data and our experience from thousands of HYDRAstor deployments worldwide, an examination of Ceph and Swift (state-of-the-art open-source systems used in multi-petabyte-scale installations), and a survey of other scale-out storage systems based on publicly available materials.

Instead of using a one-size-fits-all approach, the data arrangement in Derrick involves three sub-algorithms called Central Balancing (CentrBal), Transition Guide (TrGuide), and Distributed Balancing (DistrBal) to handle different cases. CentrBal computes a data arrangement for a perfectly stable system, disregarding hardware failures. To make data migration

[1] Multi-actuator HDDs can improve performance considerably, but the technology is fresh, so their pricing and availability in the next years are unclear.

smooth, TrGuide computes a transition plan between two arrangements provided by CentrBal. Both CentrBal and TrGuide are allowed to run calculations for minutes or even a few hours, as they activate only if the device population changes, which is a time-consuming operation.[2] The situation is much different if a hardware failure occurs and immediate action is necessary to prevent service disruption. In such a case, DistrBal quickly finds a new placement for data from the failed devices. Despite the fact that each of Derrick's three sub-algorithms has a different purpose, their structure is similar, as all of them optimize data arrangement through a hill-climbing technique. However, the assurance that each of the algorithms is able to find a solution in a given amount of time is not trivial. Therefore, we present novel techniques that are used to reduce computational complexity while giving guarantees that the outcome meets expectations.

As our experimental evaluation shows, Derrick achieves better results than the state of the art in meeting key data arrangement requirements that are important in most storage systems. Moreover, Derrick can be adapted to additionally meet very specific requirements of a particular storage system, as such being potentially broadly applicable.

The rest of the chapter is organized as follows. Section 6.1 surveys related work. Section 6.2 analyzes requirements on data arrangement in self-managed distributed storage and shows how they are met in Derrick. Section 6.3 describes Derrick's algorithms, and Section 6.4 presents further key details. Section 6.5 evaluates Derrick's implementation for HYDRAstor. Section 6.6 contains formalization of the problem and formal proofs. Section 6.7 concludes.

6.1. Data Arrangement Problems and Solutions

Each distributed storage system, to a varying extent, adjusts its data arrangement methods to meet its needs. Some systems, such as HDFS [275] and Haystack [47], store data based on decisions of central metadata servers, which limits scalability, robustness, and performance. Therefore, in large-scale applications, these systems are often replaced by decentralized solutions. For instance, Tectonic is used in Facebook [243] to provide superior scalability and resource utilization. Since its workloads may require low latency or be I/O-intensive, Tectonic arranges data dynamically to meet the specific performance requirements. Similarly, other large-scale systems, like Windows Azure Storage [59] and Spanner [81], dynamically place and move data to improve resilience and performance.

A more static approach is taken by systems that make placement decisions based on hashes of the data. For many types of workloads, including storing content-addressable blocks in HYDRAstor, such methods are extremely efficient. Consequently, in our work, we focus on this kind of data arrangement.

Some of such systems, notably OpenStack Swift with its Rings [128] and Apache Cassandra [176], are based on consistent hashing [159], a technique that limits the number of data moves when the number of hash buckets (e.g., storage devices) changes. There are many publications improving consistent hashing. In particular, elastic consistent hashing [332] aims to reduce power consumption. Aye et al. [35], in turn, describe how to better data balancing specifically in GlusterFS. Consistent hashing can also be reduced to rendezvous hashing, which is a more generic algorithm suitable for storage systems. For instance, IBM Cloud Object Storage System utilizes so-called weighted rendezvous hashing [134].

Another state-of-the-art algorithm is CRUSH [321], employed in Ceph [320], which also distributes data based on a hash-like function. CRUSH supports a multi-level hierarchy of heterogeneous devices and introduces low computational overhead. Data movements in

[2] Device addition requires unpacking, connecting cables, moving data, etc.

CRUSH have been reduced with the introduction of the straw2 [72] bucket type. Furthermore, since the arrangements calculated by CRUSH may underutilize capacity, Ceph features an additional balancer plugin [69] that alleviates this phenomenon. Another sample improvement of CRUSH is MapX [317], which calculates intermediate data placements to decrease the tail latency during data movements.

Both Swift and Ceph are utilized in thousands of deployments and are actively developed, so their algorithms are constantly improved to meet the needs of those systems. However, in the case of our system, aimed at self-managed continuous scalability, CRUSH and consistent hashing do not address some principal requirements, and their adequate modification seems very difficult, if not impossible. For this reason, we have devised Derrick, an alternative approach that is easier to extend to take into account additional constraints. Moreover, for common requirements of storage systems, Derrick also outperforms the state of the art. The requirements are described in Section 6.2 along with a brief comparison of Derrick, CRUSH, and consistent hashing in real-world systems.

The main technique underlying Derrick is hill climbing. It has already been applied to problems related to data balancing, for instance, allocating resources to tasks in a distributed system [90] or allocating data in a system built of devices with varying reliability [91]. The novelty of Derrick, however, comes not from the fact that it is based on hill climbing but rather from the way it utilizes this technique. Especially, without the separation of the aforementioned sub-algorithms and introduction of additional solutions, providing expected results in given time bounds would be hard, if possible at all.

In general, there has been a considerable amount of research on optimizations of the same metrics as in this chapter but using techniques other than finding a placement for erasure-coded or replicated fragments. To start with, there are various erasure coding schemes aiming to improve system performance [197], decrease repair degree [132], or minimize repair bandwidth [157]. Furthermore, there are techniques for dynamic replica management, such as CDRM [319], including adjusting the number of replicas to availability requirements. Another approach, adopted in HeART [155], is to decrease storage utilization by leveraging the fact that disk reliability changes over time. Pacemaker [154] and Tiger [153] further improve this approach by reducing the transition overload, providing further space savings, and bettering robustness.

Because of its design, and notably, the score dimensions introduced shortly, Derrick is prepared to support many of those requirements at the same time. Therefore, by and large, it can be adjusted to incorporate most of the aforementioned ideas. Moreover, integrating many of them into Derrick entails no changes at all. For instance, the LRCs [132] require storing additional erasure coding fragments, for which Derrick can find a proper placement out of the box. Likewise, Pacemaker [154] is meant to keep data in subclusters with homogeneous failure models, and Derrick can find placement within such subclusters as well. Methods requiring dynamic changes in data resilience can also be integrated, either by changing the resilience without modifying the fragment number (e.g., from 9+3 to 10+2 codes) or by modifying the score dimensions. In short, Derrick is truly versatile.

6.2. Requirements on Data Balancing

A recurring theme in on-premise storage systems, including Ceph, Swift, and HYDRAstor, can be summarized as follows. Every data piece (e.g., a file, an object, or a block) has its identifier (e.g., based on its pseudo-random hash). Since managing each such piece separately is infeasible, the identifier is used to assign the piece to a logical collection named a *group*. In

other words, groups are a means of aggregating individual data pieces into manageable units.

Furthermore, for resilience, data are replicated or erasure-coded. Therefore, each group consists of *components* (e.g., a group with 3 replicas has 3 components). We denote components as $IdOfGroup : IdInGroup$, so with 2 groups and 3 replicas per group, the components are: 0:0, 0:1, 0:2, 1:0, 1:1, 1:2. The notation would be the same for erasure codes with 3 parts in total per group (e.g., for 2+1 codes, with 2 data parts and 1 parity part).

The data arrangement problem is finding an assignment of components to devices that maximizes metrics entailed by the requirements of the system. A basic sample data arrangement is presented in Fig. 6.1. In the rest of this section, in turn, we analyze specific requirements on data arrangement that are common in scalable real-world storage systems.

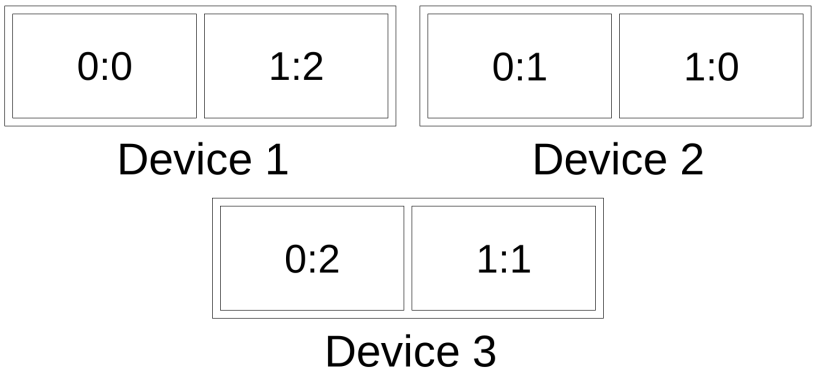


Figure 6.1: Data arrangement of two groups (0,1) with thee components each (0:0, 0:1, 0:2 and 1:0, 1:1, 1:2) on thee same-capacity devices. Capacity utilization and resilience are optimal in such a configuration: no device has more space left than another; a failure of a device affects only a single component in any group.

6.2.1. High Capacity Utilization

The most fundamental requirement on data arrangement is efficient utilization of the available storage capacity. We assume that data are assigned to groups evenly, as the hashing function is fair, so all components in the system have roughly similar sizes. In systems with deduplication, such as HYDRAstor, data are kept in small blocks, so the component sizes are naturally balanced. However, even systems that distribute entire files/objects, like Ceph, do not attempt to address the potential imbalance due to varying file/object sizes [70].

For this reason, we consider capacity utilization as maximized if the ratio of the number of components to the available device storage bytes is equal for all devices (cf. Fig. 6.1). If, in contrast, one device has a higher components-to-bytes ratio, the capacity of devices with lower ratios is wasted (cf. Fig. 6.2), because, per our assumption, all components are expected to have approximately the same size.

When the number of components is increased, the capacity waste may be decreased (as in Fig. 6.2), but this also burdens the system more. Therefore, for instance in Ceph, the recommended number of groups per device is 100 [71], while in HYDRAstor, we keep the number even smaller to increase data locality. If a system encompasses devices with different storage capacities (referred to as a heterogeneous system), the effect of wasted capacity is amplified because the size of each component constitutes a higher percentage of the capacity of a small device (see Fig. 6.3).

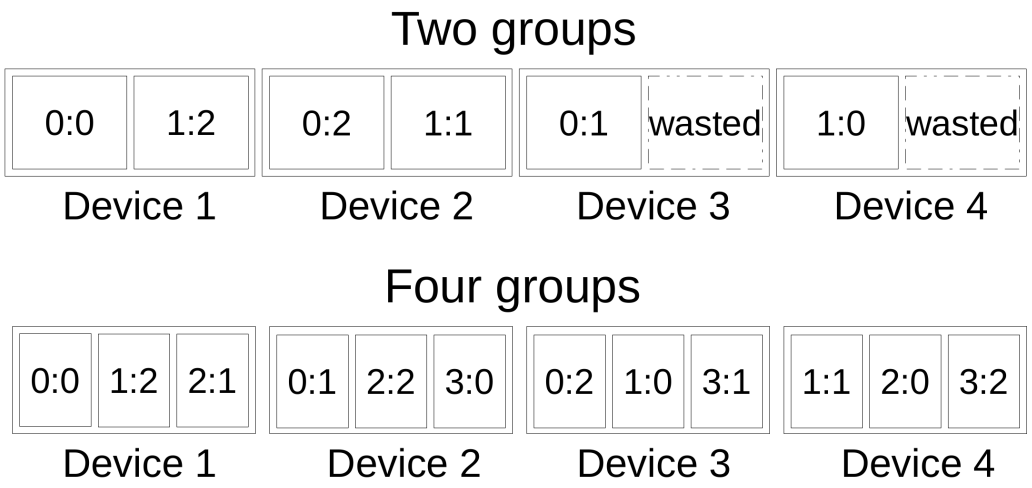


Figure 6.2: Data arrangement of groups with 3 components each. With only 2 groups 25% of capacity is wasted. With the number of groups increased to 4, the capacity is fully utilized.

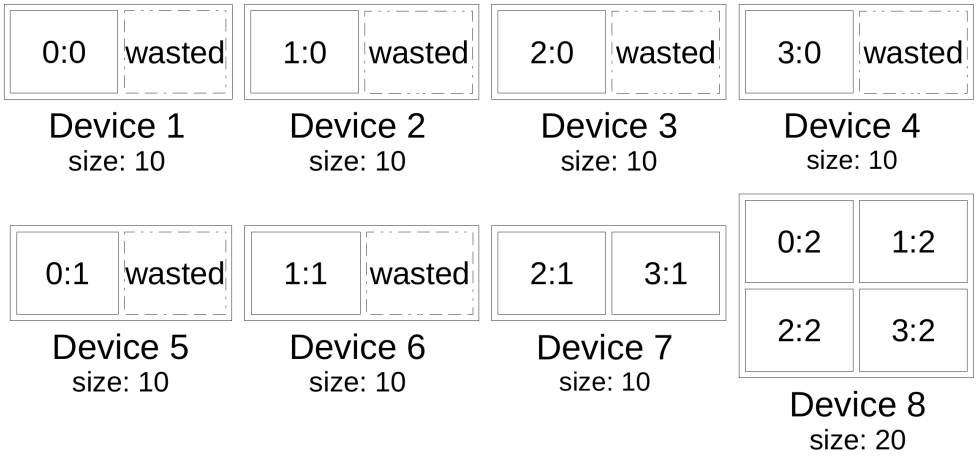


Figure 6.3: Data arrangement of 4 groups with 3 components each in a heterogeneous system. Each device has at most 4 components but, system wise, more than 25% of the capacity is wasted because there are devices with half of their capacity wasted.

6.2.2. Resilience to Failures

Another crucial requirement on data arrangement is storing components resiliently, that is, in a way that, thanks to replication or erasure coding, they can survive device failures. To reduce the probability of data loss in case of a hardware failure, two components of the same group should be kept on different devices. Since devices form a multi-level hierarchy (e.g., a server has many disks, and a rack has many servers), this rule should be followed at the different levels of the hierarchy. The highest attainable resilience depends on the particular replication/erasure-coding scheme and the system size (see Fig. 6.4).

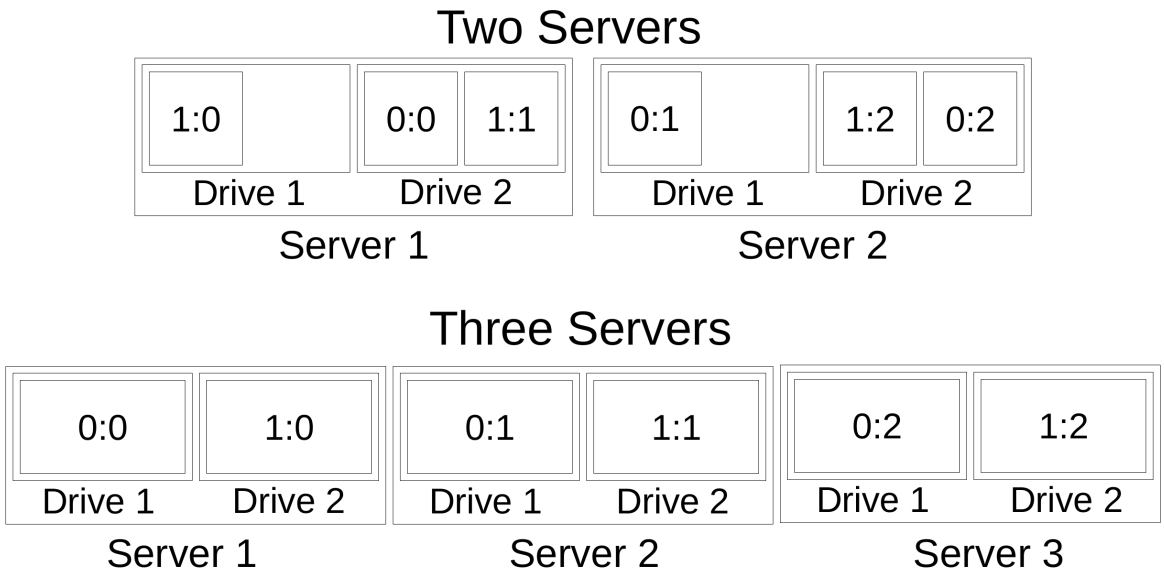


Figure 6.4: The size of a system affects its resilience to failures. With 2+1 erasure codes, a system with 2 servers and 2 disks per server can survive a disk failure but not a server failure. In contrast, a system with 3 servers is also resilient to server failure.

What is important, in heterogeneous systems, optimal capacity and optimal resilience may not be achievable simultaneously (cf. Fig. 6.5). Therefore, a need arises to describe how resiliently the data should be kept. In particular, in Ceph, crushmap rules specify how devices at each level of the hierarchy are chosen. The rules are strictly followed, so crushmap may need an update if the requirements on balancing change (e.g., when the system grows and the number of components of the same group per server can be reduced). In Swift, Rings have a configurable overload factor, which specifies a fraction of additional components that can be accepted by each device to improve system resilience. Unless the factor is high enough, Swift may not find the most resilient solution, so a careful value selection is necessary, and capacity is underutilized anyway (details in Section 6.5.1). To provide self-managed continuous scalability, Derrick always finds the most resilient solution for the maximal capacity. However, it also allows an administrator to specify a minimal level of resilience that overrules the decisions stemming from maximizing capacity utilization.

6.2.3. Balancing Distinguished Components

Data arrangement affects not only resilience and capacity but also the overall system performance. In heterogeneous systems, the performance may be improved by placing more components on faster devices (e.g., servers with SSDs or better CPUs), but storing too many components on one device leads to underutilization of the capacity of other devices. Therefore, in some systems, there are distinguished components (*DistComps*) that have additional tasks assigned. In particular, Ceph and Ursa [181] specify 1 distinguished *primary* per group, while in HYDRAstor there are 3. Overloading any device with too many DistComps should be avoided; otherwise, a bottleneck may arise. Network utilization also depends on data arrangement, as we describe afterwards.

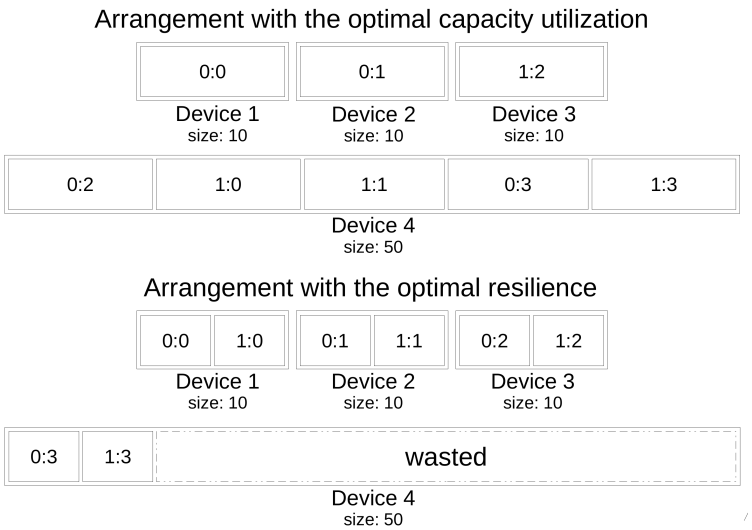


Figure 6.5: A heterogeneous system with four devices: Device 4 is five times larger than Devices 1–3. With 3+1 erasure codes, the arrangement resilient to a device failure wastes 50% of the capacity. In contrast, in the arrangement with an optimal capacity utilization, Device 4 hosts 3 components from group 1, which precludes recovery of this group upon a failure of this device.

6.2.4. Keeping Related Data in One Rack

With erasure codes, decoding multiple pieces is required to reconstruct failed data. Consequently, placing whole groups within the same rack decreases the expensive inter-rack communication during recovery. At the same time, keeping a group in one rack precludes resilience to rack failures, which is provided by many systems, including Ceph, Swift, and clouds [132]. However, some studies suggest that rack failures are less frequent than expected, and the overall system resilience is higher if data are located nearby to improve reconstruction speed [341]. Therefore, HYDRAstor gives this possibility as well. Ceph provides an option to specify a rule that keeps whole groups within one rack, but its balancer plugin tends to move components between racks more than necessary. Likewise, HDFS has a default policy to place two replicas in one rack and the third replica in another, which gives some locality and also resilience to a failure of one rack. There are also special erasure codes that trade capacity for decreased inter-rack traffic [129], but this conflicts with capacity maximization.

6.2.5. Limiting Data Movements

The data arrangement algorithm needs not only to calculate a good result for a stable system but also to react to changes that are often unexpected (e.g., hardware failures) or predictable shortly in advance (e.g., additions and removals of devices). Modification of a data arrangement requires moving data between devices, and hence it is desirable to minimize such movement as much as possible. In scale-out systems that can significantly change their size (e.g., from one to hundreds of devices), a change in the number of groups is necessary to maintain a similar number of components per device regardless of the system size. Ceph and HYDRAstor scale the number of groups automatically [71], while in Swift, the functionality for altering the number of groups without cluster downtime is under development [239]. When the number of groups changes, some components should be moved (e.g., to improve capacity utilization as in Fig. 6.2). However, if the algorithm computes a placement for new groups

independently of the previous placement of data (e.g., as in CRUSH), excessive amounts of data may be moved. A similar issue happens when other system parameters change (e.g., the configuration of the resilience hierarchy).

6.2.6. Limiting Non-stable Components

Another requirement related to data movement is how many components are moved at the same time (we refer to components during movement as *non-stable*). In Swift, only a single component from a group is moved at a time because the system cannot read data from non-stable components. HYDRAstor can read data from non-stable components, but it benefits from a limited number of such components in duplicate elimination, caching, and read-write deletion [283]. Ceph provides throttling, which also limits data moved at a time but does not limit movements per group. Keeping groups within a single rack makes maintaining components stable more difficult as it enforces the movements of entire groups (see Fig. 6.6).

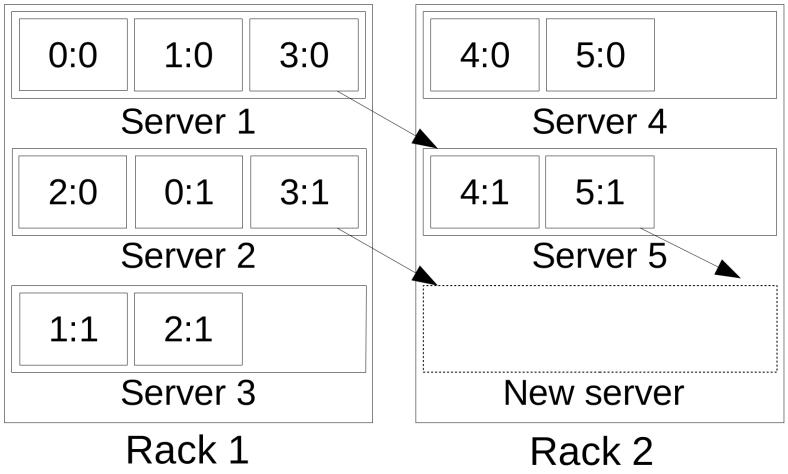


Figure 6.6: Keeping entire groups in racks increases the number of required component movements upon a change to a system. In a system with 2 racks, 1 new server is added to the 5 existing ones. In effect, 2 components have to be moved to balance the capacity, and a movement of an entire group is needed to keep entire groups in racks. More specifically, Group 3 is moved. However, its two components cannot be stored on the new server because of the resilience requirements. Therefore, component 5:1 is moved as well to create space on Server 5 for component 3:0. All in all, 3 components are moved instead of just 2.

6.2.7. Final Remarks

In Table 6.1, we summarize the discussed requirements and the way real-world implementations of the aforementioned state-of-the-art data arrangement algorithms meet them. As visible in the table, Derrick trades computation time when devices are added or removed for additional features and better results. As mentioned earlier, device additions and removals are predictable and take significant time anyway, so we find the trade-off profitable. In practice, the additional time spent on computations does not affect the system at all (i.e., the system is fully operational during such calculations), whereas better data placement provides considerable benefits.

	Derrick	CRUSH in Ceph	Swift Rings
Capacity utilization	**The Highest**	High (with Ceph's balancer)	High
Resilience with a multi-level hierarchy	**Optimal solution for given constraints and capacity**	Only given constraints are met	Space oversubscription needs to be configured
Balancing of distinguished components	**The Best**	Moderate	None
Groups can be kept within one rack	**Yes**	Yes, but Ceph's balancer spoils it	No
Data movement	**The Lowest**	Moderate	Low
Limiting non-stable components	**Precise** (preserve resilience and capacity)	None	Simplified ("move one at a time")
Computation time during failures	**Seconds**	**Seconds**	**Seconds**
Computation time when adding or removing devices	Minutes or hours	**Seconds**	**Seconds**

Table 6.1: Summary of data arrangement algorithms in real-world use cases.

6.3. Derrick's Overview

To meet all of the aforementioned requirements and provide self-managed continuous scalability, Derrick arranges data using three sub-algorithms. As we describe in this section, CentrBal, and TrGuide calculate placement for every component in the system, whereas DistrBal modifies their results in case of failures.

First, data arrangement for a healthy state is calculated using *CentrBal*, which is allowed to operate for minutes or even hours until a satisfactory solution is found. During this step, capacity utilization, resilience, balancing of distinguished components, placement of data between racks, and data movement is optimized. The calculation uses a small fraction of available resources (i.e., at most one core in a multi-server environment), and in practice has no negative impact on service quality, as our system has implemented mechanisms for effective resource sharing during various loads [278].

As the new data arrangement can differ a lot from the previous one, a mechanism is needed to prevent having too many non-stable components. Therefore, *TrGuide* orchestrates efficient component movement between two CentrBal results that preserve the stability of components and other requirements (including capacity utilization, resilience, and managing network traffic). In the event of a hardware failure, an immediate change is needed, and in such a case, *DistrBal* quickly overrides CentrBal/TrGuide decisions.

The results of each algorithm are distributed throughout the system to enable routing messages to proper servers. As CentrBal and TrGuide compute a placement for all components in the system, their result can have a significant size (e.g., multiple megabytes if there are millions of components), but the algorithms are executed occasionally. DistrBal modifies locations only for components affected by a failure, so its results are much smaller.

6.3.1. Hill Climbing in Derrick

All three subalgorithms search the space of possible arrangements, which is exponential in system size. Therefore, a heuristic approach is taken to find a solution that meets many requirements at once. To be more specific, Derrick uses the hill-climbing method, which is an optimization technique that iteratively attempts to find a better solution by making incremental changes. To achieve this, each subalgorithm calculates a multi-dimensional score function that describes how much a component arrangement fails to meet the given requirements. In each iterative step, one or more components are moved at a time, using some *operation*, as we

explain shortly. If moving the components decreases the score, the operation is applied and the procedure is repeated for a better arrangement.

Since the three subalgorithms have different goals, their score functions differ. To be more specific, each score function describes component placement as a collection of *score dimensions* (called ScoreDims). ScoreDims are compared in a lexicographic order, and each ScoreDim corresponds to a single requirement on data placement. For instance, to describe capacity utilization, a ScoreDim can contain sorted quotients of device size and a number of components assigned to it. In a situation from Fig. 6.1, assuming each device has 1 TB, the set is $\{0.5, 0.5, 0.5\}$ and if one component were moved, the set changes to $\{0.33, 0.5, 1.0\}$ (in which capacity utilization is worse, as the component size is reduced from 0.5 TB to 0.33 TB). Another example is a ScoreDim which describes resilience by counting components from the same group on each device. Such a ScoreDim contains a cartesian product of all groups and devices. For instance, in a situation from Fig. 6.1 it is $\{1, 1, 1, 1, 1, 1\}$ (each of the three devices has one component from each of the two groups), and if a component were moved, it would be $\{2, 1, 1, 1, 1, 0\}$ (one of the devices would host 2 and one of the devices would host 0 components from one of the groups). The score function orders ScoreDims by their priority. Therefore, if a more important ScoreDim has a higher value the score is worse, even if a lower-priority ScoreDims improve. In other words, a more important requirement is never violated to improve a less critical one.

To improve the score, components are moved between devices based on heuristics dubbed operations. The very basic operation is to try the movement of each component to another device and verify if any of such movements improves the score. Such an operation is not sufficient to reach an optimal score. For instance, to reach optimal resilience without decreasing the capacity utilization ScoreDim, movement of two components at a time may be necessary (see Fig. 6.7). However, trying all possible component movements is $O(mn)$ with m components and n devices, whereas trying all possible movements of two components at a time is $O(m^2n^2)$, and three is $O(m^3n^3)$. As we explain in further sections, there are cases in which three or more components must be moved at the same time to leave a local minimum. On the one hand, staying in a local minimum means that one of the ScoreDims is not improved as much as possible (e.g., TrGuide can halt the transition). On the other hand, checking all possible movements of three components at a time for a system with $m = 1000$ and $n = 100$ (which are smaller than the maximal configuration we aim to support) requires checking more than 10^{15} states, which is unacceptable. Therefore, we introduce techniques that reduce the set of tried movements (discussed in Section 6.4).

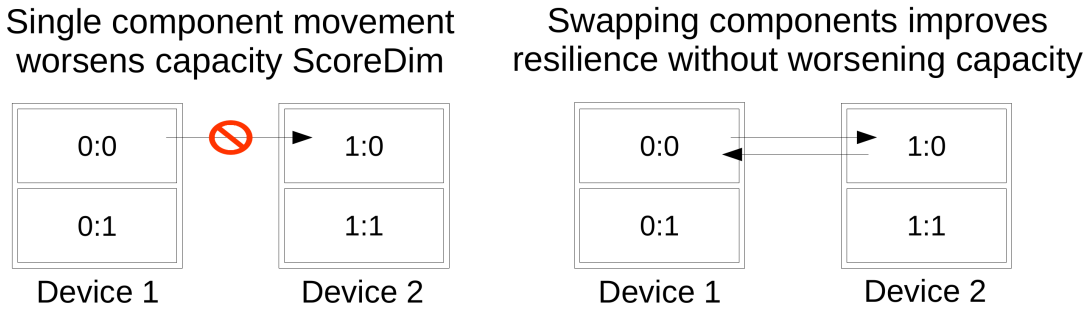


Figure 6.7: An example in which a single movement fixes the resilience but temporarily negatively affects the capacity. However, swapping allows exchanging components without changing capacity usage, therefore such movement can improve resilience without worsening the capacity.

The general idea of subalgorithms is presented in Listing 6.1. The details of each subalgorithm are encapsulated in specific requirements, score calculation, and improvement heuristics.

Listing 6.1: Pseudocode of Derrick subalgorithm

```
# devices - List of devices with relevant details
# prev_arr - Previous arrangement of components
# reqs - List of ordered requirements
def derrick_subalg(devices,prev_arr,reqs):
  score, arr = calc_score(prev_arr,reqs), prev_arr
  while True:
    old_score = score
    for req in reqs:
      new_arr = improve(arr,req)
      new_score = calc_score(new_arr,reqs)
      if new_score < score:
        score, arr = new_score, new_arr
        break
    if old_score == score:
      return arr
```

The last remark regarding hill climbing is that, in general, such an optimization technique can find a local minimum that is different from the global minimum [150]. In the case of a component arrangement problem, it means that the arrangement will not meet all requirements, for instance, resilience or capacity can be decreased, which is unacceptable. Therefore, we introduced techniques that guarantee that the result always matches the global minimum for the most important ScoreDims. We explain these techniques in detail in Section 6.4.

6.3.2. Central Balancing

CentrBal computes a component arrangement for a healthy system. Its input contains: a list of all devices with their sizes, a current component arrangement for a healthy system (calculated earlier or, in the case of a fresh system, generated arbitrarily), and an ordered list of requirements. In this section, we explain a simplified CentrBal, which ensures optimal capacity and resilience while minimizing the number of transfers (advanced techniques are discussed in Section 6.4). Three ScoreDims are necessary to meet the aforementioned requirements:

1. **Capacity**: whether a device has too many components for its size.

2. **Resilience loss**: counts components from each group per device.

3. **Movements count**: counts components with a changed placement.

With those ScoreDims, CentrBal tries to find an arrangement that first maximizes capacity, then minimizes the number of components from the same group on one device. If any of the heuristics find a data arrangement that is as good or better that the current one in terms of capacity and resilience but requires fewer component movements, such an arrangement is chosen.

In heterogeneous systems, optimal resilience and capacity may not be possible at the same time (cf. Fig. 6.5), so an additional ScoreDim above *Capacity* is added: *Accepted resilience loss*, which counts components from the same group per device, but only above a given threshold. For instance, if the threshold is 2 the system capacity is optimized as long as no device hosts more than 2 components from the same group. Moreover, the *Resilience loss* ScoreDim optimizes the resilience as much as possible without capacity decrease, so if a

solution with the optimal capacity and at most 1 component from the same group on each device exists, it will be found.

Distributed systems typically have a device hierarchy (disks, servers, racks, etc.), so both resilience and capacity can have multiple ScoreDims that represent each device type. Disk utilization is a more important ScoreDim than the utilization of servers, as disk utilization directly affects the system capacity. However, the arrangement of components at other levels of the hierarchy can also provide benefits (e.g., a better utilization of network links). The ordering of such dimensions is important because, for instance, two utilization ScoreDims can be conflicting (see an example in Table 6.2).

	Server 1		Server 2	
	Disk 1	Disk 2	Disk 3	Disk 4
Disk size [TB]	42.9	42.9	31.25	31.25
Total server size [TB]	85.8		62.5	
Case 1: A component is added to Server 2				
Disk components	19	18	14	**13 + 1**
Machine utilization ScoreDim	2.319		2.232	
Disk utilization ScoreDim	2.258	2.383	2.232	2.232
Case 2: A component is added to Server 1				
Disk components	19	**18 + 1**	14	13
Machine utilization ScoreDim	2.258		2.315	
Disk utilization ScoreDim	2.258	2.258	2.232	2.404

Table 6.2: Utilization requirements (e.g., for disks and servers) can be conflicting. In Case 1, the disk utilization is lower than in Case 2 as $\{2.383, 2.258, 2.232, 2.232\} < \{2.404, 2.258, 2.258, 2.232\}$, but for machine utilization $\{2.319, 2.232\} > \{2.315, 2.258\}$.

6.3.3. Transition Guide

TrGuide controls component movements from the current system state to the arrangement calculated by CentrBal. It ensures that system requirements are met, and network usage is balanced. Its major challenge is to keep many components stable. Just like CentrBal, TrGuide computes the score for the entire system. ScoreDims that describe the system resilience and capacity are the most important. After that, there are ScoreDims that count how many components are moved: within each group (it determines the number of non-stable components) and from each device (to balance network usage).

TrGuide needs the motivation to move components to the location calculated by CentrBal. Therefore, *ComponentsNotOnTarget* is a ScoreDim which counts components not located in their final location. *ComponentsNotOnTarget* is less important than the ScoreDim counting components moved within each group to prevent a situation in which all components are moved at once. It is expected that after a portion of component movements TrGuide will stop because every possible move will violate ScoreDims more important than *ComponentsNotOnTarget*. Later, when some movements are completed (and the number of non-stable components is decreased), TrGuide will move the next portion of components. Assuring TrGuide's liveness without moving too many components at a time is the difficult part (explained in Section 6.4.4).

6.3.4. Distributed Balancing

DistrBal overrides CentrBal and TrGuide decisions in case of hardware failures. Its goal is to find a good temporary placement for components that lost their devices. Therefore, DistrBal moves components from failed devices but does not move components hosted on their healthy

CentrBal/TrGuide targets. This is because the movement of healthy data costs resources (e.g., disk I/Os, network traffic), which are needed to reconstruct missing data and handle the additional load. The components return to the location calculated by CentrBal/TrGuide when the issue is solved, so transient failures are handled efficiently.

Unlike the other two algorithms, DistrBal improves the placement only for a subset of components and devices, and the responsibility for calculating the arrangement is distributed across many instances of the algorithm. Each device has its own DistrBal instance that considers moving components to other devices. Limiting the responsibility of a single DistrBal instance decreases its complexity and facilitates the gathering of volatile information (e.g., a current utilization or a failure state of devices). If a device fails, its components are moved by other DistrBal instances that host components from the same groups. Since many DistrBal instances can make decisions about the same component, synchronization is needed. For example, DistrBal instances that host components from the same group can conduct voting to move only one component from their group at a time. To prevent a situation in which many components are simultaneously moved to one device, a locking mechanism is implemented (e.g., the device allows only one new component at a time).

6.4. Derrick's Details

As described in the previous section, Derrick's subalgorithms are based on a simple idea to improve score functions by moving components. However, the selection of proper ScoreDims and heuristics is non trivial. In this section, we describe details of important techniques used in Derrick for HYDRAstor. Both theoretical lemmas, with their key ideas and practical observations, are presented to explain that even very detailed requirements can be met to provide self-managed continuous scalability. The methods are general and can be used in all subalgorithms, not only the subalgorithm chosen as an example to clarify each technique.

We use the following names of operations that move components to explain the techniques and heuristics:

Relocation(c, n) moves component c to device n.

Swap(*c_1*, *c_2*) swaps c_1 and c_2 between their devices.

Push(*c_1*, *c_2*, n) moves c_1 to device of c_2 and c_2 to device n.

Cycle3(*c_1*, *c_2*, *c_3*) moves c_1 to device of c_2, c_2 to device of c_3 and c_3 to device of c_1.

6.4.1. Capacity and Resilience in CentrBal

First, we show how the computational complexity of Derrick can be limited when heuristics are selected properly. Trying all possible relocations is not enough to find a data arrangement that has the best capacity for a given resilience target because the only possible relocation that improves the capacity may decrease resilience. Therefore, movement of more than one component at once may be necessary (see Fig. 6.8). In our system, the groups have equal sizes (typically 12, so erasure codes like 9+3 and 10+2 are possible), and therefore we formalized a *Lemma 1* that limits the number of operations tried at once to improve resilience and capacity.

Lemma 1 *If groups are equinumerous (in terms of components), then trying all relocations, swaps, and pushes is sufficient to find an arrangement with optimal capacity within resilience restriction.*

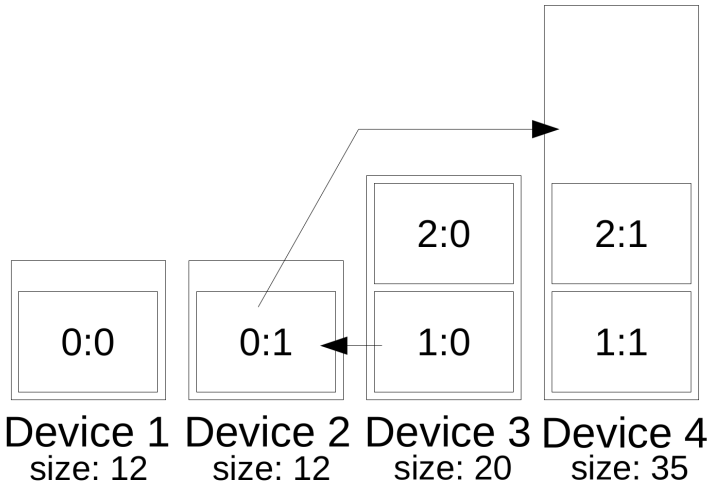


Figure 6.8: Relocations are insufficient to balance capacity with a resilience restriction. Initially, the maximal size of each component is 10, because there are 2 components on Device 3 with capacity 20. None of the possible single relocations increases the score, as they either decrease the resilience or decrease usable capacity. After a single push, the size of the component can be increased to 11.67 (limited by Device 4), so the capacity increases by 16.7%.

The key idea of the lemma is that if the system is not balanced, there must be a device, n_1, that accepts a component, c_1, from a group, g_1, and a device, n_2, that has too many components. However, because of the resilience restriction, there is a possibility that none of the components from n_2 can be accepted on n_1 (and cannot be done). As the groups are equinumerous, in such case there must be a third device, n_3, which accepts a component from n_2 and has a component from group g_1. The formalized version of the lemma with a proof is presented in Chapter 6.6. The situation is much different if groups can have different sizes (cf. Fig. 6.9).

6.4.2. Multiple ScoreDims in CentrBal

CentrBal optimizes capacity and resilience but also tries to meet all other requirements of a specific system. In our case, the device hierarchy consists of three levels: racks, physical servers, and logical servers (which host a fixed number of disks). Therefore, to meet all of the requirements described in Section 6.2, our CentrBal score function consists of over 20 ScoreDims. The most important 12 of them are explained in Table 6.3, but there are some other system-specific ScoreDims which, for example, balance DistComps even further. However, the exact selection and order of ScoreDims depend on a specific system design. In our case, we find the optimization of system capacity very important for storing backups, but we can think of a theoretical system in which optimization of DistComps (and therefore throughput) would be more important than the maximal utilization of disk space.

Having so many dimensions enables optimizing resources effectively. For instance, DistComps are first optimized across logical servers, as each logical servers can handle a similar number of I/Os per second. After that, DistComps are also optimized across physical servers. After ScoreDim #8 is optimized, a possible situation is that one physical server hosts logical servers with 7 DistComps each, and the other server has 8 DistComps per logical server. In the described situation, CentrBal would try to move DistComps to further optimize the utilization of resources shared between the servers, so each of the physical servers has 15

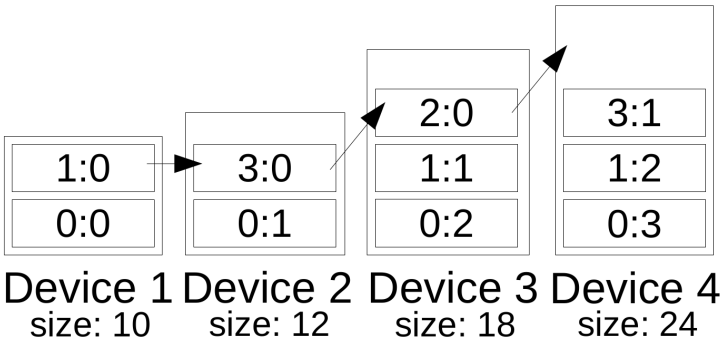


Figure 6.9: If the groups have a different number of components (group 0 – 4, group 1 – 3, group 2 – 1, and group 3 – 2 in the figure), more than two simultaneous operations can be required to balance capacity with a resilience restriction. Device 4 has enough free space to get an additional component, but cannot host any additional component from Device 1 or Device 2 without decreasing resilience. Relocating component 2:0 to Device 4 without performing other operations does not improve the score, so it is not done alone.

DistComps in total. However, having that many ScoreDims also has significant consequences as we explain next.

First of all, the arrangement that optimizes all ScoreDims typically does not exist, and more important ScoreDims impact less important ScoreDims in a non obvious way. For instance, the ScoreDim for keeping entire groups in racks can spoil the balance of DistComps (Fig. 6.10), which may be somewhat of a surprise. Second, the calculation of the entire score function for all dimensions is expensive, as each score requires its data structure of a significant size (e.g., it keeps information per device). Therefore, we implemented auxiliary data structures that are sufficient to verify how each ScoreDim changes after a single operation. Only when the best operation is selected, the full score is recalculated.

Finally, sometimes to improve a ScoreDim without spoiling another one, many operations need to be done at once (see Fig. 6.11), which increases the complexity. Therefore, a possible solution to reduce the complexity is to greedily select an operation when it improves the score, without checking if there are any better options, but then an operation that also improves less important ScoreDims can be missed. Therefore, in our implementation, we often try a few improvements and choose the best. Another technique is to limit the components and devices that are considered for each heuristic. For example, when the resilience is not optimal, the component movement can be initially limited to components from devices that host most components from one group. Only when all heuristics related to system resilience have been tried, are heuristics optimizing other ScoreDims started.

6.4.3. DistrBal ScoreDims

DistrBal tries to find a placement for components from failed devices, which is as good as the placement that would be computed by CentrBal. In fact, DistrBal score consists of very similar ScoreDims as these of CentrBal. However, the time constraints for the computing time of both algorithms are different. Therefore, the first difference is that the set of heuristics used by DistrBal is limited to the least complicated ones. Moreover, the set of components that are even tried to be moved is limited only to components from the failed devices.

As DistrBal communicates with other devices hosting components from a group, it has access to additional information, like the very recent capacity utilization of each device. There-

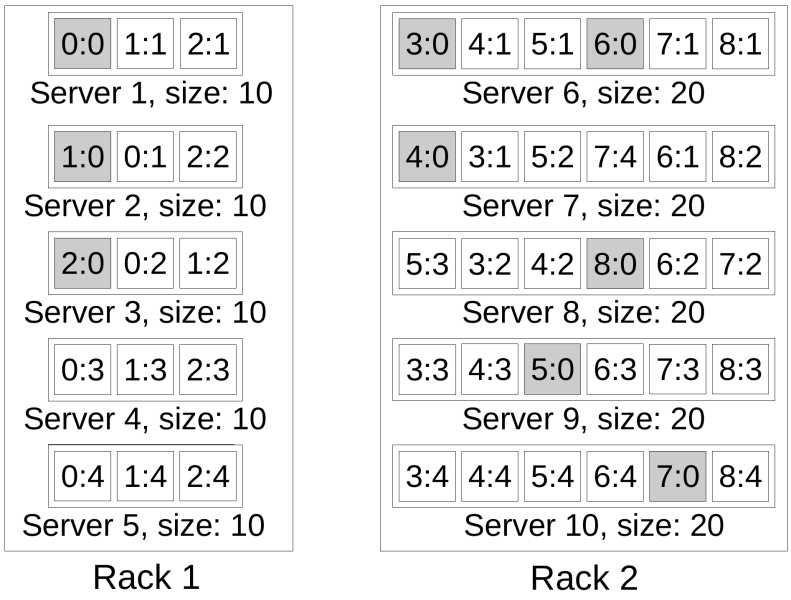


Figure 6.10: An example in which placing whole groups in racks spoils the balance of DistComps. Components with IdInGroup=0 are distinguished. Rack 2 has servers with larger devices, so it receives more groups (3,4,5,6,7, and 8) than Rack 1 (0,1,2). As the result, 6 DistComps need to be placed on 5 servers, and hence in this case Server 6 ens up with 2 DistComps.

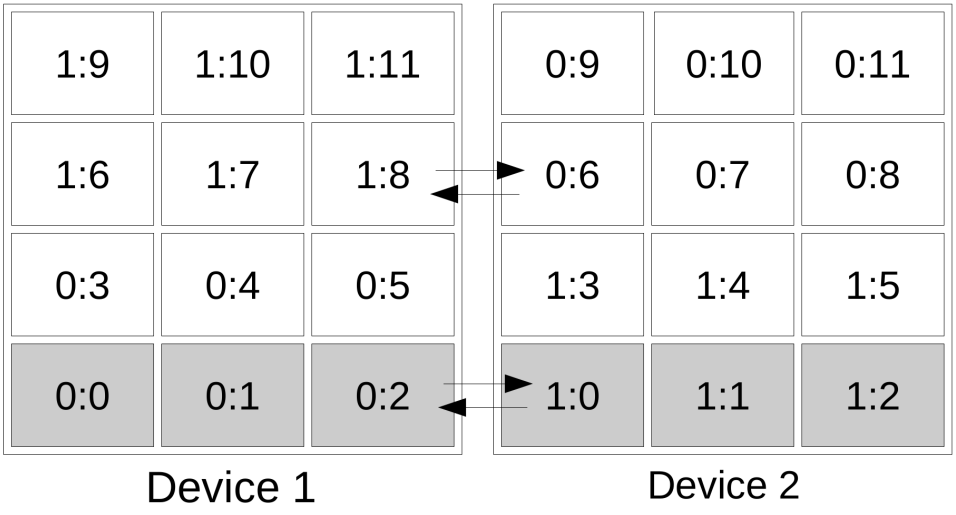


Figure 6.11: An example in which two simultaneous swaps are required to limit the number of DistComps of the same group on one device. In that example, components with IdInGroup=0,1,2 are distinguished. Operations that move fewer components would decrease resilience, capacity, or the number of DistComps per device.

#	Score Dimension	Description
1	Accepted resilience loss	Ensures target resiliency in heterogeneous systems
2	System capacity	Optimizes system capacity
3	Resilience (logical server)	Decreases the number of components from the same group on each logical server.
4	Resilience (physical server)	Decreases the number of components from the same group on each server.
5	Num. components to the size of physical server	Balances components per physical server for even consumption of server resources.
6	Num. components to the size of logical server	Balances components per physical server for even consumption of logical server resources.
7	Keeping groups within a single rack	Reduces the number of racks among which each group is spread.
8	DistComps distribution across logical server	Balances distribution of DistComps across different levels of the hierarchy to balance unequal resource consumption.
9	DistComps distribution across physical servers	
10	DistComps distribution across racks	
11	DistComps from the same group (logical server)	Decreases the number of DistComps from the same group on one logical server.
12	Number of transfers	Decreases the number of transfers required to optimize all of the above.

Table 6.3: Major ScoreDims of CentrBal in HYDRAstor

fore, if a device currently has some additional capacity (because the system has some free space), a component can be placed there. If the device becomes full before the failed device is restored, the recovered component will be moved to a different location, which has free capacity but is worse in terms of other ScoreDims.

6.4.4. Component Stability in TrGuide

The goal of TrGuide is to move all components to their target locations calculated by CentrBal while keeping requirements on data arrangement. In HYDRAstor one of the requirements is to not exceed 3 non stable components within each group. Therefore, a transition plan is generated that does not move more than 3 components from each group at a time. After the maximal possible number of components is moved, TrGuide waits until any of the components is fully transferred (and therefore stable again). The operation is repeated until all of the components reach their final locations provided by CentrBal.

The transition plan is prepared based on the following TrGuide_Score:

TrGuide_Score

1. Servers with too many components (exceeding the capacity)
2. Groups with decreased resilience
3. Groups with more than 3 non-stable components
4. Components not in their target locations

TrGuide is capable of moving all components to their targets, without violating resilience and capacity restrictions, while moving at most three components of one group at a time. Therefore, it is guaranteed that the algorithm can progress without making more than 3 non-stable components within each group. The fact that TrGuide can move forward by moving 3 components also limits its computational complexity, as fewer component movements need to be tried at a time.

The key idea of why moving only three components at a time is sufficient is based on a construction of a component arrangement in which *relocations*, *swaps*, and $cycle_3$ violate resilience restrictions, as *swaps* and $cycle_3$ do not change the capacity. Such an arrangement must contain a cycle longer than 3 but we found two ways of effectively breaking such a long cycle into a smaller one (see Fig. 6.12). One of the methods requires keeping a reserve for one additional component on each server, which decreases the system capacity. Therefore, we introduced another method that breaks the long cycle into parts. Such breaking of the cycle is always possible (an example is presented in Fig. 6.13), as we explained in the proof of Lemma 2, Lemma 3, and Lemma 4 in Section 6.6.

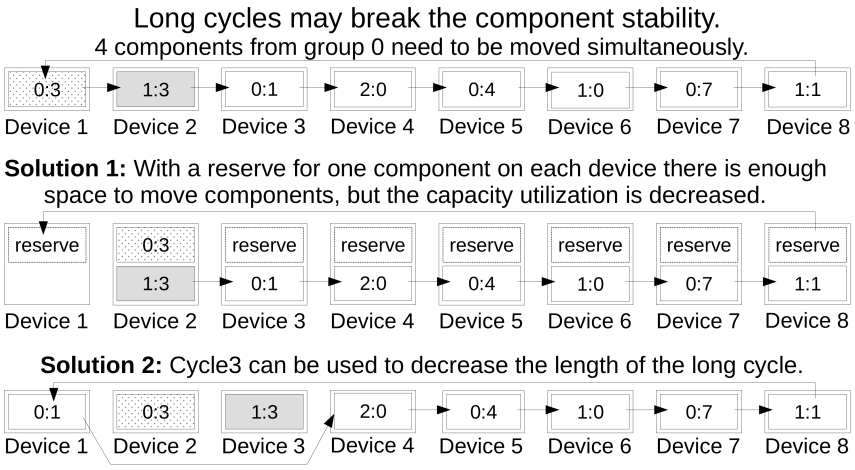


Figure 6.12: Moving longer cycles at once can violate the stability of components. A possible solution is to keep an additional reserve for one component on each device, but it decreases system capacity. Therefore, we break longer cycles into smaller ones with a different technique.

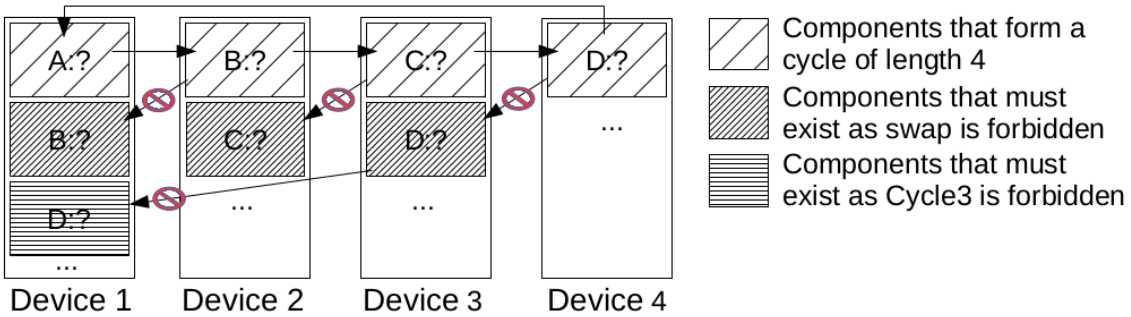


Figure 6.13: If a cycle of length 4 can be done without worsening resilience or capacity, then a *swap* or *cycle3* can be done; otherwise, the Device 1 does not accept a component *D:?* from the Device 4 which is a contradiction, as a longer cycle is assumed to be possible.

6.4.5. Stability of DistComps in TrGuide

An extreme example of meeting requirements by Derrick is maximizing the system performance by ensuring that at most one (out of three) DistComps is non-stable in each group. TrGuide is able to ensure such requirements. However, additional ScoreDims are needed to make TrGuide progressing. This is because there is a difficult case in which two DistComps

from the same group need to be swapped. Such a swap cannot be done without making both components unstable. Therefore, TrGuide needs to make three swap operations with non-distinguished components (as presented in Fig. 6.14). To enforce such an operation, an additional *Unwanted Distinguished Components* dimension was needed, that counts DistComps that are placed on the device that according to CentrBal will host another DistComp from the same group. In this way, TrGuide has the motivation to swap the DistComp with a non-distinguished component to improve that ScoreDim. Additionally, *Components not on the final target of components from its group* ScoreDim, that counts components that occupy place for a different component from the same group, enforces that the swap is done with another component of the same group.

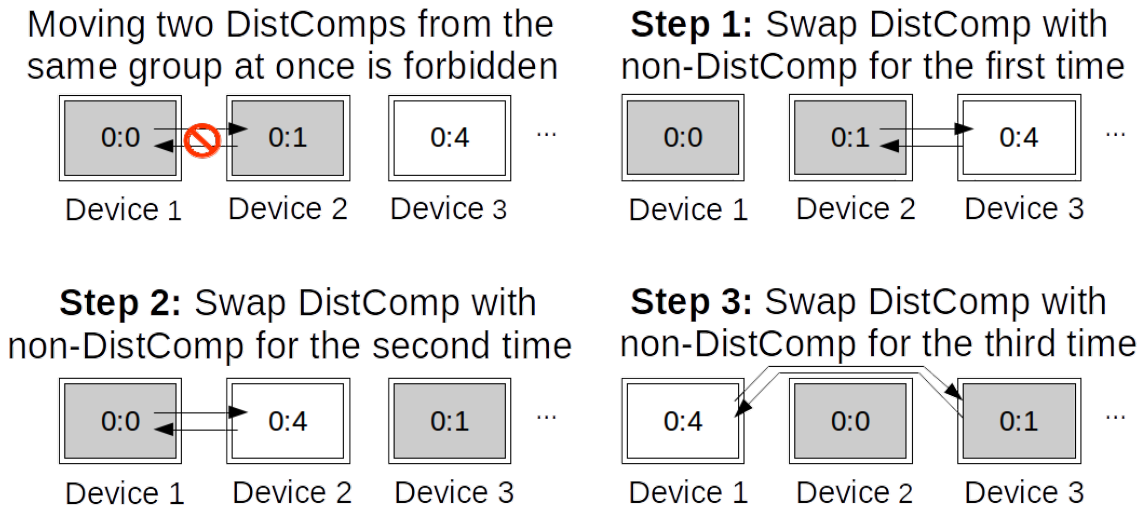


Figure 6.14: Additional operations are needed to swap two DistComps from the same group without making both of them non-stable at the same time.

To sum up, the final score function for TrGuide looks as follows:

TrGuide_Score_Extended

1. Servers with too many components (i.e., too much data)
2. Groups with decreased resilience
3. Groups with more than one non-stable distinguished component
4. Groups with more than 3 non-stable components
5. Components not on the final target of components from its group
6. Unwanted Distinguished Components
7. Distinguished Components not in their final target
8. Components not in their target location

6.4.6. Final Remarks

Derrick is flexible and, as explained hitherto, it can meet diverse system requirements. Using our techniques, we were able to express every needed requirement in adequate ScoreDims. Typically, an addition of a single straightforward ScoreDim is enough, but in some cases, a more complex design needs to be used (as in our TrGuide). Similarly, the basic operations or their compositions are typically enough to implement heuristics that find data arrangement meeting each requirement. We evaluate the performance of Derrick in the next section.

6.5. Evaluation

The evaluation is divided into three parts. First, Derrick is compared with the state-of-the-art algorithms used in Ceph and Swift in terms of meeting the requirements from Section 6.2. Second, we analyze the differences between DistrBal and CentrBal. Lastly, we measure the computation time of our implementation.

6.5.1. Comparison with Ceph and Swift

To ensure self-managed continuous scalability, a system needs to meet many requirements on data placement, so we compare data arrangements generated by Derrick, CRUSH in Ceph, and Swift Rings. We conducted a series of experiments to show how the algorithms differ in optimizing capacity and resilience, limiting the number of transfers required after changes in the system, keeping groups within racks, and balancing DistComps.

In most experiments, we used average-sized, heterogeneous configurations, which are typical for on-premise storage. They are additionally easy to understand. However, in the relevant cases, we present the results from larger configurations to show how the algorithms behave during scaling. Many experiments used a variable number of groups, as the number often affects the results. The number of components per group in the presented experiments was 12, which is the default value for both the erasure-code scheme in Ceph documentation and for HYDRAstor, but we did not observe any meaningful differences between experiments with 2–15 components per group. In our system, each logical device internally manages its disks in a manner similar to RAID0, while the other systems assign components per disk. Our approach improves capacity utilization when the number of components does not equally divide the number of disks. However, we decided to ignore this difference in the presented results to avoid favoring our system.

The first two experiments were conducted using actual multi-server installations and real data being written. In these experiments, we evaluated the three algorithms in terms of integration with the entire system and its practical behavior. In further experiments, we only used tools that calculate the placement of the components offline, as it saved us from setting up a new testbed for each experiment. The tools that can compute component placement are already provided with Ceph (*osdmaptool*) and Swift (*swift-ring-builder*), and we implemented a similar tool for Derrick. All three tools share code with the production systems.

In all of the experiments involving Ceph, CRUSH was configured to use the *straw2* bucket type, which minimizes component movement. In relevant cases, the CRUSH result was further improved by Ceph's balancer (noted as *CRUSH_bal*). Swift required modification of the *overload* parameter in some experiments (noted as *SWIFT_overl*).

Deployed Systems Evaluation

In the first experiment, we used fully deployed systems to compare how each system is balanced initially and how failures are handled by each algorithm. We built one of the smallest possible heterogeneous configurations capable of storing data resiliently with 9+3 codes and up to three device failures. Therefore, the system consisted of 16 servers with the following configuration. Each of the servers had two disks formatted to have equal sizes. In 14 of the servers, the devices had 10 GB (denoted *largeServer*), and in 2 the devices had 5 GB *smallServer*, as Ceph does not allow devices below 5 GB. We intentionally decreased the size of HDDs, as conducting the same experiments with original, multi-terabyte disk size would increase time of each experiments to hundreds of hours, whereas from the perspective of the balancing algorithm only the ratio between device sizes is important (what we confirmed by conducting selected experiments with larger disk sizes). The system hosted 384 groups. The servers used Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz and Intel(R) Xeon(R) CPU E5-2430 @ 2.20GHz.

In the beginning, the systems had no data, and all of the servers were up and running. Therefore, Derrick placed all components just as computed by CentrBal, as there was no need to make any adjustments with DistrBal. Similarly, Ceph's balancer and Swift Rings placed components for the stable system. Already at this stage, the difference is considerable, as each algorithm placed a different number of components on small servers (Fig. 6.15). On devices of small servers (5 GB), Derrick placed at most 6 components, Swift placed 7 and Ceph's balancer placed 8. On devices of large servers (10 GB), each algorithm allowed at most 13 components. Therefore, according to the placement, the maximal component size for Swift is 5/7 GB, for Ceph's balancer 5/8 GB, and for Derrick 10/13 GB (as $10/13 < 5/6$). We elaborate on the utilization of capacity by Swift and Ceph in Section 6.5.1, as differences occur in other cases as well.

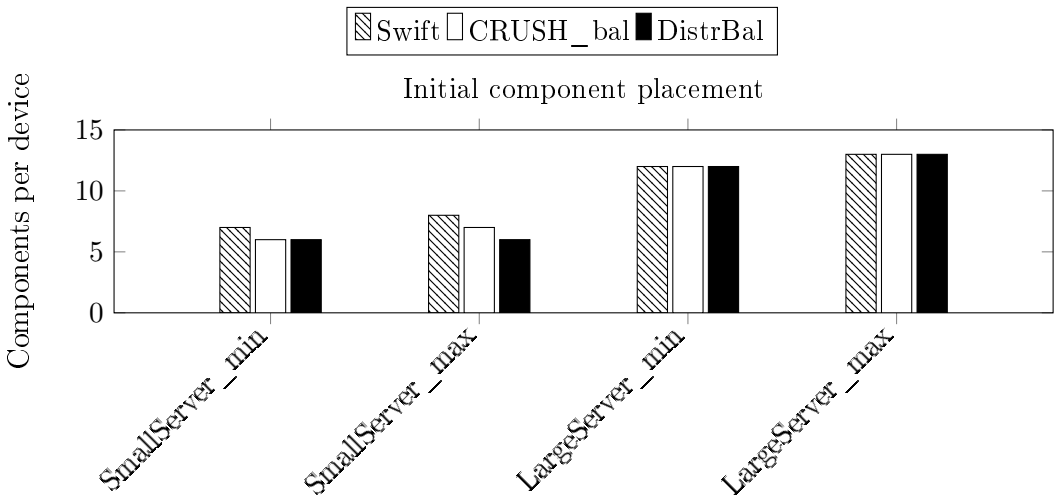
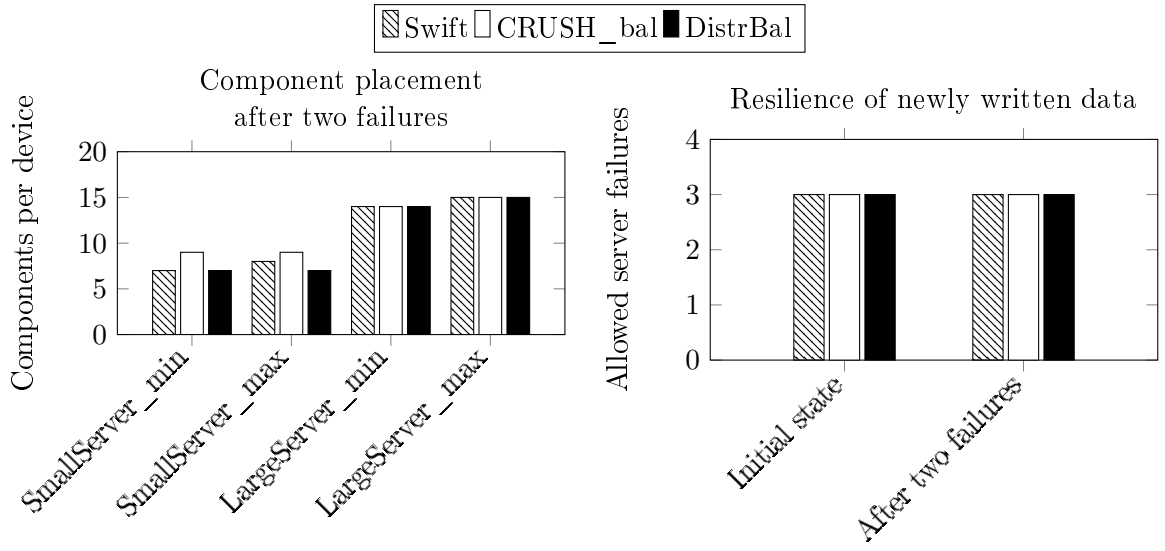


Figure 6.15: Minimal and maximal number of components at the beginning.

Failure Handling in Deployed Systems

In the next step, we simulated a hardware failure of two large servers, one after another. First, we killed processes responsible for handling storage on one server. After a minute, we killed processes on the other server. The systems responded as follows:



(a) Minimal and maximal number of components after two failures.

(b) After two server failures, new data are placed with resilience to three server failures.

Figure 6.16: Deployed system evaluation in a heterogeneous system with 16 servers.

- **Derrick** started DistrBal computations to adjust the component placement. In both cases, components ended up in locations optimal in terms of capacity (Fig. 6.16a) and resilience (Fig. 6.16b). In the wall-clock time, the computations took in total just below 3.5s, but during that time the CPU consumption of the service which conducts the DistrBal computation (and implements some other functionalities) was below 10% of a single core on each server.

- **Swift** does not remove failed devices from Rings automatically but uses pre-computed handoff locations [238]. We experimented with the handoff locations, but they did not balance load perfectly and, as a result, writing data to the system using handoff locations resulted in having some server without any free capacity, while other server had over 30% of free space. Therefore, we assumed that Ring rebuilding can be handled automatically after a failure, and included such an operation in our experiment. Rebuilding of the Ring after each failure took 2.2s but the component placement was not optimal in terms of capacity. Moreover, changing the Ring twice causes problems. Since Swift cannot read objects while they are moved, and does not implement any mechanism similar to our TrGuide, it requires waiting for transfers and reconstructions to be finished before the next balancing can be started. In fact, it is implemented by a timer which prevents changing the system balance until a given number of hours. Therefore, safe handling of two consecutive failures that happen shortly after each other is difficult.

- **Ceph** detects failed services promptly, but by default it waits for 10 minutes[3] before the service is considered down and removed from the CRUSH map. When a service is removed from the map, the computations take milliseconds, but typically the balancer needs to be executed a few times before it reaches its final results. As the balancer is

[3]The value is configurable with mon_osd_down_out_interval setting.

triggered by a timeout (which is by default 60 seconds), some time is required before it reaches its final state. Nevertheless, in terms of capacity, the placement was the worst of the three, as 9 components were placed on the small servers.

Finally, we started to write data. Data were written in small (512 KB) files / objects, to minimize the impact of uneven distribution of objects to components. All systems were able to accept the amount of data proportional to their component sizes, without any negative impact on data resilience (Fig. 6.16b): it was possible to restore the data even after three additional server failures.

This experiment leads to the following conclusions. First of all, despite the simple scenario, each algorithm delivered different usable capacity and Derrick provided the best result of the three. In a stable system, DistrBal bases on CentrBal results, and during a failure DistrBal moves components from failed devices to optimal locations. However, this does not mean that CentrBal can be entirely replaced with DistrBal, as we evaluate differences between CentrBal and DistrBal further in Section 6.5.2.

Furthermore, the model of handling failures in each system is different. Despite the fact that algorithms used in Ceph and Swift are able to compute results within seconds, their default use cases rely on a manual intervention or are delayed by minutes.

Finally, all considered algorithms provided resilient component placement. However, as we will describe shortly, it is not always the case.

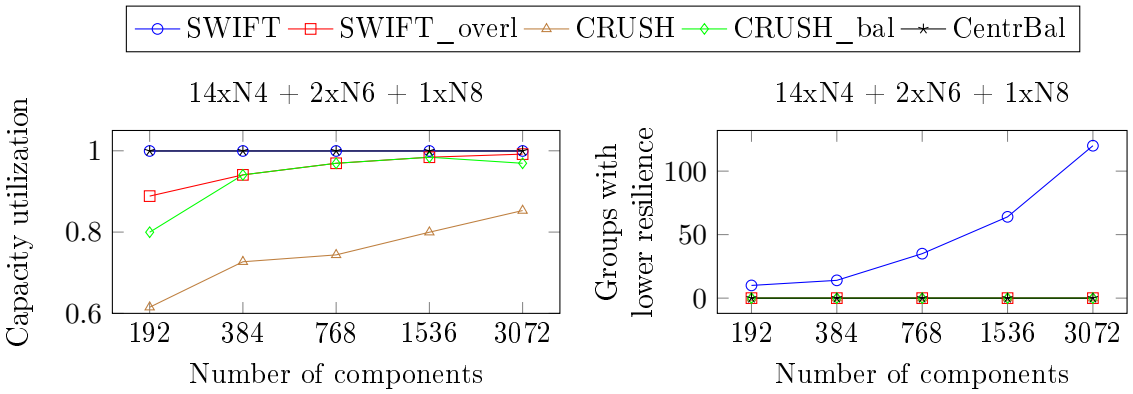
Capacity Utilization and Resilience

Using the aforementioned tools, we verified capacity utilization and system resilience in a configuration that contained 17 servers: 14 with 1 TB disks, 2 with 6 TB disks, and 1 with 8 TB disks. In such a configuration, with 12 components per group, two components of the same group were never on the same server and almost all available disk space was consumed ($<$0.0005 of space was not available because the 4 TB disk was not exactly 2x smaller than the 8 TB disk).

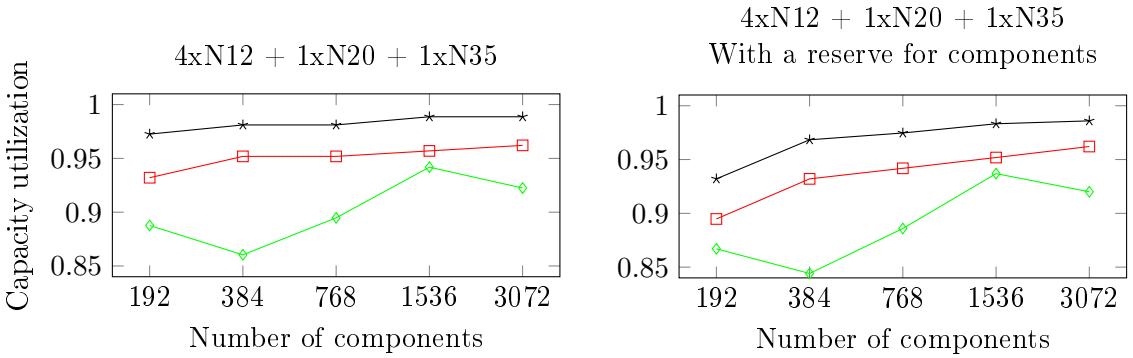
Nevertheless, data arrangements calculated by each algorithm differed significantly (see Fig. 6.17a). First of all, only CentrBal and Swift Rings optimally utilized the capacity. CRUSH was 1%–40% off, depending on the total number of groups and whether Ceph's balancer was used. For instance, with 256 groups, one of the servers with 4 TB disks received 150 components, which is 22 over the optimal result. Therefore, the component size (and so the capacity of the whole system) was decreased by 15%.

Swift Rings did failed to provide optimal resiliency, placing two components from the same group on one server. In heterogeneous systems, Swift allows changing the *overload* factor, which is a float value x that determines whether up x to additional device capacity can be used to improve system resilience. Although it was theoretically possible to find a perfect arrangement without overloading any server, Swift required a change of the overload factor to 0.0002 to find an arrangement that provided optimal resilience, but it decreased the capacity by 1%-11%.

Underutilization of the capacity is at odds with with the market demand for cost reduction in scalable storage. Moreover, Swift Rings requires additional attention to make sure that the data are kept resiliently. To clarify, we present the results of another experiment (Fig. 6.17b), which uses a system with 4x1.2 TB servers, 1x2.0 TB server, and 1x3.5 TB server (an analogue of Example 1 from Fig. 6.8 for 12 components per group). In that experiment, our goal was to keep at most 4 components per group on each server, which is more than in the most resilient solution (2 components per group), but allows much better capacity. For 192 groups,



(a) Capacity utilization and resilience violation in a heterogenous system with 17 servers total.



(b) Capacity utilization in a heterogeneous system with 6 servers.

(c) Lowered capacity utilization with an additional reserve for one component on each device.

Figure 6.17

the optimal solution in terms of capacity allows at most 4 components per group, but Swift may not find such an arrangement unless the overload is increased to a proper value (e.g., 0.01 overload was too small). When the overload was set to the smallest value that guaranteed the expected resilience, 2.6%–4.1% of the capacity was wasted compared to CentrBal with the same resilience. The aforementioned experiments show that CRUSH exactly follows the resilience requirements, but it achieves the lowest capacity utilization. Swift Rings does not find a resilient arrangement unless the overload value is set higher than it is really necessary. CentralBal results had an optimal capacity for the given resilience requirements in every experiment. The differences in capacity utilization were 1%–40%.

The reason for the capacity utilization differences is as follows. CRUSH depends on probability distribution, so its results are the worst due to variance. Therefore, Ceph's balancer improves the CRUSH results, but both the balancer and algorithms of Swift Rings are implemented in a way that allows one or two components off the perfect result, mostly to speed up the calculations. Therefore, the capacity loss depends on how many components are on each device: from a fraction of a percent if there are hundreds of components per device, to even tens of percents if there are few. Especially, if the difference between the largest and smallest devices is high, the capacity loss on each misplaced component increases (Fig 6.3). When a 20 TB disk hosts 100 components, 0.5 TB hosts only 3 and misplacing two components makes

a big difference.

In the end, to confirm a need for TrGuide that does not require a reserve of one component per device (as described in Section 6.4.4), we verified how such a reserve affects the capacity. Fig. 6.17c presents how such a reserve decreases the system capacity by up to 4%.

Transfers Required During Transition

In the next group of experiments, we verified how many components need to be transferred when a data arrangement changes. Limiting the transfers is important, because component movement consumes resources, especially network and disks, as data need to be read, sent, and written. In the presented results, we checked how many components needed to be transferred, which can be converted to the amount of data that need to be moved by multiplying the number of transferred components by the total system capacity, then dividing by the total number of components in the system. For instance, if a 100 TB system has 3072 components (256 groups), performing 400 transfers requires reading, sending, and writing 13 TB.

We compared CRUSH, CRUSH_bal, and Swift Rings with two versions of CentrBal to show how additional requirements affect the number of transfers: CentrBal_full optimizes all requirements, including DistComps and in-rack placement, and CentrBal_min only optimizes resilience, capacity, and the number of transfers (as in Section 6.3.2).

The simplest experiment adds servers one by one to a homogeneous system that starts with 12 servers (where there is exactly one component of each group on each of the servers). The system had a single rack and 256 groups. Swift Rings and CentrBal_min required the same number of transfers, and CentrBal_full required on average 2.3% additional movements to optimize the placement of DistComps (Fig. 6.18). CRUSH required on average 25% more transfers, and like in the previous experiments, the capacity was underutilized, so CRUSH_bal required even more. Therefore, in the basic experiment, CRUSH required many more transfers than necessary.

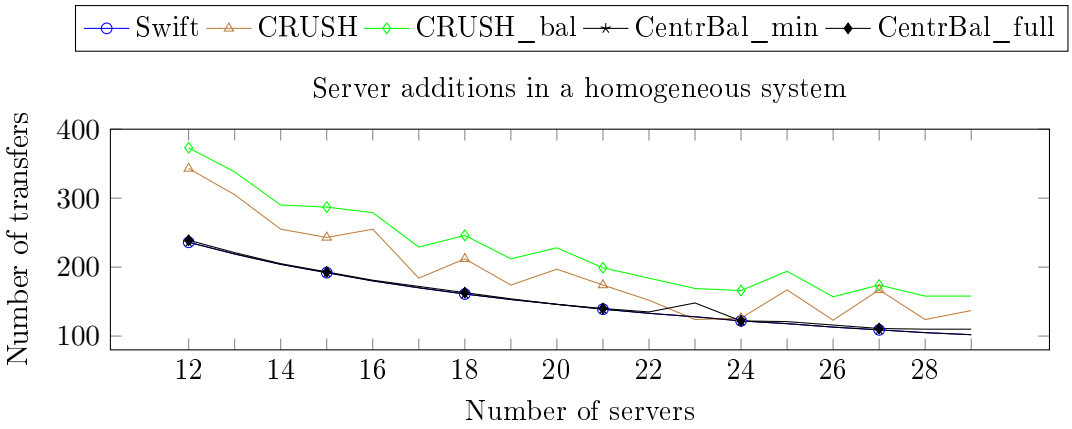


Figure 6.18: Number of transfers required when adding one server to a homogeneous system with a single rack and 256 groups.

In a multi-rack heterogeneous system, the results were more diverse. We verified the number of transfers needed during the addition of entire racks (15 servers) of randomly selected servers with 4/6/8 TB disks and 512 groups (Fig. 6.19). Swift Rings and CentrBal_min required almost identical numbers of transfers. As Swift Rings finds only an approximate solution, on average its result had 1.9% lower capacity and a manual[4] improvement to maximize

[4]In that experiment, Swift Rings could not find an optimal placement even when the *force* option was set.

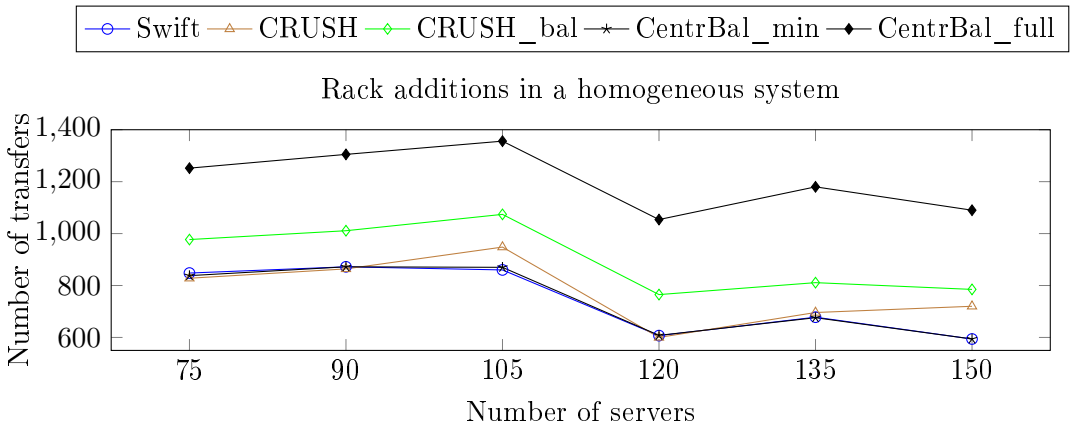


Figure 6.19: Number of transfers required when adding racks (15 servers) in a heterogeneous system with 512 groups.

the capacity required on average 3.5% more transfers.

Furthermore, in Swift Rings and CentrBal_min there is no option to keep components from one group in the same rack. Such a requirement can be described in CRUSH, but the balancer plugin spoils it. With CentrBal_full the distribution of groups across racks can be imperfect as well because there are more important requirements such as capacity, but CRUSH_bal gave 27%–45% worse results (Fig. 6.20). In terms of data movement, CRUSH_bal needed on average 18% more transfers than CentrBal_min and CentrBal_full required 30% more to improve a lot of placement of groups within racks.

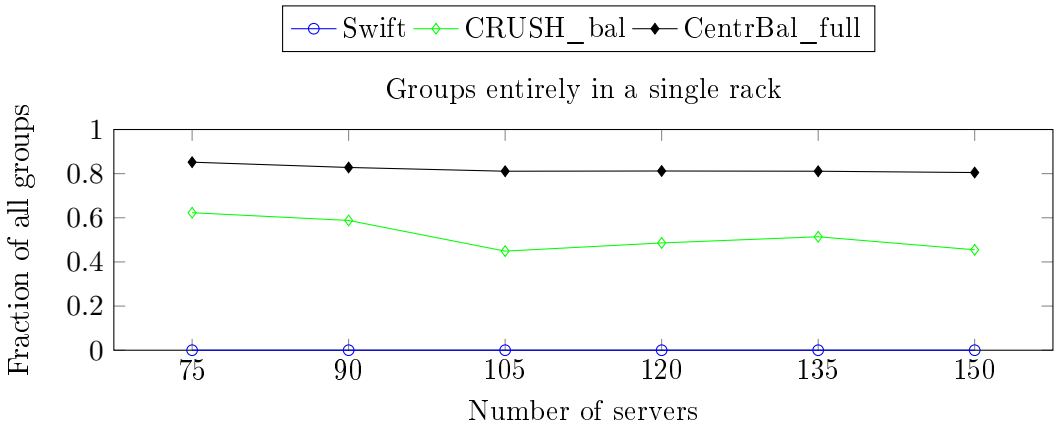


Figure 6.20: CRUSH_bal keeps fewer groups entirely in one rack in comparison to CentrBal_full.

Finally, we compared data movement in three scenarios in which not only the number of servers changes. If the number of groups is doubled in a homogeneous system with 32 servers Swift Rings and CentrBal_min require no changes, CentrBal_full moves a handful of components to optimize DistComps, but CRUSH moves half of the data (Fig. 6.21). Similarly, when changing whether capacity or resilience is more important in a system with 3x 4 TB and 1x 6 TB servers, Swift Rings requires 36% more transfers than CentrBal_full, and CRUSH

In other experiments, we observed that sometimes using the *force* flag helps to find an optimal result, but it also dramatically increases the number of transfers, as the algorithm moves many components unnecessarily.

moves far more components (Fig. 6.22). When a requirement for server-level resilience is changed, because the size of a homogeneous system is doubled, Swift Rings requires 13%–30% more transfers than CentrBal_full, and CRUSH requires 86%–89% more (Fig. 6.23).

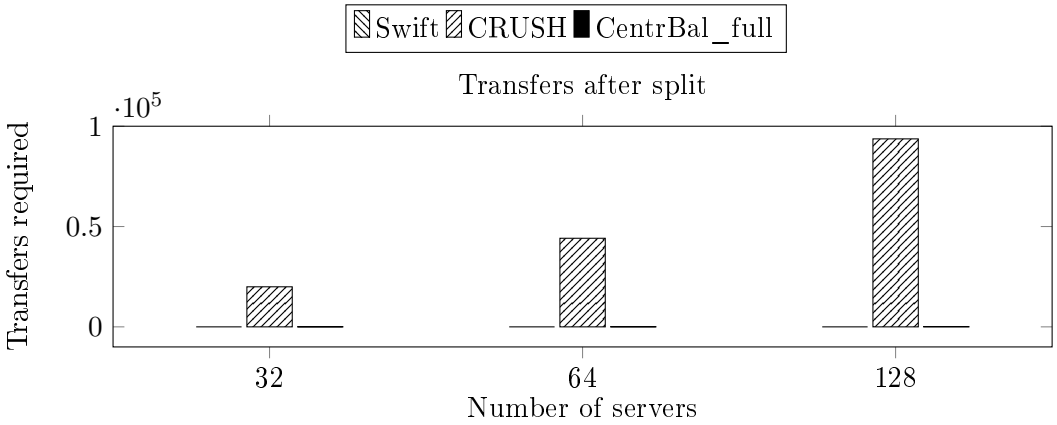


Figure 6.21: Transfers after a group split (first from 1024 to 2048, last from 4096 to 8192).

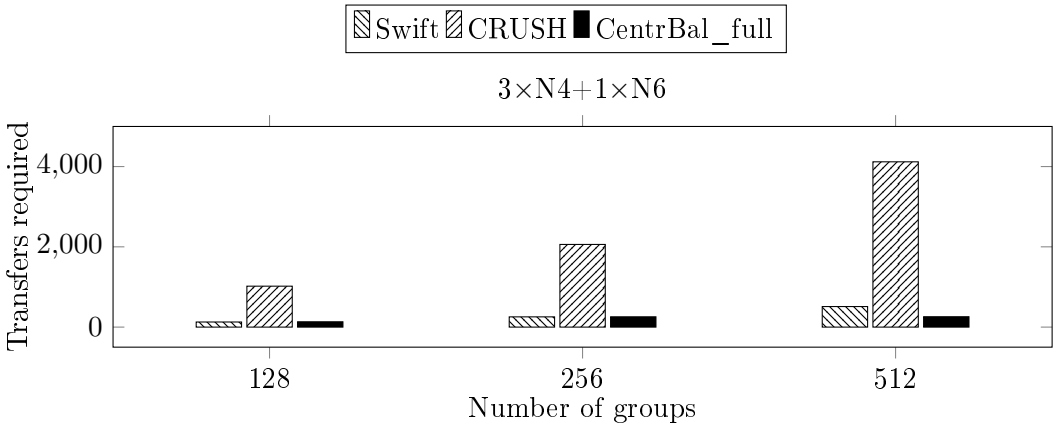


Figure 6.22: Transfers when started favoring capacity instead of resilience.

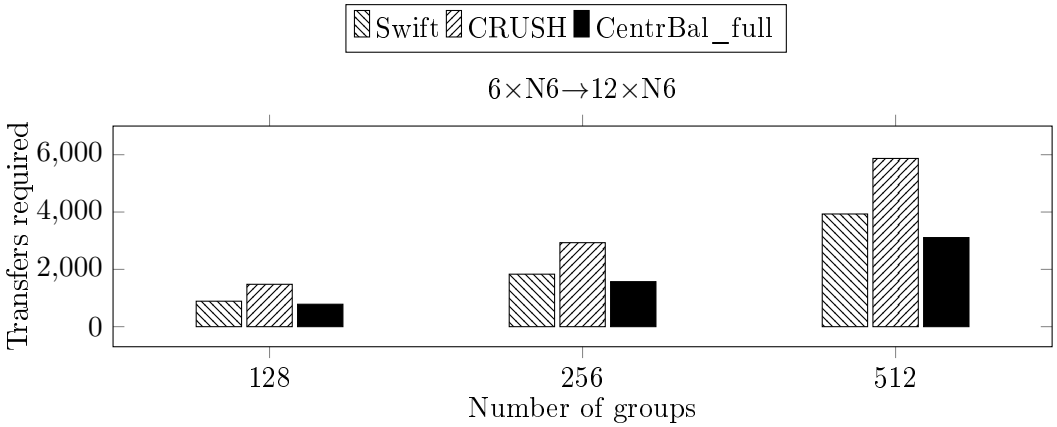


Figure 6.23: Transfers when changing server resilience during expansion.

To sum up, CentrBal_min not only ensures superior capacity utilization and resilience but also requires the lowest number of transfers. The main reason for the differences in the number of transfers is that CentrBal can spend additional time seeking solutions that provide the same results with the reduced number of data movements. CentrBal always uses a previous component placement as a starting point and with each moved component, it tries to improve as many metrics as possible. For comparison, in the other algorithms, a change of requirements such as desired resilience means practically balancing data from scratch. This is why even CentrBal_full, which additionally optimizes other metrics, moves less data than the other algorithms.

Distinguished Components Placement

Balancing of DistComps is important to evenly utilize system resources, which leads to improved performance. In our system, the first three components of each group are distinguished, and in Ceph the first component is distinguished, so we evaluated balancing of 1 or 3 DistComps (Fig. 6.24). In both CRUSH and CRUSH_bal, some servers have up to 15%-40% more DistComps than necessary, because neither of the algorithms tries to balance the distinguished components more than based on a probabilistic distribution. Swift does not use DistComps, so Swift Rings can even put all components with indexes 0 or 0,1,2 on the same servers (such server gets several times more DistComps than others). CentrBal_full finds a perfect or almost perfect distribution of DistComps in every case, which allows the highest resource utilization.

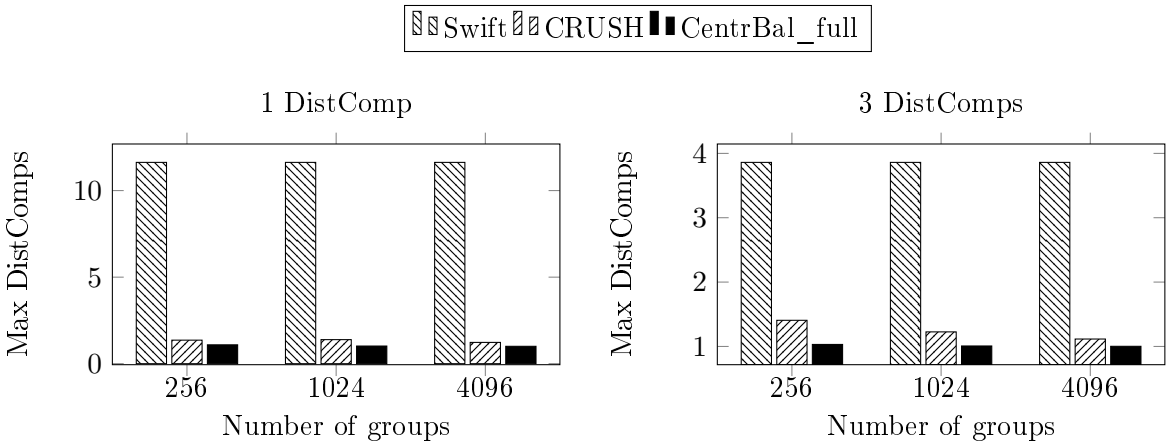


Figure 6.24: Maximal number of DistComps per server, normalized to the value in optimal components arrangement.

6.5.2. Evaluation of Distributed Balancing

In a stable system (without recent failures or changes in the number of devices), DistrBal simply uses the results already computed by CentrBal. In case of failures, DistrBal quickly adjusts the results provided by CentrBal. Therefore, we evaluate DistrBal by comparing its results with results computed by CentralBal in longer calculations. We randomly chose 25 heterogeneous configurations. In each configuration, we removed 1, 2, or 3 logical servers in two ways. First, we killed the chosen servers, so DistrBal moves the components from the removed servers. Then, we completely removed these servers from the system, so CentrBal could calculate its data arrangement. In 84% of cases, CentrBal found a better arrangement in terms of requirements on data arrangement, and in the remaining 16% the arrangement

was equivalent to DistrBal result. As DistrBal does not try to move as many components as CentrBal, it cannot find every improvement. Therefore, there were frequent differences on less important ScoreDims. For example. in 56% of cases the maximal number of DistComps per server was higher (not plotted).

The data arrangement calculated by CentrBal is superior to the one provided by DistrBal. However, the fact that DistrBal does not perform more complex operations is also its advantage, as it only computes a temporary state during failures. More complex operations require additional data movement, and thus also higher resource consumption, but there are already fewer resources due to missing devices and data reconstructions, so it is likely better to avoid such operations.

6.5.3. Computational Overhead

CentrBal and TrGuide Performance

The computational overhead was evaluated on a server with Intel Xeon E5640 Westmere 2.66GHz and 20GB of RAM (experiments needed even less memory). Fig. 6.25 presents the computation time of CentrBal when the size of a one server system (starting with 8 groups) is increased one server at a time. Initially, the computations take around 1 second, but as the number of groups is doubled each time the system size doubles. The final computation (with 256 groups total) took over 2 minutes. Fig. 6.26 shows the execution time of CentrBal_full during rack additions (15 servers) in a heterogeneous system (randomly selected servers with 4/6/8 TB disks) and 512 groups.

The computation time does not increase with the number of servers because with more servers, fewer components require movement. Another experiment shows how the execution time depends on the number of groups (Fig. 6.27). In all experiments, CentrBal_full finishes within 5 hours.

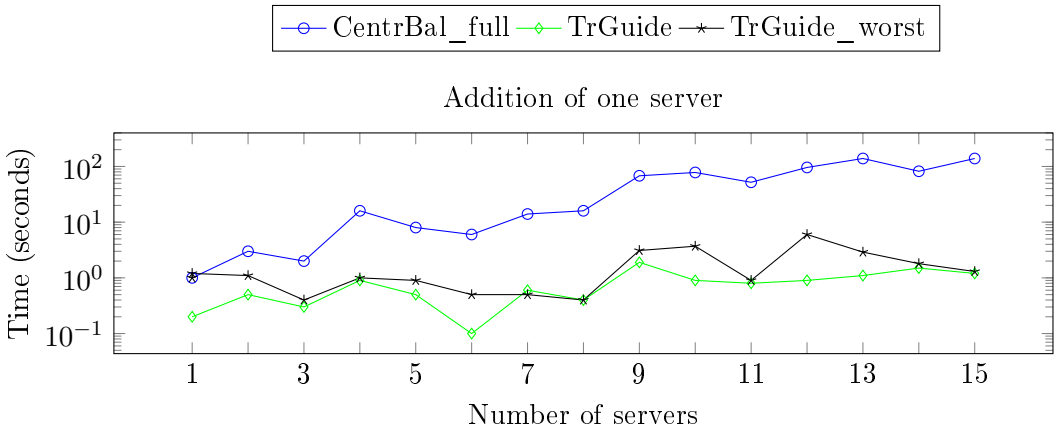


Figure 6.25: Computation time after addition one server in a small heterogeneous configuration. Number of groups scales with the system from 8 to 256.

TrGuide typically moves most components using relocations, so the computations take less than a few minutes. The very worst case (TrGuide_worst in Fig. 6.25, 6.26, and 6.27) is a theoretical situation in which there are a lot of components to move because the system size was increased a lot and immediately the system was entirely filled. In such a case, TrGuide must exceed the limit of non-stable components to avoid reaching out of space, and

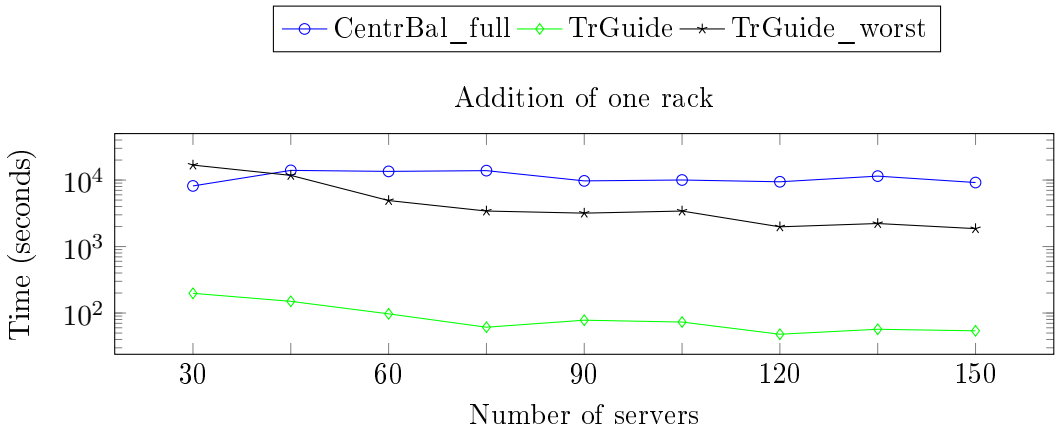


Figure 6.26: Computation time after addition of a rack (15 servers) with 512 groups.

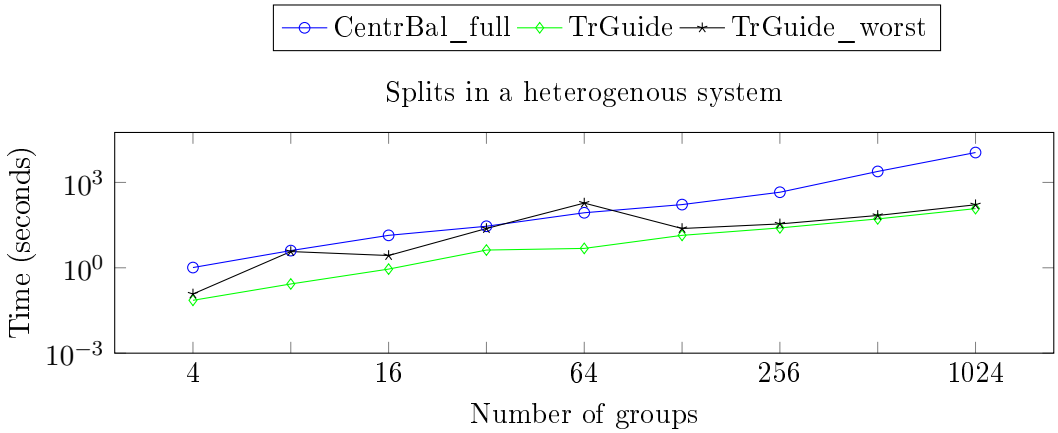


Figure 6.27: Computation time of splits in a heterogeneous system with 150 servers.

more computations are needed to find an optimal plan in terms of resilience and non-stable components.

Our implementations of CentrBal and TrGuide were tuned to find a solution within hours using a single core, and the goal was achieved. Moreover, both algorithms work much quicker in smaller systems, where there are typically fewer CPU resources. Therefore, in every conducted experiment, the execution of all three algorithms consumed far below 1% of daily CPU resources available in the system (considering computation time divided by the number of cores in all servers).

Speed-Up Considerations

We find the computation time of Derrick good enough, but the execution time can be further reduced by orders of magnitude. First of all, we expect that our implementation of Derrick still has room for optimizations. Second, the number of heuristics used to improve less significant ScoreDims can be reduced. As we will describe shortly, our evaluation of CentrBal_min confirms that finding a solution that uses only basic heuristics can be very quick. Finally, in large systems, the computation can be parallelized. In a larger system, two racks typically do not share any groups. Therefore, a system can be divided into smaller parts which can be balanced separately and then merged into one system.

To evaluate the parallelisation idea, we randomly selected 10 heterogeneous configurations with 4 racks in total and executed CentrBal_full as follows. First, we divided the system into two parts (2 racks each). Second, we calculated a component arrangement using CentrBal_full for each part. Finally, we executed CentrBal_full for the entire system with 4 racks, starting with the arrangement computed for the system parts. Such a method reduced the computation time on average by 31%. For instance, in one of the experiments, the computation of an arrangement for each part took 61s, and merging the results took 73s, while computing the arrangement for the entire system at once took 199s. Therefore, the total amount of work was similar (195s vs 199s), but the wall-clock time was reduced to 134s.

CentrBal_min Performance

We evaluated CentrBal_min, to verify the computational overhead with the limited number of requirements. CentrBal_min can promptly compute its result even in a massive system with thousands of servers and 3M components (Fig. 6.28). The computation after an addition of 15 machines took only 26 minutes, and in a theoretical scenario when the initial arrangement is random, the computing took under 10 hours.

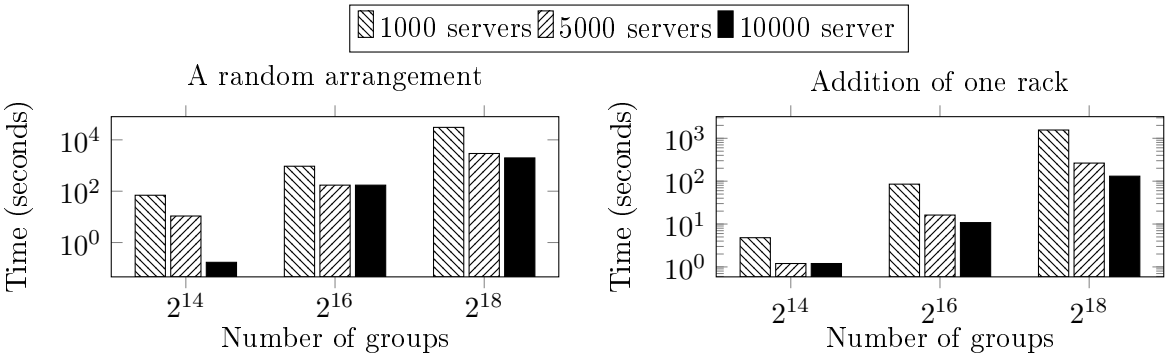


Figure 6.28: Computation time of CentrBal_min.

Query Time in Derrick's Results

In the end, we also evaluated what is the overhead of finding a component location in a result generated by Derrick (Fig. 6.29). As components are represented by two numbers, a hash map representation of Derrick's result is efficient. Our evaluation showed that even with millions of components and thousands of servers, performing a million queries using a single core does not even take half a second.

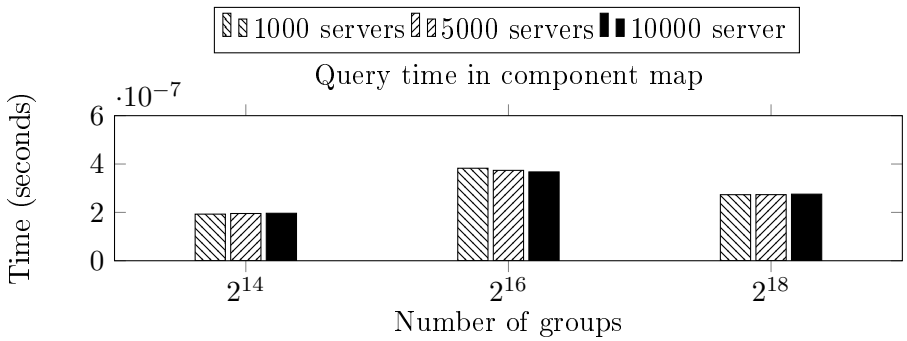


Figure 6.29: Query time in Derrick's result.

6.6. Formalization

This section contains a formalization of proofs using a mathematical notation. First, we formalize the problem of finding a specific component arrangement. Then, we formulate *Lemma 1*, *Lemma 2*, *Lemma 3*, and *Lemma 4* using the notation and prove them.

6.6.1. Problem Statement

The problem of finding a specific arrangement is selecting a particular $a_i \in A$ for a given system

$$system =< S, G, C, A, group, capacity >: System \tag{6.1}$$

where

$$S = \{s_0, s_1, ..., s_x\} \text{ is a finite set of servers.} \tag{6.2}$$
$$G = \{g_0, g_1, ..., g_y\} \text{ is a finite set of groups.} \tag{6.3}$$
$$C = \{c_0, c_1, ..., c_z\} \text{ is a finite set of components.} \tag{6.4}$$
$$group : C \to G \text{ is a function that maps each component to a group.} \tag{6.5}$$
$$capacity : S \to \mathbb{N} \text{ maps each server to its capacity (in bytes).} \tag{6.6}$$
$$a_x : C \to S \text{ maps each component to a server. We call } a_x \text{ an arrangement.} \tag{6.7}$$
$$A = \{a_0, a_1, ..., a_n\} \text{ is a finite set of all possible arrangements.} \tag{6.8}$$

6.6.2. Auxiliary Functions, Definitions and Corollaries

Function 1 *Function* ***components*** *provides a subset of components for a given arrangement and a server:*

$$components : A \times S \to 2^C; \tag{6.9}$$
$$components(a_x, s) = \{c \in C | a_x(c) = s\} \tag{6.10}$$

Function 2 *Function* ***componentSize*** *computes a quotient of a server's capacity and its number of components:*

$$componentSize : A \times S \to \mathbb{N}; \tag{6.11}$$
$$componentSize(a_x, s) = capacity(s)/|components(a_x, s)| \tag{6.12}$$

Function 3 *Function* ***componentSize'*** *computes a quotient of a server's capacity and its number of components increased by 1, which equals componentSize after the server accepts an additional component:*

$$componentSize' : A \times S \to \mathbb{N}; \tag{6.13}$$
$$componentSize'(a_x, s) = capacity(s)/(|components(a_x, s)| + 1) \tag{6.14}$$

Function 4 *Function* ***systemCapacity*** *computes the capacity available in the system for a given arrangement:*

$$systemCapacity : A \to \mathbb{N}; \tag{6.15}$$
$$systemCapacity(a_x) = |C| * \min_{s \in S}(\{componentSize(a_x, s)\}) \tag{6.16}$$

Definition 1 *Arrangement a_x has an optimal capacity if and only if*

$$systemCapacity(a_x) = \max_{a \in A}(systemCapacity(a)) \tag{6.17}$$

Function 5 *Function* ***conflicts*** *returns the number of components for a given group on a server:*

$$conflicts : A \times S \times G \to \mathbb{N}; \tag{6.18}$$
$$conflicts(a_x, s, g) = |\{c \in C | group(c) = g \wedge c \in components(a_x, s)\}| \tag{6.19}$$

Function 6 *Function* ***maxConflicts*** *returns the maximal number of conflicts in the system:*

$$maxConflicts : A \to \mathbb{N}; \tag{6.20}$$
$$maxConflicts(a_x) = \max_{s \in S, g \in G}(conflicts(a_x, s, g)) \tag{6.21}$$

Definition 2 *a_x is an arrangement with an optimal resilience if and only if*

$$maxConflicts(a_x) = \min_{a \in A}(maxConflicts(a)) \tag{6.22}$$

Definition 3 *a_x meets a resilience restriction of $r \in \mathbb{N}$ if and only if*

$$maxConflicts(a_x) <= r \tag{6.23}$$

Definition 4 *a_x has an optimal capacity within a resilience restriction of $r \in \mathbb{N}$ iff*

$$maxConflicts(a_x) <= r \text{ and} \tag{6.24}$$
$$systemCapacity(a_x) = \max_{\{a \in A | maxConflicts(a) <= r\}}(systemCapacity(a)) \tag{6.25}$$

Corollary 1 *If a_x meets a resilience restriction of $r \in \mathbb{N}$ then*

$$\forall s \in S \ |components(a_x, s)| <= r * |G| \tag{6.26}$$

Function 7 *Function* ***groupSize*** *returns the number of components in a group:*

$$groupSize : G \to \mathbb{N}; \tag{6.27}$$
$$groupSize(g) = |\{c \in C | group(c) = g\}| \tag{6.28}$$

Definition 5 *All groups in the system are equinumerous if and only if*

$$\exists_{n \in \mathbb{N}} \forall_{g \in G} |groupSize(g)| = n \tag{6.29}$$

Function 8 *Function* ***sumConflicts*** *returns the sum of conflicts for one group in total (i.e., on all servers):*

$$sumConflicts : A \times G \to \mathbb{N}; \tag{6.30}$$
$$sumConflicts(a_x, g) = \sum_{s \in S} conflicts(a_x, s, g) \tag{6.31}$$

Corollary 2

$$\forall a_1, a_2 \in A \ \forall g \in G \ sumConflicts(a_1, g) = sumConflicts(a_2, g) = groupSize(g) \tag{6.32}$$

6.6.3. Operations

Derrick solves the problem of finding an arrangement by moving components between servers. Therefore, Derrick has a set of possible operations that move components between servers. Each operation is a partial function that is defined only if the components are indeed located on their initial servers. We specify the following operations:

Operation 1 ***Relocation*** *is an operation that moves a component from its initial location to a different server:*

$$relocation : C \times S \times S \times A \nrightarrow A; \tag{6.33}$$

$$relocation(c_a, s_{initial}, s_{new}, a_x) = a_y \Leftrightarrow a_y(c) = \begin{cases} s_{new} & if\ c = c_a \wedge a_x(c_a) = s_{initial} \\ a_x(c) & if\ c \neq c_a \wedge a_x(c_a) = s_{initial} \end{cases} \tag{6.34}$$

Operation 2 ***Push*** *is an operation that moves a component from its initial location to a different server, then takes a different component from that servers and moves it somewhere else:*

$$push : C \times C \times S \times S \times S \times A \nrightarrow A; \tag{6.35}$$

$$push(c_a, c_b, s_a, s_b, s_c, a_x) = a_y \Leftrightarrow a_y(c) = \begin{cases} s_b\ if\ c = c_a \wedge a_x(c_a) = s_a \wedge a_x(c_b) = s_b \\ s_c\ if\ c = c_b \wedge a_x(c_a) = s_a \wedge a_x(c_b) = s_b \\ a_x(c)\ otherwise \end{cases} \tag{6.36}$$

Definition 6 *A* ***composition*** ϕ ***of operations*** $o_1, o_2, ..., o_n$ *is an operation that arises from combining the operations, i.e.* $\phi(a_x) = a_y \Leftrightarrow o_n(...o_2(o_1(a_x))...) = a_y$

Definition 7 *Arrangement* a_x ***can be reached*** *from arrangement* a_y *using operations* $o_1, o_2, ..., o_n$*, iff there exists a composition of operations* ϕ *for which* $\phi(a_x) = a_y$*.*

6.6.4. Lemma 1

Lemma 1 *Assume a system and* $r \in \mathbb{N}$ *for which an arrangement meeting resilience restriction* r *exists. If groups are equinumerous, for every arrangement* a_x *there exists an arrangement* a_y *with an optimal capacity and within resilience restriction* r*, such that* a_y *can be reached from* a_x *using relocations and pushes.*

Proof: The proof consists of two steps. First, we show that there exists an arrangement a_r that meets a resilience restriction r and can be reached from a_x by using only relocations. Then, we show that a_y can be reached from a_r using relocations and pushes.

If a_x meets resilience restriction r, then $a_x = a_r$. Otherwise, from Definition 3, at least one server s_a and would one group g_a exist for which $conflicts(a_x, s_a, g_a) > r$. Moreover, for at least one server s_b, $conflicts(a_x, s_b, g_a) < r$, otherwise $sumConflicts(a_x, g_a) > r * |S|$ that contradicts the existence of an arrangement meeting a resilience restriction r (from Corollary 1). Therefore a following relocation of a component c_a from s_a can be done:

$$a_i = relocation(c_a, s_a, s_b, a_x) \tag{6.37}$$

$$conflicts(a_i, s_b, g_a) <= r \tag{6.38}$$

$$conflicts(a_i, s_a, g_a) = conflicts(a_x, s_a, g_a) - 1 \tag{6.39}$$

Repeating such steps, for every server and group that does not meet resilience restriction r leads to a_r.

If a_r has the optimal capacity within resilience restriction r, then $a_y = a_r$. Otherwise, at least one server, s_a, which lowers the system capacity exists ($componentSize(a_r, s_a) < \frac{systemCapacity(a_y)}{|C|}$). $Capacity(s_a)$ is given but the usable component size can be increased by moving a component out of s_a.

As capacity of a_r is not optimal within resilience restriction r, there must also be a server s_b satisfying the two following formulas:

$$componentSize'(a_r, s_b) >= \frac{systemCapacity(a_y)}{|C|} \tag{6.40}$$

$$\exists g_b \; conflicts(a_r, s_b, g_b) < r \tag{6.41}$$

The existence of s_b can be explained as follows. As s_a underutilizes the capacity, another server must be able to take at least one additional component without negatively affecting the system capacity. Therefore, at least one server can accept additional component in terms of the capacity. Assume that for every server, s_f, being able to accept the additional component for each $g_b \in G$, $conflicts(a_r, s_f, g_b) = r$. It would mean that there are $|S|*|G|*r$ components in the system, and hence each of the servers hosts exactly $r*|G|$ components, so none of these servers can accept any component. This is a contradiction, as it means that no arrangement meeting the resilience restriction r exists in which s_a hosts fewer components (as none of the other servers can accept it).

Therefore, s_b can take a component in terms of capacity and we want to move one of the components from s_a. If any component c_a on s_a with $group(c_a) = g_b$ exists, then c_a can be moved to s_b with the relocation. Otherwise, placing any c_a such that $g(c_a) = g_a$ on s_b would break the resilience restriction r. Therefore, $conflicts(a_r, s_a, g_a) > conflicts(a_r, s_a, g_b) = 0$ (as there are no components from g_b), and $conflicts(a_r, s_b, g_a) > conflicts(a_r, s_b, g_b)$ (as s_b accepts g_b but not g_a). However, the groups are equinumerous, so there must be at least one server s_c for which $conflicts(a_r, s_c, g_b) > conflicts(a_r, s_c, g_a)$ (there are two servers with more components from g_a, we need to place components from g_b somewhere). Therefore, s_c hosts component c_b with $group(c_b) = g_b$, so $push(c_a, c_b, s_a, s_c, s_b, a_x)$ can be done (see Fig. 6.30).

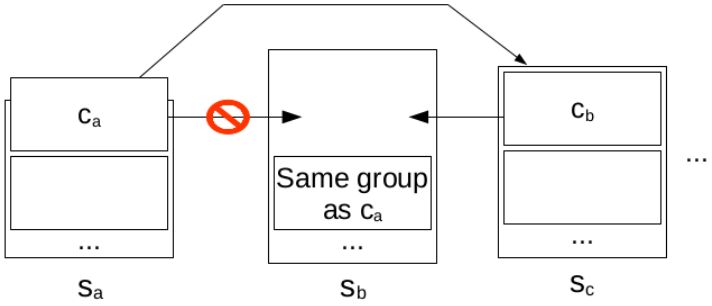


Figure 6.30: An example of a push that improves capacity. A concrete example of such situation is presented in Fig. 6.8.

The reasoning can be repeated until the optimal capacity with resilience restriction r is reached QED.

6.6.5. TrGuide Definitions

To formalize *Lemma 2*, *Lemma 3*, *Lemma 4* additional definitions and operations related to TrGuide are introduced:

Operation 3 ***Swap*** *is an operation that exchanges the location of two components between servers:*

$$swap : C \times C \times S \times S \times A \nrightarrow A; \tag{6.42}$$

$$swap(c_a, c_b, s_a, s_b, a_x) = a_y \Leftrightarrow a_y(c) = \begin{cases} s_a \text{ if } c = c_b \wedge a_x(c_a) = s_a \wedge a_x(c_b) = s_b \\ s_b \text{ if } c = c_a \wedge a_x(c_a) = s_a \wedge a_x(c_b) = s_b \\ a_x(c) \text{ otherwise} \end{cases} \tag{6.43}$$

Operation 4 *$Cycle_n$ is a generalization of a swap to more than two components:*

$$cycle_n : \underbrace{C \times ... \times C}_{n} \times \underbrace{S \times ... \times S}_{n} \times A \nrightarrow A; \tag{6.44}$$

$$cycle_n(c_0, ..., c_{n-1}, s_0, ..., s_{n-1}, a_x) = a_y \Leftrightarrow \tag{6.45}$$

$$a_y(c) = \begin{cases} s_1 \text{ if } c = c_0 \wedge \forall i \in \{0, ..., n-1\}\ a_x(c_i) = s_i \\ ... \\ s_0 \text{ if } c = c_{n-1} \wedge \forall i \in \{0, ..., n-1\}\ a_x(c_i) = s_i \\ a_x(c) \text{ otherwise} \end{cases} \tag{6.46}$$

Definition 8 *Each operation* ***moves at a time*** *a number of components equal to the number of components that change their position as a result of the operation. Relocation moves 1 component at a time, swap and push move 2 at a time, and $cycle_n$ moves n components at a time.*

Definition 9 *For any two arrangements a_x and a_y, $nextBalancingStep(a_x, a_y)$ is an arrangement, a_z, meeting all of the following criteria:*

$$crit.\ \#1:\ |\{c \in C | a_x(c) = a_y(c)\}| < |\{c \in C | a_z(c) = a_y(c)\}| \tag{6.47}$$

$$crit.\ \#2:\ max(maxConflicts(a_x), maxConflicts(a_y)) >= maxConflicts(a_z) \tag{6.48}$$

$$crit.\ \#3:\ min(systemCapacity(a_x), systemCapacity(a_y)) <= systemCapacity(a_z) \tag{6.49}$$

The intuition behind the three criteria is as follows. Crit. #1 ensures that a_z has more components on their locations from a_y than a_x, so the arrangement moves towards a_y. Crit. #2 ensures that the system resilience is not worsen. Crit. #3 ensures that the system capacity is not worsen.

6.6.6. Lemma 2

Lemma 2 *For any given arrangements a_x and a_y, and a component, c_a, such that $a_x(c_a) = s_a \wedge a_y(c_a) = s_b$, if $relocation(c_a, s_a, s_b, a_x)$ violates crit.#2, then there is a component, c_b, such that $a_x(c_b) = s_b$ and $swap(c_a, c_b, s_a, s_b, a_x)$ is $nextBalancingStep(a_x, a_y)$.*

Proof: If relocating c_a to s_b violates *crit.*#2 (on s_b), then s_b already hosts the maximal number of components of $group(c_a)$, which is $max(maxConflicts(a_x), maxConflicts(a_y))$. This implies that a component, c_b, with $group(c_a) = group(c_b)$ for which $a_y(c_b) \neq s_b$ exists,

because c_a is guaranteed to be allowed on s_b in a_y (see Fig. 6.31). Swaps can always be done without violating *crit.*#3, as a swap does not change the number of components on any machine. Moreover, *crit.*#1 is improved, as c_a reaches its location from a_y, whereas c_b has a different location in a_x than in a_y (i.e. moving c_b has no negative impact on *crit.*#1). Therefore $swap(c_a, c_b, s_a, s_b, a_x)$ can be the $nextBalancingStep(a_x, a_y)$.

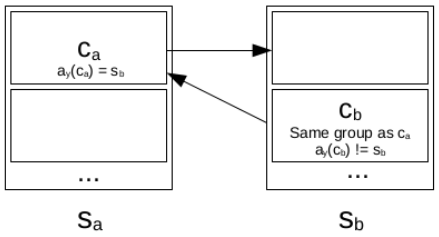


Figure 6.31: An example of swap that is possible when there is *crit.*#2 violation on s_b.

6.6.7. Lemma 3

Lemma 3 *For any given arrangements a_x and a_y, if a cycle, p, of length $n > 3$, formed by components $c_0, c_2, ..., c_{(n-1)}$ for which $a_y(c_0) = a_x(c_1) \wedge ... \wedge a_y(c_{(n-1)}) = a_x(c_0)$ exists, TrGuide can find the $nextBalancingStep(a_x, a_y) = a_z$ by moving at most three components at a time.*

Proof: First, consider the situation that $\exists i, j \in \{0...n-1\}((i < j) \wedge (a_x(c_i) = a_x(c_j)))$. In such a case, reduce the considered cycle to $c_i, ..., c_j$. If the length of the cycle is 2 or 3, use *swap* or $cycle_3$ as **nextBalancingStep**. Otherwise, repeat the reasoning with the reduced cycle $(c_i, ..., c_j)$. Using this approach multiple times we can eliminate all sub-cycles of lengths at most 3.

At this point, we can thus assume that the length of the cycle is $n > 3$ and $a_x(c_1), ..., a_x(c_n)$ are pairwise different. Using $cycle_n(c_0, ..., c_{n-1}, a_x(c_0), ..., a_x(c_{n-1}), a_x)$ does not change the capacity but moves more than three components at a time. If possible for any i, one of the following operations should be selected as $nextBalancingStep(a_x, a_y)$:

$$relocation(c_i, a_x(c_i), a_y(c_i), a_x) \tag{6.50}$$

$$swap(c_i, c_{(i+1)\%n}, a_x(c_i), a_x(c_{(i+1)\%n}), a_x) \tag{6.51}$$

$$cycle_3(c_i, c_{(i+1)\%n}, c_{(i+2)\%n}, a_x(c_i), a_x(c_{(i+1)\%n}), a_x(c_{(i+2)\%n}), a_x) \tag{6.52}$$

If none of the three operations is allowed for any i, $cycle_n$ does not violate *crit.*#2, otherwise a swap would be possible (from *Lemma 2* applied to every relocation that comprises the cycle). Therefore, $cycle_n$ could be the $nextBalancingStep(a_x, a_y)$ but it moves too many components.

We can prove that if all possible *swap* and $cycle_3$ operations violate *crit.*#2, then $cycle_n$ violates it as well, which leads to a contradiction (as we already concluded that $cycle_n$ does not violate *crit.*#2). For every $j < n$, $a_x(c_j)$ hosts the maximal allowed number of components from the $group(c_{(j+1)\%n})$, otherwise $swap(c_j, c_{(j+1)\%n}, a_x(c_j), a_x(c_{(j+1)\%n}), a_x)$ would possible. Alike, for every $j < n$, $a_x(c_j)$ hosts the maximal allowed number of components from the $group(c_{(j+2)\%n})$, otherwise the $cycle_3(c_j, c_{(j+1)\%n}, c_{(j+2)\%n}, a_x(c_j)$, $a_x(c_{(j+1)\%n}), a_x(c_{(j+2)\%n}), a_x)$ would be possible. Moreover, as such $cycle_3$ moves the components $c_j, c_{(j+1)\%n}$ to their a_y location, moving any of the components from server $a_x(c_{(j+2)\%n})$

(even the one already on its a_y location) does not violate *crit.*#1. Therefore, any of $a_x(c_{(j+2)\%n})$ components can be moved to $a_x(c_{(j)})$ without violating *crit.*#1. Since both *crit.*#1 and *crit.*#3 cannot be violated by such a $cycle_3$, it means it violates the *crit.*#2 and $a_x(c_j)$ must host the maximal allowed number of components from every group hosted on $a_x(c_{(j+2)\%n})$. For the same reason, $a_x(c_{(j+2)\%n})$ must host the maximal allowed number of components from every group hosted on $a_x(c_{(j+4)\%n})$, which also implies that they are also hosted on $a_x(c_{(j)\%n})$. In general, $a_x(c_{j\%n})$ must host host the maximal allowed number of components from every group hosted on $a_x(c_{(j+2*m)\%n})$ for $m \in \mathbb{N}$.

As we already stated, $a_x(c_j)$ hosts the maximal allowed number of components from the $group(c_{(j+1)\%n})$ (swap violates *crit.*#2), just as $a_x(c_{(j-1)\%n})$ hosts the maximal allowed number of components from the $group(c_{(j+1)\%n})$ ($cycle_3$ violates *crit.*#2). Therefore, the limit of allowed components from $group(c_{(j+1)\%n})$ is reached on every server (see Fig. 6.32) because the limit is reached for both $a_x(c_{(j+2*m)\%n})$ and $a_x(c_{(j-1+2*m)\%n})$ for $m \in \mathbb{N}$. It contradicts the fact that $cycle_n$ is possible (because *crit.*#2 would be violated) QED.

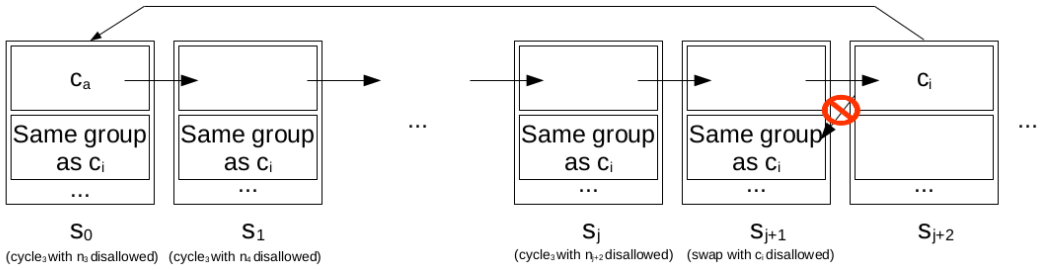


Figure 6.32: An example where component c_i is not allowed on s_1 (contradiction).

6.6.8. Lemma 4

Lemma 4 *For any given arrangements a_x and a_y, TrGuide can find $nextBalancingStep(a_x, a_y) = a_z$ by moving at most three components at a time.*

Proof: Let us select a component c_a for which $a_x(c_a) = s_a \wedge a_y(c_a) = s_b \wedge s_a \neq s_b$. If s_b can accept c_a without violating *crit.*#2 and *crit.*#3, then the $nextBalancingStep(a_x, a_y)$ is $relocation(c_a, s_a, s_b, a_x)$.

If relocating c_a to s_b violates *crit.*#2 (on s_b), then *Lemma 2* can be used to do a swap.

If relocating c_a to s_b violates *crit.*#3 but does not violate *crit.*#2, a component, c_b, exists for which $a_y(c_b) \neq s_b \wedge a_x(c_b) = s_b$ (because in a_x there is no space on s_b for c_a but there is in a_y). If c_b can be swapped with c_a without violating *crit.*#2 (by relocating c_b to s_a when swapping c_a and s_b), then $swap(c_a, c_b, s_a, s_b, a_x)$ is $nextBalancingStep(a_x, a_y)$.

If none of the relocations or swaps introduced to this point are possible for c_a, then repeat the steps for every other component, c_i, for which $a_x(c_i) = s_i \wedge a_y(c_i) = s_j \wedge s_i \neq s_j$. If none of the relocations or swaps are possible for any of such components, it means there is a cycle, because each of the components has its target on a server where there is no additional space (*crit.*#3 violated when relocating), so each target server also hosts at least one component that will change its location. Cycles of length 2 and 3 can be trivially solved by moving up to three components. If the cycle is longer than 3 components, *Lemma 3* can be used to find the $nextBalancingStep(a_x, a_y)$ within the cycle.

In this way, all cases have been examined QED.

6.7. Conclusions

We examined the challenges in providing self-managed continuous scalability in heterogeneous distributed storage systems. As a solution, we presented Derrick, which is a novel algorithm for finding data arrangements that meet multiple requirements. We implemented Derrick in HYDRAstor and evaluated it against existing state-of-the-art solutions in Ceph and Swift. A deployment of the described techniques in production also confirmed their effectiveness. Our approach guarantees maximal resilience, higher capacity utilization, and less data movement. For specific requirements, such as balancing distinguished components or keeping groups within racks, Derrick achieves respectively 15%–40% and 27%–45% better results than the state of the art. Moreover, to ensure flexible scalability, it introduces the changes in data arrangement gradually, without superfluous disruption of system operations.

Derrick finds satisfactory solutions in systems from a few to thousands of devices within the given time. The general idea can be adapted to systems with a different set of requirements.

Chapter 7

Conclusions and Future Work

Efficient implementation of distributed storage systems with deduplication is challenging, and adaptation of such systems to the requirements of the cloud computing era is a broad research topic. In this dissertation, we proposed ObjDedup, InftyDedup, and Derrick, which introduce novel solutions to three important research problems in the area. Each of the three solutions brings a significant improvement over the state of the art. They have been or will be deployed in HYDRAstor, a commercial PBBA utilized by many organizations. In effect, our work betters actual backup and archival storage systems.

ObjDedup, an object storage layer for backup systems with block-level deduplication, achieves a 1.8–3.93x higher write throughput than object storage without in-line deduplication. Compared to object storage on top of a state-of-the-art file-based backup system, it also processes 5.26–11.34x more object put operations per time unit. Since, to the best of our knowledge, ObjDedup is the first object storage layer of that kind, we expect it to change how PBBA systems can be employed to implement cloud-oriented functionality and how PBBA systems can interact with cloud applications.

InftyDedup, unlike other tiering to cloud solutions, maximizes scalability by utilizing cloud services not only for storage but also for computation to deduplicate multi-petabyte backups from multiple sources at costs on the order of a couple of dollars. The design of InftyDedup allows scaling it without restrictions, and hence, deduplicated data can be moved to the cloud on an unprecedented scale. Moreover, InftyDedup dynamically selects between hot and cold cloud storage based on the characteristics of each data chunk to reduce the costs further. As deduplicated data are kept in small blocks (which are not well-suited for cold cloud storage) without our technique, storing deduplicated data in cold cloud storage might drastically increase the overall costs. Again, we are not aware of any prior solution that would effectively combine the characteristics of deduplicated data and the cold cloud storage pricing model, and hence our work can open up new research directions.

Derrick is a data balancer designed to make its decisions quickly in case of failures, yet to be allowed to take extra time to find a nearly optimal data arrangement and a plan for reaching it when the device population changes. Compared to balancing algorithms in two other state-of-the-art systems, Derrick provides better capacity utilization, reduced data movement, and improved performance. Moreover, it can be easily adapted to meet custom placement requirements. Data balancing in distributed storage is a general problem that exists in various systems, including those without deduplication. Therefore, not only does Derrick significantly improve systems that implement ObjDedup and InftyDedup, but it can also be adapted to many other distributed storage systems.

When it comes to future research directions, history shows that backup workloads evolve

in time [6]. Therefore, detailed quantitative research on how ObjDedup, InftyDedup, and Derrick work over longer periods of time is such important future work. For all three, we made numerous assumptions based on modern use cases of backup systems. The quantitative analysis would not only verify if our assumptions were correct but would also examine if new possibilities introduced by our work changed the behavior of the systems' users.

Furthermore, clouds are constantly evolving as well, and our research was primarily inspired by changes introduced in recent years. For instance, the fact that multiple backup applications implement writing to object storage with peculiar workloads necessitated efficient metadata handling in ObjDedup. In turn, cold archive cloud storage with instant access facilitated mixing storage types in InftyDedup. We expect that in the future, both cloud providers and applications that utilize clouds will offer new features. Consequently, new research will be needed to take advantage of the emerging technologies. Similarly, future data carriers, such as the aforementioned DNA storage, quartz glass, or HDDs one hundred times greater than today, can change the way storage systems organize their data, and hence our methods might require adjustments (or even a redesign).

Finally, novel ideas appear in storage research regularly, and many of them can be used to further improve storage systems with deduplication. In particular, as explained in Section 5.1.4, security in cloud storage with deduplication has received significant research attention recently, and hence, the integration of new methods with our solutions is an interesting challenge. Similarly, we think it is possible to improve our systems by applying recently developed methods of increasing reliability mentioned in Section 6.1. Overall, based on my conversations with experts met during the leading storage conferences, deduplication increasingly seems to be a desired feature in cloud storage systems, and thus further work on the topic will likely follow.

Bibliography

[1] ABCSI. Dell EMC PowerEdge R740xd 2U Rack Server, 2023. https://abcsistore.com/products/dell-emc-poweredge-r740xd-2u-rack-server.

[2] Acronis. Retention rules: how and when they work, 2022. https://kb.acronis.com/content/68304.

[3] Agarwala, A., Singh, P., and Atrey, P. K. DICE: A dual integrity convergent encryption protocol for client side secure data deduplication. In *2017 IEEE international conference on systems, man, and cybernetics (SMC)* (2017), IEEE, pp. 2176–2181.

[4] Ajdari, M., Park, P., Kim, J., Kwon, D., and Kim, J. CIDR: A cost-effective in-line data reduction system for terabit-per-second scale SSD arrays. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2019), IEEE, pp. 28–41.

[5] Alibaba Cloud. Alibaba Cloud Object Storage Service, 2021. https://alibabacloud.com/product/oss.

[6] Allu, Y., Douglis, F., Kamat, M., Prabhakar, R., Shilane, P., and Ugale, R. Can't we all get along? redesigning protection storage for modern workloads. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association.

[7] Allu, Y., Douglis, F., Kamat, M., Shilane, P., Patterson, H., and Zhu, B. Backup to the future: How workload and hardware changes continually redefine data domain file systems. *Computer 50*, 7 (2017), 64–72.

[8] Almeida, J. B., Barbosa, M., Barthe, G., Campagna, M., Cohen, E., Gregoire, B., Pereira, V., Portela, B., Strub, P.-Y., and Tasiran, S. A machine-checked proof of security for AWS key management service. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019), pp. 63–78.

[9] Alon, B. MezzFS — mounting object storage in Netflix's media processing platform, 2021. https://netflixtechblog.com/mezzfs-mounting-object-storage-in-netflixs-media-processing-platform-cda01c446ba.

[10] Alouffi, B., Hasnain, M., Alharbi, A., Alosaimi, W., Alyami, H., and Ayaz, M. A systematic literature review on cloud computing security: Threats and mitigation strategies. *IEEE Access 9* (2021), 57792–57807.

[11] Alzahrani, H. A brief survey of cloud computing. *Global Journal of Computer Science and Technology: Cloud and Distributed, Global Journals Inc.(USA)* (2016), 0975–4172.

[12] Amazon Web Services Inc. Amazon Simple Storage Service User Guide, 2021. https://docs.aws.amazon.com/AmazonS3/latest/userguide/s3-userguide.pdf.

[13] Amazon Web Services Inc. Pinterest on AWS, 2021. https://aws.amazon.com/solutions/case-studies/innovators/pinterest/.

[14] Amazon Web Services, Inc. Amazon EBS pricing, 2023. https://aws.amazon.com/ebs/pricing/.

[15] Amazon Web Services, Inc. Amazon EFS, 2023. https://aws.amazon.com/efs/.

[16] Amazon Web Services, Inc. Amazon Elastic Container Service pricing, 2023. https://aws.amazon.com/ecs/pricing/.

[17] Amazon Web Services, Inc. Amazon S3 pricing, 2023. https://aws.amazon.com/s3/pricing/.

[18] Amazon Web Services, Inc. Amazon S3 storage classes, 2023. https://aws.amazon.com/s3/storage-classes/.

[19] Amazon Web Services, Inc. AWS Direct Connect Locations, 2023. https://aws.amazon.com/directconnect/locations/.

[20] Amazon Web Services, Inc. AWS Lambda - FAQs, 2023. https://aws.amazon.com/lambda/faqs/.

[21] Amazon Web Services, Inc. AWS Snow Family FAQs, 2023. https://aws.amazon.com/snow/faqs/.

[22] Amazon Web Services, Inc. Best practices for cluster configuration, 2023. https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-plan-instances-guidelines.html.

[23] Amazon Web Services, Inc. Billing for interrupted spot instances, 2023. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/billing-for-interrupted-spot-instances.html.

[24] Amazon Web Services, Inc. General purpose instances - network performance, 2023. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/general-purpose-instances.html#general-purpose-network-performance.

[25] Amazon Web Services, Inc. Link aggregation groups, 2023. https://docs.aws.amazon.com/directconnect/latest/UserGuide/lags.html.

[26] Amvrosiadis, G., and Bhadkamkar, M. Identifying trends in enterprise data protection systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (Santa Clara, CA, July 2015), USENIX Association.

[27] Amvrosiadis, G., and Bhadkamkar, M. Getting back up: Understanding how enterprise data backups fail. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, June 2016), USENIX Association.

[28] Amvrosiadis, G., and Bhadkamkar, M. Getting back up: Understanding how enterprise data backups fail. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, June 2016), USENIX Association.

[29] Anderson, P., Black, R., Cerkauskaite, A., Chatzieleftheriou, A., Clegg, J., Dainty, C., Diaconu, R., Drevinskas, R., Donnelly, A., Gaunt, A. L., Georgiou, A., Diaz, A. G., Kazansky, P. G., Lara, D., Legtchenko, S., Nowozin, S., Ogus, A., Phillips, D., Rowstron, A., Sakakura, M., Stefanovici, I., Thomsen, B., Wang, L., Williams, H., and Yang, M. Glass: A new media for a new era? In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)* (Boston, MA, July 2018), USENIX Association.

[30] Andrew Smith, Research Manager; Archana Venkatraman, A. R. D. Enterprise data growth and adoption of cloud applications challenge traditional data protection strategies, 2021. `https://afi.ai/r/US48310921.pdf`.

[31] Arcserve. Arcserve UDP 8.0 is now available, 2021. `support.arcserve.com/s/article/Arcserve-UDP-8-0-Is-Now-Available`.

[32] Armknecht, F., Boyd, C., Davies, G. T., Gjøsteen, K., and Toorani, M. Side channels in deduplication: Trade-offs between leakage and efficiency. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017).

[33] Aronovich, L., Asher, R., Bachmat, E., Bitner, H., Hirsch, M., and Klein, S. T. The design of a similarity based deduplication system. In *Annual Haifa Experimental Systems Conference* (2009).

[34] Athanassoulis, M., and Idreos, S. Design tradeoffs of data access methods. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 2195–2200.

[35] Aye, K. N., and Thein, T. A data rebalancing mechanism for gluster file system. `https://meral.edu.mm/record/3572/files/12046.pdf`.

[36] Backblaze. Backblaze b2, 2021. `https://backblaze.com/b2/cloud-storage.html`.

[37] Backblaze. Backblaze drive stats for q1 2023, 2023. `https://www.backblaze.com/blog/backblaze-drive-stats-for-q1-2023/`.

[38] Bacs, A., Musaev, S., Razavi, K., Giuffrida, C., and Bos, H. DUPEFS: Leaking data over the network with filesystem deduplication side channels. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (Santa Clara, CA, Feb. 2022), USENIX Association.

[39] Bairagi, S. I., and Bang, A. O. Cloud computing: History, architecture, security issues. In *National Conference "CONVERGENCE"* (2015), vol. 2015, p. 28.

[40] Balaji, S., Krishnan, M. N., Vajha, M., Ramkumar, V., Sasidharan, B., and Kumar, P. V. Erasure coding for distributed storage: An overview. *Science China Information Sciences 61* (2018), 1–45.

[41] Balmau, O., Dinu, F., Zwaenepoel, W., Gupta, K., Chandhiramoorthi, R., and Didona, D. SILK: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association.

[42] Bareos. Bareos droplet storage backends, 2021. `https://docs.bareos.org/TasksAndConcepts/StorageBackends.html`.

[43] Barr, J. New Amazon S3 Storage Class – Glacier Deep Archive, 2019. https://aws.amazon.com/blogs/aws/new-amazon-s3-storage-class-glacier-deep-archive/.

[44] Barr, J. AWS Nitro SSD – High Performance Storage for your I/O-Intensive Applications, 2021. https://aws.amazon.com/blogs/aws/aws-nitro-ssd-high-performance-storage-for-your-i-o-intensive-applications/.

[45] Barr, J. Celebrate 15 years of Amazon S3, 2021. https://aws.amazon.com/blogs/aws/amazon-s3s-15th-birthday-it-is-still-day-1-after-5475-days-100-trillion-objects/.

[46] Basu, A., Sampson, J., Qian, Z., and Jaeger, T. Unsafe at any copy: Name collisions from mixing case sensitivities. In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (Santa Clara, CA, Feb. 2023), USENIX Association.

[47] Beaver, D., Kumar, S., Li, H. C., Sobel, J., and Vajgel, P. Finding a needle in Haystack: Facebook's photo storage. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)* (Vancouver, BC, Oct. 2010), USENIX Association.

[48] Belady, L. A., Nelson, R. A., and Shedler, G. S. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM 12*, 6 (1969), 349–353.

[49] Bellare, M., Keelveedhi, S., and Ristenpart, T. Message-locked encryption and secure deduplication. In *Annual international conference on the theory and applications of cryptographic techniques* (2013).

[50] Bertrand, C. ESG research report: The evolution of data protection cloud strategies, 2021. https://www.esg-global.com/research/esg-research-report-the-evolution-of-data-protection-cloud-strategies.

[51] Bhata, S. Introducing azure cool blob storage, 2016. https://azure.microsoft.com/en-us/blog/introducing-azure-cool-storage/.

[52] Black, J. Compare-by-hash: A reasoned analysis. In *2006 USENIX Annual Technical Conference (USENIX ATC 06)* (Boston, MA, May 2006), USENIX Association.

[53] Bögelsack, A., Chakraborty, U., Kumar, D., Rank, J., Tischbierek, J., and Wolz, E. *Introduction to Public Cloud and Hyperscalers*. Apress, Berkeley, CA, 2022, pp. 1–27.

[54] Bolosky, W. J., Corbin, S., Goebel, D., and Douceur, J. R. Single instance storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium* (2000), Seattle, WA, pp. 13–24.

[55] Bolosky, W. J., Douceur, J. R., Ely, D., and Theimer, M. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. *ACM SIGMETRICS Performance Evaluation Review 28*, 1 (2000), 34–43.

[56] Bose, M. Data protection fundamentals: How to backup an Amazon S3 bucket, 2021. https://nakivo.com/blog/how-to-backup-an-amazon-s3-bucket/.

[57] Brett, B. Memory performance, 2023. https://www.intel.com/content/www/us/en/developer/articles/technical/memory-performance-in-a-nutshell.html.

[58] Buyya, R., Broberg, J., and Goscinski, A. M. *Cloud computing: Principles and paradigms*. John Wiley & Sons, 2010.

[59] Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), pp. 143–157.

[60] Camacho-Rodríguez, J., Chauhan, A., Gates, A., Koifman, E., O'Malley, O., Garg, V., Haindrich, Z., Shelukhin, S., Jayachandran, P., Seth, S., et al. Apache hive: From mapreduce to enterprise-grade big data warehousing. In *Proceedings of the 2019 International Conference on Management of Data* (2019).

[61] Cao, Z., Liu, S., Wu, F., Wang, G., Li, B., and Du, D. H. Sliding Look-Back window assisted data chunk rewriting for improving deduplication restore performance. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (Boston, MA, Feb. 2019), USENIX Association.

[62] Cao, Z., Wen, H., Wu, F., and Du, D. H. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *16th USENIX Conference on File and Storage Technologies (FAST 18)* (Oakland, CA, Feb. 2018), USENIX Association.

[63] Carlson, J. *Redis in action*. Simon and Schuster, 2013.

[64] Carver, B., Han, R., Zhang, J., Zheng, M., and Cheng, Y. lambdaFS: A scalable and elastic distributed file system metadata service using serverless functions. *arXiv preprint arXiv:2306.11877* (2023).

[65] Caulk, P. M. The design of a petabyte archive and distribution system for the NASA ECS project. In *NASA. Goddard Space Flight Center, Fourth NASA Goddard Conference on Mass Storage Systems and Technologies* (1994).

[66] Ceph. Erasure coded placement groups, 2016. https://docs.ceph.com/en/mimic/dev/osd_internals/erasure_coding/.

[67] Ceph. Ceph object gateway, 2021. https://docs.ceph.com/en/latest/radosgw/.

[68] Ceph. Rados gateway data layout, 2021. https://docs.ceph.com/en/latest/radosgw/layout/.

[69] Ceph. Balancer plugin, 2022. https://docs.ceph.com/en/mimic/mgr/balancer/.

[70] Ceph. Chapter 3. placement groups (PGS). https://access.redhat.com/documentation/en-us/red_hat_ceph_storage/5/html/storage_strategies_guide/placement_groups_pgs.

[71] Ceph. Placement groups, 2022. https://docs.ceph.com/en/latest/rados/operations/placement-groups/.

[72] Ceph. V0.94.10 HAMMER, 2022. https://docs.ceph.com/docs/master/releases/hammer/.

[73] Chatzieleftheriou, A., Stefanovici, I., Narayanan, D., Thomsen, B., and Rowstron, A. Could cloud storage be disrupted in the next decade? In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)* (July 2020), USENIX Association.

[74] Cheng, G., Guo, D., Luo, L., Xia, J., and Gu, S. Lofs: A lightweight online file storage strategy for effective data deduplication at network edge. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 33*, 10 (2021), 2263–2276.

[75] Chervenak, A., Vellanki, V., and Kurmas, Z. Protecting file systems: A survey of backup techniques. In *Joint NASA and IEEE Mass Storage Conference* (1998).

[76] Ciurana, E. *Developing with google app engine*. Apress, 2009.

[77] CockroachLabs. CockroachDB backup, 2021. `https://cockroachlabs.com/docs/dev/backup`.

[78] Commvault. Public cloud architecture guide for Microsoft Azure, 2020. `https://documentation.commvault.com/commvault/v11/others/pdf/public-cloud-architecture-guide-for-microsoft-azure11-19.pdf`.

[79] Constantinescu, C., Glider, J., and Chambliss, D. Mixing deduplication and compression on active data sets. In *2011 Data Compression Conference* (2011), IEEE, pp. 393–402.

[80] Cooper, P. One of tech's most elusive mysteries: The secret of Amazon Glacier. `https://www.itpro.com/cloud/367950/one-of-techs-most-elusive-mysteries-the-secret-of-amazon-glacier`.

[81] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS) 31*, 3 (2013), 1–22.

[82] Coughlin, T. C1Q 2021 HDD Update. `https://forbes.com/sites/tomcoughlin/2021/05/04/c1q-2021-hdd-update/`.

[83] Dagnaw, G., Hua, W., and Zhou, K. SSD assisted caching for restore optimization in distributed deduplication environment. In *2020 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS)* (2020), IEEE, pp. 1–8.

[84] Dai, Y., Xu, Y., Ganesan, A., Alagappan, R., Kroth, B., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. From WiscKey to bourbon: A learned index for Log-Structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association.

[85] De Guise, P. *Data protection: Ensuring data availability*. CRC Press, 2020.

[86] Debnath, B., Sengupta, S., and Li, J. Chunkstash: Speeding up inline storage deduplication using flash memory. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)* (June 2010), USENIX Association.

[87] DELL EMC. Data Domain deduplication storage systems, 2018. `https://delltechnologies.com/asset/en-us/products/data-protection/technical-support/h11340-datadomain-ss.pdf`.

[88] DELL EMC. Storage S3 in backup, 2021. https://dellemc.com/content/dam/uwaem/production-design-assets/pl-pl/events/forum/2017/presentations/cloud_lanscape4.pdf.

[89] DELL Technologies. PowerProtect DP Series, 2023. https://www.delltechnologies.com/partner/pl-pl/partner/powerprotect-dp.htm.

[90] Distante, F., and Piuri, V. Hill-climbing heuristics for optimal hardware dimensioning and software allocation in fault-tolerant distributed systems. *IEEE transactions on reliability 38*, 1 (1989), 28–39.

[91] Douceur, J. R., and Wattenhofer, R. P. Large-scale simulation of replica placement algorithms for a serverless distributed file system. In *MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (2001).

[92] Douglis, F., Duggal, A., Shilane, P., Wong, T., Yan, S., and Botelho, F. The logic of physical garbage collection in deduplicating storage. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, Feb. 2017), USENIX Association.

[93] Druva. What is backup retention policy? how is it implemented? https://docs.druva.com/Knowledge_Base/inSync/Client/010_FAQ/What_is_Backup_Retention_Policy%3F_How_is_it_implemented%3F.

[94] Dubnicki, C., Gryz, L., Heldt, L., Kaczmarczyk, M., Kilian, W., Strzelczak, P., Szczepkowski, J., Ungureanu, C., and Welnicki, M. HYDRAstor: A scalable secondary storage. In *7th USENIX Conference on File and Storage Technologies (FAST 09)* (San Francisco, CA, Feb. 2009), USENIX Association.

[95] Duggal, A., Jenkins, F., Shilane, P., Chinthekindi, R., Shah, R., and Kamat, M. Data Domain Cloud Tier: Backup here, backup there, deduplicated everywhere! In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association.

[96] Ekwe-Ekwe, N., and Barker, A. Location, location, location: exploring Amazon EC2 spot instance pricing across geographical regions. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (2018).

[97] Elias, N., Shilane, P., Sheinvald, S., and Yadgar, G. Dedupsearch: Two-phase deduplication aware keyword search. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (Santa Clara, CA, Feb. 2022), USENIX Association.

[98] Emma Haruka Iwao, R. M. Introducing open saves: Open-source cloud-native storage for games, 2021. https://cloud.google.com/blog/products/media-entertainment/introducing-open-saves.

[99] Eric Burgener, J. R. High data growth and modern applications drive new storage requirements in digitally transformed enterprises, 2022. https://www.delltechnologies.com/asset/en-my/products/storage/industry-market/h19267-wp-idc-storage-reqs-digital-enterprise.pdf.

[100] Färber, F., May, N., Lehner, W., Grosse, P., Müller, I., Rauhe, H., and Dees, J. The SAP HANA database–an architecture overview. *IEEE Data Eng. Bull. 35*, 1 (2012), 28–33.

[101] Fu, M., Han, S., Lee, P. P., Feng, D., Chen, Z., and Xiao, Y. A simulation analysis of redundancy and reliability in primary storage deduplication. *IEEE Transactions on Computers 67*, 9 (2018), 1259–1272.

[102] Fu, Y., Jiang, H., Xiao, N., Tian, L., and Liu, F. Aa-dedupe: An application-aware source deduplication approach for cloud backup services in the personal computing environment. In *2011 IEEE International Conference on Cluster Computing* (2011), IEEE, pp. 112–120.

[103] Garfinkel, S. The cloud imperative. *Technology Review 114*, 6 (2011), 74–76.

[104] Gervasi, B. Will carbon nanotube memory replace DRAM? *IEEE Micro 39*, 2 (2019), 45–51.

[105] Gholami Taghizadeh, R., Gholami Taghizadeh, R., Khakpash, F., Binesh Marvasti, M., and Asghari, S. A. CA-Dedupe: Content-aware deduplication in SSDs. *The Journal of Supercomputing 76* (2020), 8901–8921.

[106] Global Data Vault. Data backup: Developing an effective data retention. `https://www.globaldatavault.com/blog/data-retention-policy-and-scheduled-backups/`.

[107] Goda, K., and Kitsuregawa, M. The history of storage systems. *Proceedings of the IEEE 100*, Special Centennial Issue (2012), 1433–1440.

[108] Golab, L., Hadjieleftheriou, M., Karloff, H., and Saha, B. Distributed data placement via graph partitioning. *arXiv preprint arXiv:1312.0285* (2013).

[109] Google Cloud. Migrating from Amazon S3 to Cloud Storage, 2021. `https://cloud.google.com/storage/docs/migrating`.

[110] Google Cloud. Request rate and access distribution guidelines, 2021. `https://cloud.google.com/storage/docs/request-rate`.

[111] Google Cloud. Cloud storage pricing, 2023. `https://cloud.google.com/storage/pricing#north-america`.

[112] Google Cloud. Disk pricing, 2023. `https://cloud.google.com/compute/disks-image-pricing#disk`.

[113] Google Cloud. Storage classes, 2023. `https://cloud.google.com/storage/docs/storage-classes#descriptions`.

[114] Goyal, S. Public vs private vs hybrid vs community-cloud computing: a critical review. *International Journal of Computer Network and Information Security 6*, 3 (2014), 20–29.

[115] Graefe, G., et al. Modern b-tree techniques. *Foundations and Trends® in Databases 3*, 4 (2011), 203–402.

[116] Gray, J. Computer technology forecast for virtual observatories. https://www.microsoft.com/en-us/research/publication/computer-technology-forecast-for-virtual-observatories/.

[117] Guidi, G., Ellis, M., Buluç, A., Yelick, K., and Culler, D. 10 years later: Cloud computing is closing the performance gap. In *Companion of the ACM/SPEC International Conference on Performance Engineering* (2021), pp. 41–48.

[118] Guo, F., and Efstathopoulos, P. Building a high-performance deduplication system. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)* (Portland, OR, June 2011), USENIX Association.

[119] Hamandawana, P., Khan, A., Lee, C.-G., Park, S., and Kim, Y. Crocus: Enabling computing resource orchestration for inline cluster-wide deduplication on scalable storage systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 31*, 8 (2020), 1740–1753.

[120] Han, S., Lee, P. P. C., Xu, F., Liu, Y., He, C., and Liu, J. An in-depth study of correlated failures in production SSD-based data centers. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (Feb. 2021), USENIX Association.

[121] Hanushevsky, A., and Nowak, M. Pursuit of a scalable high performance multi-petabyte database. In *16th IEEE Symposium on Mass Storage Systems in cooperation with the 7th NASA Goddard Conference on Mass Storage Systems and Technologies (Cat. No. 99CB37098)* (1999), IEEE, pp. 169–175.

[122] Harari, E. Flash memory — the great disruptor! In *2012 IEEE International Solid-State Circuits Conference* (2012), pp. 10–15.

[123] Harnik, D., Hershcovitch, M., Shatsky, Y., Epstein, A., and Kat, R. Sketching volume capacities in deduplicated storage. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (Boston, MA, Feb. 2019), USENIX Association.

[124] Haynes, T. Network file system (NFS) version 4 minor version 2 protocol. Tech. rep., 2016.

[125] Henson, V. An analysis of compare-by-hash. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)* (Lihue, HI, May 2003), USENIX Association.

[126] Hill, Z., Li, J., Mao, M., Ruiz-Alvarez, A., and Humphrey, M. Early observations on the performance of Windows Azure. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (2010), pp. 367–376.

[127] Hoeck, M., Simpson, N., Rozeman, J., and Donham, J. Magic quadrant for enterprise backup and recovery software solutions, 2022. https://www.gartner.com/en/documents/4017101.

[128] Holt, G. Building a consistent hashing ring, 2011. https://docs.openstack.org/swift/latest/ring_background.html.

[129] Hou, H., Lee, P. P., Shum, K. W., and Hu, Y. Rack-aware regenerating codes for data centers. *IEEE Transactions on Information Theory 65*, 8 (2019), 4730–4745.

[130] HPE. Storageexperts what's new with HPE scalable object storage with Scality RING? https://community.hpe.com/t5/Around-the-Storage-Block/What-s-new-with-HPE-Scalable-Object-Storage-with-Scality-RING/ba-p/7008183.

[131] Hsiao, H.-C., Chung, H.-Y., Shen, H., and Chao, Y.-C. Load rebalancing for distributed file systems in clouds. *IEEE transactions on parallel and distributed systems (TPDS) 24*, 5 (2012), 951–962.

[132] Huang, C., Simitci, H., Xu, Y., Ogus, A., Calder, B., Gopalan, P., Li, J., and Yekhanin, S. Erasure coding in windows azure storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)* (Boston, MA, June 2012), USENIX Association.

[133] Hughes, J. P. Economics of information storage: The value in storing the long tail. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)* (2019), pp. 185–192.

[134] IBM. IBM Cloud Object Storage System™, storage pool expansion guide. https://www.ibm.com/docs/en/STXNRM_3.14.1/coss.doc/pdfs/storagePoolExpansion_bookmap.pdf.

[135] IDC. Data creation and replication will grow at a faster rate than installed storage capacity, 2021. https://idc.com/getdoc.jsp?containerId=prUS47560321.

[136] IDC. Enterprise storage systems market share, 2022. https://idc.com/promo/enterprise-storage-systems.

[137] IDC. Worldwide quarterly purpose built backup appliance tracker, 2023. https://www.idc.com/getdoc.jsp?containerId=IDC_P23469.

[138] IEEE. Ieee standard for information technology–portable operating system interface (posix(tm)) base specifications, issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (2018), 1–3951.

[139] IEEE. IEEE Xplore search results, 2023. https://ieeexplore.ieee.org/search/searchresult.jsp?queryText=distributed%20data%20storage&highlight=true&returnType=SEARCH&matchPubs=true&pageNumber=1&ranges=2018_2023_Year&returnFacets=ALL.

[140] Im, J., Bae, J., Chung, C., Arvind, and Lee, S. Pink: High-speed in-storage key-value store with bounded tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association.

[141] Inc., M. High performance object storage for veeam backup and recovery, 2020. https://blog.min.io/minio-high-performance-object-storage-for-veeam-backup-and-recovery/.

[142] Inc., M. MinIO object storage, 2021. https://min.io.

[143] Intel. Intel® Optane™ business update: What does this mean for warranty and support. https://www.intel.com/content/www/us/en/support/articles/000091826/memory-and-storage.html.

[144] Intel. Intel® Optane™ SSD DC P4800X Series. https://www.intel.com/content/www/us/en/products/sku/97161/intel-optane-ssd-dc-p4800x-series-375gb-2-5in-pcie-x4-3d-xpoint/specifications.html.

[145] Jackowski, A., Gryz, L., Wełnicki, M., Dubnicki, C., and Iwanicki, K. Derrick: A three-layer balancer for self-managed continuous scalability. *ACM Transaction on Storage 19*, 3 (jun 2023). https://doi.org/10.1145/3594543.

[146] Jackowski, A., Ślusarczyk, Ł., Lichota, K., Wełnicki, M., Wijata, R., Kielar, M., Kopeć, T., Dubnicki, C., and Iwanicki, K. Objdedup: High-throughput object storage layer for backup systems with block-level deduplication. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 34*, 7 (2023), 2180–2197, © 2023 IEEE.

[147] Jaffer, S., Mahdaviani, K., and Schroeder, B. Improving the reliability of next generation SSDs using WOM-v codes. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (Santa Clara, CA, Feb. 2022), USENIX Association.

[148] Jayapandian, N., and Md Zubair Rahman, A. Secure deduplication for cloud storage using interactive message-locked encryption with convergent encryption, to reduce storage space. *Brazilian Archives of Biology and Technology 61* (2018).

[149] Ji, X., Yang, B., Zhang, T., Ma, X., Zhu, X., Wang, X., El-Sayed, N., Zhai, J., Liu, W., and Xue, W. Automatic, application-aware i/o forwarding resource allocation. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (Boston, MA, Feb. 2019), USENIX Association.

[150] Johnson, A. W., and Jacobson, S. H. On the convergence of generalized hill climbing algorithms. *Discrete applied mathematics 119*, 1-2 (2002), 37–57.

[151] Kaczmarczyk, M., Barczynski, M., Kilian, W., and Dubnicki, C. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR)* (2012), pp. 1–12.

[152] Kaczmarczyk, M., and Dubnicki, C. Reducing fragmentation impact with forward knowledge in backup systems with deduplication. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR)* (2015), pp. 1–12.

[153] Kadekodi, S., Maturana, F., Athlur, S., Merchant, A., Rashmi, K. V., and Ganger, G. R. Tiger: Disk-Adaptive redundancy without placement restrictions. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association.

[154] Kadekodi, S., Maturana, F., Subramanya, S. J., Yang, J., Rashmi, K. V., and Ganger, G. R. PACEMAKER: Avoiding HeART attacks in storage clusters with disk-adaptive redundancy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association.

[155] Kadekodi, S., Rashmi, K. V., and Ganger, G. R. Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (Boston, MA, Feb. 2019), USENIX Association.

[156] Kadekodi, S., Silas, S., Clausen, D., and Merchant, A. Practical design considerations for wide locally recoverable codes (LRCs). In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (2023).

[157] Kamath, G. M., Prakash, N., Lalitha, V., and Kumar, P. V. Codes with local regeneration and erasure correction. *IEEE Transactions on information theory 60*, 8 (2014), 4637–4660.

[158] Kang, Q., Lee, S., Hou, K., Ross, R., Agrawal, A., Choudhary, A., and Liao, W.-k. Improving mpi collective i/o for high volume non-contiguous requests with intra-node aggregation. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 31*, 11 (2020), 2682–2695.

[159] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. 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* (1997), pp. 654–663.

[160] Keelveedhi, S., Bellare, M., and Ristenpart, T. Dupless: Server-aided encryption for deduplicated storage. In *22nd USENIX Security Symposium (USENIX Security 13)* (2013).

[161] Khan, A., Hamandawana, P., and Kim, Y. A content fingerprint-based cluster-wide inline deduplication for shared-nothing storage systems. *IEEE Access 8* (2020), 209163–209180.

[162] Khan, A., Lee, C.-G., Hamandawana, P., Park, S., and Kim, Y. A robust fault-tolerant and scalable cluster-wide deduplication for shared-nothing storage systems. In *IEEE MASCOTS* (2018).

[163] Khan, R. I. S., Yazdani, A. H., Fu, Y., Paul, A. K., Ji, B., Jian, X., Cheng, Y., and Butt, A. R. SHADE: Enable fundamental cacheability for distributed deep learning training. In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (Santa Clara, CA, Feb. 2023), USENIX Association.

[164] Khattar, R. K., Murphy, M. S., Tarella, G. J., and Nystrom, K. E. *Introduction to Storage Area Network, SAN*. IBM Corporation, International Technical Support Organization, 1999.

[165] Kim, J., Campes, C., Hwang, J.-Y., Jeong, J., and Seo, E. Z-journal: Scalable per-core journaling. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (July 2021), USENIX Association.

[166] KIOXIA. EDSFF - a new form factor for next gen servers and storage. `https://europe.kioxia.com/en-europe/business/ssd/solution/edsff.html`.

[167] Kisous, R., Kolikant, A., Duggal, A., Sheinvald, S., and Yadgar, G. The what, the from, and the to: The migration games in deduplicated systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (Santa Clara, CA, Feb. 2022), USENIX Association.

[168] Kotka, T., Kask, L., Raudsepp, K., Storch, T., Radloff, R., and Liiv, I. Policy and legal environment analysis for e-government services migration to the public

cloud. In *Proceedings of the 9th International Conference on Theory and Practice of Electronic Governance* (2016), pp. 103–108.

[169] Kotlarska, I., Jackowski, A., Lichota, K., Welnicki, M., Dubnicki, C., and Iwanicki, K. InftyDedup: Scalable and cost-effective cloud tiering with deduplication. In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (Santa Clara, CA, Feb. 2023), USENIX Association.

[170] Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data* (2018), pp. 489–504.

[171] Krishnaprasad, P., and Narayamparambil, B. A. A proposal for improving data deduplication with dual side fixed size chunking algorithm. In *2013 Third International Conference on Advances in Computing and Communications* (2013), IEEE, pp. 13–16.

[172] Kruus, E., and Ungureanu, C. Bimodal content defined chunking for backup streams. In *8th USENIX Conference on File and Storage Technologies (FAST 10)* (San Jose, CA, Feb. 2010), USENIX Association.

[173] Kumar, P., and Huang, H. H. Graphone: A data store for real-time analytics on evolving graphs. *ACM Transactions on Storage (TOS) 15*, 4 (2020), 1–40.

[174] Kwon, H., Cho, Y., Khan, A., Park, Y., and Kim, Y. DeNOVA: Deduplication extended nova file system. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2022), IEEE, pp. 1360–1371.

[175] Kwon, M., Lee, S., Choi, H., Hwang, J., and Jung, M. Vigil-kv: Hardware-software co-design to integrate strong latency determinism into log-structured merge Key-Value stores. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association.

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

[177] Learn, M. Understand data store models, 2023. `https://learn.microsoft.com/en-us/azure/architecture/guide/technology-choices/data-store-overview#keyvalue-stores`.

[178] Lee, C., Sim, D., Hwang, J., and Cho, S. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, Feb. 2015), USENIX Association.

[179] Leesakul, W., Townend, P., and Xu, J. Dynamic data deduplication in cloud storage. In *2014 IEEE 8th International Symposium on Service Oriented System Engineering* (2014), IEEE, pp. 320–325.

[180] Leibovici, T. Hierarchical storage from NVMe to tapes. In *Proceedings of the 2022 Workshop on Emerging Open Storage Systems and Solutions for Data Intensive Computing* (2022), pp. 3–4.

[181] Li, H., Zhang, Y., Li, D., Zhang, Z., Liu, S., Huang, P., Qin, Z., Chen, K., and Xiong, Y. Ursa: Hybrid block storage for cloud-scale virtual disks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (2019), pp. 1–17.

[182] Li, J., Qin, C., Lee, P. P. C., and Li, J. Rekeying for encrypted deduplication storage. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2016).

[183] Li, P., Hua, Y., Zuo, P., Chen, Z., and Sheng, J. ROLEX: A scalable RDMA-oriented learned key-value store for disaggregated memory systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (Santa Clara, CA, Feb. 2023), USENIX Association.

[184] Li, Q., Xiang, Q., Wang, Y., Song, H., Wen, R., Yao, W., Dong, Y., Zhao, S., Huang, S., Zhu, Z., et al. More than capacity: performance-oriented evolution of Pangu in Alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (2023).

[185] Li, X., Li, R., Lee, P. P. C., and Hu, Y. OpenEC: Toward unified and configurable erasure coding management in distributed storage systems. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (Boston, MA, Feb. 2019), USENIX Association.

[186] Li, X., Lillibridge, M., and Uysal, M. Reliability analysis of deduplicated and erasure-coded storage. *ACM SIGMETRICS Performance Evaluation Review 38*, 3 (2011), 4–9.

[187] Liang, J. Government cloud: enhancing efficiency of e-government and providing better public services. In *2012 International Joint Conference on Service Sciences* (2012), IEEE, pp. 261–265.

[188] Lillaney, K., Tarasov, V., Pease, D., and Burns, R. The case for dual-access file systems over object storage. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)* (Renton, WA, July 2019), USENIX Association.

[189] Lillibridge, M., Eshghi, K., and Bhagwat, D. Improving restore speed for backup systems that use inline chunk-based deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST 13)* (San Jose, CA, Feb. 2013), USENIX Association.

[190] Lillibridge, M., Eshghi, K., Bhagwat, D., Deolalikar, V., Trezise, G., and Camble, P. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *7th USENIX Conference on File and Storage Technologies (FAST 09)* (San Francisco, CA, Feb. 2009), USENIX Association.

[191] Lim, H., Andersen, D. G., and Kaminsky, M. Towards accurate and fast evaluation of Multi-Stage log-structured designs. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, Feb. 2016), USENIX Association.

[192] Lin, L., Xiao, K., and Liu, W. Utilizing SSD to alleviate chunk fragmentation in de-duplicated backup systems. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)* (2016), IEEE, pp. 616–624.

[193] Lin, X., Douglis, F., Li, J., Li, X., Ricci, R., Smaldone, S., and Wallace, G. Metadata considered harmful to deduplication. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)* (Santa Clara, CA, July 2015), USENIX Association.

[194] Liu, L., Dou, X., and Chen, Y. Intelligent resource scheduling for co-located latency-critical services: A multi-model collaborative learning approach. In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (Santa Clara, CA, Feb. 2023), USENIX Association.

[195] Liu, Z., Bai, Z., Liu, Z., Li, X., Kim, C., Braverman, V., Jin, X., and Stoica, I. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (Boston, MA, Feb. 2019), USENIX Association.

[196] Lu, R., Xu, E., Zhang, Y., Zhu, F., Zhu, Z., Wang, M., Zhu, Z., Xue, G., Shu, J., Li, M., et al. Perseus: A fail-slow detection framework for cloud storage systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (2023).

[197] Luby, M., Watson, M., Gasiba, T., Stockhammer, T., and Xu, W. Raptor codes for reliable download delivery in wireless broadcast systems. In *CCNC* (2006), vol. 6, pp. 192–197.

[198] Luke Stone. Bringing Pokémon GO to life on Google Cloud, 2016. `https://cloud.google.com/blog/products/containers-kubernetes/bringing-pokemon-go-to-life-on-google-cloud`.

[199] Luo, C., and Carey, M. J. Lsm-based storage techniques: a survey. *The VLDB Journal 29*, 1 (2020), 393–418.

[200] Ma, J., Wang, G., and Liu, X. Dedupeswift: object-oriented storage system based on data deduplication. In *IEEE Trustcom/BigDataSE/ISPA* (2016).

[201] Malhotra, J., and Bakal, J. A survey and comparative study of data deduplication techniques. In *2015 International Conference on Pervasive Computing (ICPC)* (2015), IEEE, pp. 1–5.

[202] Mandagere, N., Zhou, P., Smith, M. A., and Uttamchandani, S. Demystifying data deduplication. In *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion* (2008), pp. 12–17.

[203] Maneas, S., Mahdaviani, K., Emami, T., and Schroeder, B. A study of ssd reliability in large scale enterprise storage deployments. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association.

[204] Mao, B., Zhou, J., Wu, S., Jiang, H., Chen, X., and Yang, W. Improving flash memory performance and reliability for smartphones with i/o deduplication. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 38*, 6 (2018), 1017–1027.

[205] Maruti, M. V., and Nighot, M. K. Authorized data deduplication using hybrid cloud technique. In *2015 International Conference on Energy Systems and Applications* (2015), IEEE, pp. 695–699.

[206] Matah, P. Minecraft Earth and Azure Cosmos DB part 1, 2021. `https://azure.microsoft.com/pl-pl/blog/minecraft-earth-and-azure-cosmos-db-part-1-extending-minecraft-into-our-real-world/`.

[207] McKusick, D. M. K. Keynote address: A brief history of the BSD fast filesystem. USENIX Association.

[208] Meghan Rimol, G. Gartner forecasts worldwide public cloud end-user spending to reach nearly $500 billion in 2022, 2022. https://www.gartner.com/en/newsroom/press-releases/2022-04-19-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-500-billion-in-2022.

[209] Meister, D., and Brinkmann, A. dedupv1: Improving deduplication throughput using solid state drives (SSD). In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (2010).

[210] Merenstein, A., Tarasov, V., Anwar, A., Guthridge, S., and Zadok, E. F3: Serving files efficiently in serverless computing. In *Proceedings of the 16th ACM International Conference on Systems and Storage (SYSTOR)* (2023), pp. 8–21.

[211] Meyer, D. T., and Bolosky, W. J. A study of practical deduplication. *ACM Transactions on Storage (ToS) 7*, 4 (2012), 1–20.

[212] Mhalagi, S. R. *Performance evaluation of cloud object storage for big data.* PhD thesis, The University of Texas at San Antonio, 2016.

[213] Microsoft. Azure Blob Storage, 2021. https://azure.microsoft.com/en-us/services/storage/blobs/.

[214] Microsoft. Data deduplication overview, 2022. https://learn.microsoft.com/en-us/windows-server/storage/data-deduplication/overview.

[215] Microsoft. Motivations and business risks in the cost management discipline, 2022. https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/govern/cost-management/business-risks.

[216] Microsoft. Azure storage pricing, 2023. https://azure.microsoft.com/en-us/pricing/details/storage/blobs/#pricing.

[217] MinIO Inc. MinIO surpasses one billion cumulative docker downloads as business surges, 2022. https://www.globenewswire.com/news-release/2022/09/22/2521051/0/en/MinIO-Surpasses-One-Billion-Cumulative-Docker-Downloads-as-Business-Surges.html.

[218] Mohamed, S. M., and Wang, Y. A survey on novel classification of deduplication storage systems. *Distributed and Parallel Databases 39* (2021), 201–230.

[219] Mosley, D. Seagate analyst day. https://s24.q4cdn.com/101481333/files/doc_downloads/2021/2/2021-Seagate-Analyst-Day.pdf.

[220] Muthitacharoen, A., Chen, B., and Mazieres, D. A low-bandwidth network file system. In *Proceedings of the Eighteenth ACM symposium on Operating systems principles* (2001), pp. 174–187.

[221] Nachman, A., Yadgar, G., and Sheinvald, S. Goseed: Generating an optimal seeding plan for deduplicated storage. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association.

[222] Nakivo. Ransomware protection with NAKIVO backup & replication, 2021. https://nakivo.com/ransomware-protection/.

[223] Neal, I., Zuo, G., Shiple, E., Khan, T. A., Kwon, Y., Peter, S., and Kasikci, B. Rethinking file mapping for persistent memory. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (Feb. 2021), USENIX Association.

[224] NEC. 10 reasons to choose NEC HYDRAstor, 2016. https://www.nec.com/en/global/prod/storage/product/backup/file/pdf/10_Reason_To_Choose_Hydrastor.pdf.

[225] NEC. HYDRAstor® HS8-50S, 2020. https://www.necam.com/docs/?id=dcb67490-2973-443a-b81c-13c6600f4627.

[226] NEC. HYDRAstor® OpenStorage Suite, 2021. https://www.nec-enterprise.com/documents/hydrastor-openstorage-suite?id=1525&hash=3da9b8e521d26b75f7e0ac93d06360e860e300e1bb5feb31edb19e4d8e86bfd7.

[227] NEC. FAQ: NEC Storage HS, 2023. https://www.nec.com/en/global/prod/storage/product/backup/hs/faq/index.html.

[228] NEC. NEC Global, 2023. https://www.nec.com/.

[229] NEC. NEC Storage HS: Storage, 2023. https://www.nec.com/en/global/prod/storage/product/backup/index.html.

[230] NEC. NEC Storage HYDRAstor Virtual Appliance, 2023. https://www.nec-enterprise.com/documents/infosheet-hydrastor-va?id=1023&hash=84074cd8aee9ff5501cd03502995693a47192110083b3669588e5e626839fd97.

[231] NetApp Inc. Dreamworks takes imagination to new heights, 2022. https://www.netapp.com/media/72414-CSS-7241_DreamWorks-Hybrid-Cloud_Case-Study.pdf.

[232] Netapp Inc. Cloud tiering documentation. https://docs.netapp.com/us-en/cloud-manager-tiering/pdfs/fullsite-sidebar/Cloud_Tiering_documentation.pdf.

[233] Oh, J., Yoo, S. W., Nam, H., Min, C., and Won, Y. CJFS: Concurrent journaling for better scalability. In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (Santa Clara, CA, Feb. 2023), USENIX Association.

[234] Oh, M., Park, S., Yoon, J., Kim, S., Lee, K.-w., Weil, S., Yeom, H. Y., and Jung, M. Design of global data deduplication for a scale-out distributed storage system. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)* (2018), IEEE, pp. 1063–1073.

[235] OpenStack Foundation. Configure object storage with the S3 API, 2021. https://docs.openstack.org/mitaka/config-reference/object-storage/configure-s3.html.

[236] OpenStack Foundation. OpenStack Object Storage, 2021. https://wiki.openstack.org/wiki/Swift.

[237] OpenStack Foundation. S3/Swift REST API comparison matrix, 2021. https://docs.openstack.org/swift/latest/s3_compat.html.

[238] OpenStack Foundation. Administrator's guide, 2022. https://docs.openstack.org/swift/latest/admin_guide.html.

[239] OpenStack Foundation. Increasing partition power, 2022. https://specs.openstack.org/openstack/swift-specs/specs/in_progress/increasing_partition_power.html.

[240] Oracle. Oracle® ZFS storage appliance administration guide, 2014. https://docs.oracle.com/cd/E37831_01/html/E52872/shares__shares__general__data_deduplication.html.

[241] Oracle. Backing up file-system data. https://docs.oracle.com/cd/E91325_01/OBADM/osb_filesystem_backup.htm.

[242] Palmer, J., Rozeman, J., Mukhyala, C., and Vogel, J. Magic quadrant for distributed file systems and object storage, 2021. ID: G00738148.

[243] Pan, S., Stavrinos, T., Zhang, Y., Sikaria, A., Zakharov, P., Sharma, A., P, S. S., Shuey, M., Wareing, R., Gangapuram, M., Cao, G., Preseau, C., Singh, P., Patiejunas, K., Tipton, J., Katz-Bassett, E., and Lloyd, W. Facebook's Tectonic Filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (Feb. 2021), USENIX Association.

[244] Park, D., Fan, Z., Nam, Y. J., and Du, D. H. A lookahead read cache: improving read performance for deduplication backup storage. *Journal of Computer Science and Technology 32*, 1 (2017), 26–40.

[245] Patterson, D. A., Gibson, G., and Katz, R. H. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data* (1988), pp. 109–116.

[246] Patterson, S. *Learn AWS Serverless Computing: A Beginner's Guide to Using AWS Lambda, Amazon API Gateway, and Services from Amazon Web Services*. Packt Publishing Ltd, 2019.

[247] Paul, M. Cloud object storage deep dive – part two, implementation, 2021. https://www.veeam.com/blog/cloud-object-storage-implementation.html.

[248] Paulo, J., and Pereira, J. A survey and classification of storage deduplication systems. *ACM Computing Surveys (CSUR) 47*, 1 (2014), 1–30.

[249] Paulsen, J. Energy assisted magnetic recording will solve the need for capacity. https://blog.seagate.com/enterprises/energy-assisted-magnetic-recording-will-solve-the-need-for-capacity/.

[250] Pooranian, Z., Chen, K.-C., Yu, C.-M., and Conti, M. RARE: Defeating side channels based on data-deduplication in cloud storage. In *IEEE INFOCOM 2018-IEEE conference on computer communications workshops (INFOCOM wkshps)* (2018), IEEE, pp. 444–449.

[251] Przybylski, S. A. *Cache and memory hierarchy design: a performance directed approach*. Morgan Kaufmann, 1990.

[252] Puzak, T. R. *Analysis of cache replacement-algorithms.* University of Massachusetts Amherst, 1985.

[253] Quantum Corporation. DXi-Series backup appliances, 2021. https://cdn.allbound.com/iq-ab/2021/04/DXi-DS00549A.pdf.

[254] Quantum Corporation. Why Quantum DXi purpose-built appliance?, 2021. https://cdn.allbound.com/iq-ab/2020/02/ST02281A-v01.pdf.

[255] Quinlan, S., and Dorward, S. Venti: A new approach to archival data storage. In *Conference on File and Storage Technologies (FAST 02)* (Monterey, CA, Jan. 2002), USENIX Association.

[256] Rabin, M. O. Fingerprinting by random polynomials. *Technical report* (1981).

[257] Research, I. G. *Global Purpose-built Backup Appliance (PBBA) Market (2023-2028) by Components, Enterprise, System, Industry Vertical, and Geography, Competitiv Analysis, Impact of Covid19, Impact of Economic Slowdown & Impending Recession with Ansoff Analysis.* 2023.

[258] researchandmarkets.com. Public cloud market by service model. https://www.researchandmarkets.com/reports/5739332/public-cloud-market-service-model.

[259] Romański, B., Heldt, Ł., Kilian, W., Lichota, K., and Dubnicki, C. Anchor-driven subchunk deduplication. In *Proceedings of the 4th Annual International Conference on Systems and Storage (SYSTOR)* (2011), pp. 1–13.

[260] Rupprecht, L., Zhang, R., Owen, B., Pietzuch, P., and Hildebrand, D. Swiftanalytics: Optimizing object storage for big data analytics. In *2017 IEEE International Conference on Cloud Engineering (IC2E)* (2017), IEEE, pp. 245–251.

[261] Rydning, D. R.-J. G.-J., Reinsel, J., and Gantz, J. The digitization of the world from edge to core. *Framingham: International Data Corporation 16* (2018).

[262] Ryu, J., Lee, D., Shin, K. G., and Kang, K. Fast application launch on personal computing/communication devices. In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (Santa Clara, CA, Feb. 2023), USENIX Association.

[263] Salah, T., Zemerly, M. J., Yeun, C. Y., Al-Qutayri, M., and Al-Hammadi, Y. Performance comparison between container-based and vm-based services. In *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)* (2017), IEEE, pp. 185–190.

[264] Saltzer, J., and Kaashoek, M. F. *Principles of computer system design: An introduction.* Morgan Kaufmann, 2009.

[265] Sam Fineberg, Thomas Rivera, B. R. 100-year archive survey results: 2007-2017. SNIA. https://www.brighttalk.com/webcast/663/335255.

[266] Samsung. Samsung introduces world's largest capacity (15.36tb) SSD for enterprise storage systems. https://news.samsung.com/global/samsung-now-introducing-worlds-largest-capacity-15-36tb-ssd-for-enterprise-storage-systems.

[267] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and implementation of the sun network filesystem. In *Proceedings of the summer 1985 USENIX conference* (1985), pp. 119–130.

[268] Satyanarayanan, M., Howard, J. H., Nichols, D. A., Sidebotham, R. N., Spector, A. Z., and West, M. J. The ITC distributed file system: Principles and design. *ACM SIGOPS Operating Systems Review 19*, 5 (1985), 35–50.

[269] Server Direct. NGD Systems SSD NVMe 16TB EDSFF (NE1S10-160T1-C). `https://www.serverdirect.nl/en/components/solid-state-drives/nvme/edsff/ngd-systems-ssd-nvme-16tb-edsff-ne1s10-160t1-c/8997`.

[270] Shah, M., Shaikh, M., Mishra, V., and Tuscano, G. Decentralized cloud storage using blockchain. In *2020 4th International conference on trends in electronics and informatics (ICOEI)(48184)* (2020).

[271] Shannon, C. E. A mathematical theory of communication. *The Bell system technical journal 27*, 3 (1948), 379–423.

[272] Sharma, P., Jindal, R., and Borah, M. D. Blockchain-based decentralized architecture for cloud storage system. *Journal of Information Security and Applications 62* (2021), 102970.

[273] Shilane, P., Chitloor, R., and Jonnala, U. K. 99 deduplication problems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)* (Denver, CO, June 2016), USENIX Association.

[274] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (2010).

[275] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)* (2010), Ieee, pp. 1–10.

[276] Singh, G. Leader election in the presence of link failures. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 7*, 3 (1996), 231–236.

[277] Sivasubramanian, S. Amazon dynamoDB: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), pp. 729–730.

[278] Skowron, P., Biskup, M. T., Heldt, Ł., and Dubnicki, C. Fuzzy adaptive control for heterogeneous tasks in high-performance storage systems. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR)* (2013), pp. 1–11.

[279] Soni, M. *Practical AWS Networking: Build and manage complex networks using services such as Amazon VPC, Elastic Load Balancing, Direct Connect, and Amazon Route 53*. Packt Publishing Ltd, 2018.

[280] Srinivasan, K., Bisson, T., Goodson, G. R., and Voruganti, K. idedup: Latency-aware, inline data deduplication for primary storage. In *10th USENIX Conference on File and Storage Technologies (FAST 12)* (San Jose, CA, Feb. 2012), USENIX Association.

[281] STORAGE EXPERTS, H. P. E. HPE Cloud Bank Storage: A data protection solution you can bank on. https://community.hpe.com/t5/Around-the-Storage-Block/HPE-Cloud-Bank-Storage-A-Data-Protection-Solution-You-Can-Bank/ba-p/6965903.

[282] STRUNK, J. D., GOODSON, G. R., SCHEINHOLTZ, M. L., SOULES, C. A., AND GANGER, G. R. Self-securing storage: Protecting data in compromised systems. In *OSDI* (2000).

[283] STRZELCZAK, P., ADAMCZYK, E., HERMAN-IZYCKA, U., SAKOWICZ, J., SLUSARCZYK, L., WRONA, J., AND DUBNICKI, C. Concurrent deletion in a distributed content-addressable storage system with global deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST 13)* (San Jose, CA, Feb. 2013), USENIX Association.

[284] SU, Y., JIN, H., LIU, F., AND LI, W. SACache: Size-aware load balancing for large-scale storage systems. In *Advances in Artificial Intelligence and Security: 7th International Conference, ICAIS 2021, Dublin, Ireland, July 19-23, 2021, Proceedings, Part II 7* (2021), Springer, pp. 89–105.

[285] SUN, Z., KUENNING, G., MANDAL, S., SHILANE, P., TARASOV, V., XIAO, N., ET AL. A long-term user-centric analysis of deduplication patterns. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)* (2016).

[286] SUN, Z. J., KUENNING, G., MANDAL, S., SHILANE, P., TARASOV, V., XIAO, N., AND ZADOK, E. Cluster and single-node analysis of long-term deduplication patterns. *ACM Transactions on Storage (TOS)* (2018).

[287] SURBIRYALA, J., AND RONG, C. Cloud computing: History and overview. In *2019 IEEE Cloud Summit* (2019), pp. 1–7.

[288] SUTTISIRIKUL, K., AND UTHAYOPAS, P. Accelerating the cloud backup using gpu based data deduplication. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)* (2012), IEEE, pp. 766–769.

[289] SYNERGY RESEARCH GROUP. Huge cloud market still growing at 34% per year; amazon, microsoft and google now account for 65% of the total. https://www.srgresearch.com/articles/huge-cloud-market-is-still-growing-at-34-per-year-amazon-microsoft-and-google-now-account-for-65-of-all-cloud-revenues.

[290] TAKAHASHI, C. N., NGUYEN, B. H., STRAUSS, K., AND CEZE, L. Demonstration of end-to-end automation of DNA data storage. *Scientific reports 9*, 1 (2019), 4998.

[291] TANG, L., AND JAIN, N. Join strategies in Hive. *Hive Summit* (2011).

[292] TARASOV, V., MUDRANKIT, A., BUIK, W., SHILANE, P., KUENNING, G., AND ZADOK, E. Generating realistic datasets for deduplication analysis. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)* (Boston, MA, June 2012), USENIX Association.

[293] TERADATA. Teradata using Amazon S3 storage as the backup target, 2021. https://docs.teradata.com/r/CCZ_TZJngXILEsdDOzUoAw/WOt0MU3umZEx7P7mcKH8Eg.

[294] TICHY, W. F., AND RUAN, Z. Towards a distributed file system.

[295] Tsuchiya, Y., and Watanabe, T. Dblk: Deduplication for primary block storage. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)* (2011), IEEE, pp. 1–5.

[296] Tulloch, M. *Introducing Windows Azure for IT Professionals*. Microsoft Press, 2013.

[297] Ultrium LTO. Roadmap. https://www.lto.org/roadmap/.

[298] Ungureanu, C., Atkin, B., Aranya, A., Gokhale, S., Rago, S., Całkowski, G., Dubnicki, C., and Bohra, A. HydraFS: A high-throughput file system for the hydrastor content-addressable storage system. In *8th USENIX Conference on File and Storage Technologies (FAST 10)* (San Jose, CA, Feb. 2010), USENIX Association.

[299] Varma, A., Sahai, V., and Bryant, R. Performance evaluation of a high-speed switching system based on the fibre channel standard. In *[1993] Proceedings The 2nd International Symposium on High Performance Distributed Computing* (1993), pp. 144–151.

[300] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013).

[301] Veeam. Object storage repository, 2021. https://helpcenter.veeam.com/docs/backup/vsphere/object_storage_repository.html.

[302] Veeam Software. Step 7. configure long-term retention, 2021. https://helpcenter.veeam.com/docs/backup/vsphere/backup_job_gfs_vm.html?ver=110.

[303] Veeam Software. 2022 data protection trends, 2022. https://go.veeam.com/wp-data-protection-trends-2022.

[304] Veeam Software. Define job schedule, 2022. https://helpcenter.veeam.com/docs/backup/vsphere/backup_job_schedule_vm.html.

[305] Veeam Software. Short-term retention policy, 2022. https://helpcenter.veeam.com/docs/backup/vsphere/backup_copy_simple_retention.html.

[306] Veeam Software. Data protection trends report 2023, 2023. https://www.veeam.com/wp-data-protection-trends-report-2023.html?wpty.

[307] Veeam Software. Restore operator activity, 2023. https://helpcenter.veeam.com/docs/one/reporter/restore_operator_activity.html?ver=110.

[308] Veritas NetBackup. Veritas NetBackup™ cloud administrator's guide, 2019. https://veritas.com/content/support/en_US/doc/58500769-135186602-0/v126619409-135186602.

[309] Veritas NetBackup. About NetBackup WORM storage support for immutable and indelible data. https://www.veritas.com/support/en_US/doc/25074086-143197427-0/v143250065-143197427.

[310] Veritas NetBackup. Veritas NetBackup™ deduplication guide. https://www.veritas.com/support/en_US/doc/25074086-146020141-0/v145698641-146020141.

[311] Veritas NetBackup. AWS Cloud Storage with Veritas NetBackup. https://www.veritas.com/content/dam/www/en_us/documents/white-papers/WP_aws_cloud_storage_with_netbackup_long_term_retention_solution_V1259.pdf.

[312] Veritas NetBackup. NetBackup: #1 in enterprise backup solutions, 2023. https://www.veritas.com/protection/netbackup.

[313] Villalba, M. Amazon s3 glacier is the best place to archive your data – introducing the s3 glacier instant retrieval storage class, 2021. https://aws.amazon.com/blogs/aws/amazon-s3-glacier-is-the-best-place-to-archive-your-data-introducing-the-s3-glacier-instant-retrieval-storage-class/.

[314] Waldspurger, C. A. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 181–194.

[315] Wallace, G., Douglis, F., Qian, H., Shilane, P., Smaldone, S., Chamness, M., and Hsu, W. Characteristics of backup workloads in production systems. In *10th USENIX Conference on File and Storage Technologies (FAST 12)* (San Jose, CA, Feb. 2012), USENIX Association.

[316] Wang, J., Wang, Y., Wang, H., Ye, K., Xu, C., He, S., and Zeng, L. Towards cluster-wide deduplication based on ceph. In *2019 IEEE International Conference on Networking, Architecture and Storage (NAS)* (2019), IEEE, pp. 1–8.

[317] Wang, L., Zhang, Y., Xu, J., and Xue, G. Mapx: Controlled data migration in the expansion of decentralized object-based storage systems. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association.

[318] Wei, J., Jiang, H., Zhou, K., and Feng, D. MAD2: A scalable high-throughput exact deduplication approach for network backup services. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (2010), IEEE, pp. 1–14.

[319] Wei, Q., Veeravalli, B., Gong, B., Zeng, L., and Feng, D. CDRM: A cost-effective dynamic replication management scheme for cloud storage cluster. In *2010 IEEE international conference on cluster computing* (2010), IEEE, pp. 188–196.

[320] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A scalable, High-Performance distributed file system. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)* (Seattle, WA, Nov. 2006), USENIX Association.

[321] Weil, S. A., Brandt, S. A., Miller, E. L., and Maltzahn, C. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (2006), IEEE, pp. 31–31.

[322] Western Digital. WD Gold Enterprise Class SATA Hard Drive Up To 22TB. https://www.westerndigital.com/products/internal-drives/wd-gold-sata-hdd#WD221KRYZ.

[323] Woo, H., Han, D., Ha, S., Noh, S. H., and Nam, B. On stacking a persistent memory file system on legacy file systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (Santa Clara, CA, Feb. 2023), USENIX Association.

[324] Wu, F., Fan, Z., Yang, M.-C., Zhang, B., Ge, X., and Du, D. H. Performance evaluation of host aware shingled magnetic recording (HA-SMR) drives. *IEEE Transactions on Computers 66*, 11 (2017), 1932–1945.

[325] Wu, K., Tu, K., Patel, Y., Sen, R., Park, K., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. Nyxcache: Flexible and efficient multi-tenant persistent memory caching. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (Santa Clara, CA, Feb. 2022), USENIX Association.

[326] Wu, S., Mao, B., Jiang, H., Luan, H., and Zhou, J. PFP: improving the reliability of deduplication-based storage systems with per-file parity. *IEEE Transactions on Parallel and Distributed Systems 30*, 9 (2019), 2117–2129.

[327] Wulf, F., Lindner, T., Strahringer, S., and Westner, M. IaaS, PaaS, or SaaS? The why of cloud computing delivery model selection: Vignettes on the post-adoption of cloud computing. In *Proceedings of the 54th Hawaii International Conference on System Sciences, 2021* (2021), pp. 6285–6294.

[328] Xia, N., Tian, C., Luo, Y., Liu, H., and Wang, X. UKSM: Swift memory deduplication via hierarchical and adaptive memory region distilling. In *16th USENIX Conference on File and Storage Technologies (FAST 18)* (Oakland, CA, Feb. 2018), USENIX Association.

[329] Xia, W., Jiang, H., Feng, D., Douglis, F., Shilane, P., Hua, Y., Fu, M., Zhang, Y., and Zhou, Y. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE 104*, 9 (2016), 1681–1710.

[330] Xia, W., Jiang, H., Feng, D., Tian, L., Fu, M., and Zhou, Y. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation 79* (2014), 258–272.

[331] Xia, W., Zhou, Y., Jiang, H., Feng, D., Hua, Y., Hu, Y., Liu, Q., and Zhang, Y. Fastcdc: A fast and efficient content-defined chunking approach for data deduplication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, June 2016), USENIX Association.

[332] Xie, W., and Chen, Y. Elastic consistent hashing for distributed storage systems. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2017), IEEE, pp. 876–885.

[333] Xu, E., Zheng, M., Qin, F., Xu, Y., and Wu, J. Lessons and actions: What we learned from 10k SSD-related storage system failures. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association.

[334] Yang, T.-Y., Liang, Y., and Yang, M.-C. Practicably boosting the processing performance of bfs-like algorithms on semi-external graph system via i/o-efficient graph ordering. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (Santa Clara, CA, Feb. 2022), USENIX Association.

[335] Yu, C.-M. Counteracting side channels in cross-user client-side deduplicated cloud storage. *IEEE Internet of Things Journal* (2023).

[336] Yuan, H., Chen, X., Li, J., Jiang, T., Wang, J., and Deng, R. H. Secure cloud data deduplication with efficient re-encryption. *IEEE Transactions on Services Computing 15*, 1 (2019), 442–456.

[337] Zerto. Maximize recovery achieve your best RTOs and RPOs. `https://www.zerto.com/wp-content/uploads/2020/08/Fastest-RTO-and-RPO-in-the-Industry_Guide.pdf`.

[338] Zerto. Deploy & configure Zerto long-term retention Amazon S3. `https://www.zerto.com/page/deploy-configure-zerto-long-term-retention-amazon-s3/`.

[339] Zhang, H., Chen, G., Ooi, B. C., Tan, K.-L., and Zhang, M. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering 27*, 7 (2015), 1920–1948.

[340] Zhang, J., Lin, M., Pan, Y., and Xu, Z. Crftl: cache reallocation-based page-level flash translation layer for smartphones. *IEEE Transactions on Consumer Electronics* (2023).

[341] Zhang, M., Han, S., and Lee, P. P. A simulation analysis of reliability in erasure-coded data centers. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)* (2017), IEEE, pp. 144–153.

[342] Zhang, Y., Jiang, H., Feng, D., Xia, W., Fu, M., Huang, F., and Zhou, Y. AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *2015 IEEE Conference on Computer Communications (INFOCOM)* (2015), IEEE, pp. 1337–1345.

[343] Zhang, Y., Xia, W., Feng, D., Jiang, H., Hua, Y., and Wang, Q. Finesse: Fine-grained feature locality based fast resemblance detection for post-deduplication delta compression. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 121–128.

[344] Zhao, N., Albahar, H., Abraham, S., Chen, K., Tarasov, V., Skourtis, D., Rupprecht, L., Anwar, A., and Butt, A. R. DupHunter: Flexible high-performance deduplication for docker registries. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association.

[345] Zheng, Q., Chen, H., Wang, Y., Zhang, J., and Duan, J. Cosbench: Cloud object storage benchmark. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering* (2013), pp. 199–210.

[346] Zheng, S., Hoseinzadeh, M., and Swanson, S. Ziggurat: A tiered file system for Non-Volatile main memories and disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (Boston, MA, Feb. 2019), USENIX Association.

[347] Zhou, S., Xu, E., Wu, H., Du, Y., Cui, J., Fu, W., Liu, C., Wang, Y., Wang, W., Sun, S., Wang, X., Feng, B., Zhu, B., Tong, X., Kong, W., Liu, L., Wu, Z., Wu, J., Luo, Q., and Wu, J. SMRSTORE: A storage engine for cloud object storage on hm-smr drives. In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (Santa Clara, CA, Feb. 2023), USENIX Association.

[348] Zhu, B., Li, K., and Patterson, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *6th USENIX Conference on File and Storage Technologies (FAST 08)* (San Jose, CA, Feb. 2008), USENIX Association.

[349] Zissis, D., and Lekkas, D. Securing e-government and e-voting with an open cloud computing architecture. *Government Information Quarterly 28*, 2 (2011), 239–251.

[350] Zou, Q., and Mao, B. Revisiting temporal storage i/o behaviors of smartphone applications: Analysis and synthesis. In *2022 IEEE International Symposium on Workload Characterization (IISWC)* (2022), IEEE, pp. 215–227.

[351] Zuo, P., Hua, Y., Zhao, M., Zhou, W., and Guo, Y. Write deduplication and hash mode encryption for secure nonvolatile main memory. *IEEE Micro 39*, 1 (2018), 44–51.