ADVANCED DATA DEDUPLICATION TECHNIQUES AND THEIR APPLICATION

DIRK MEISTER

Dissertation
zur Erlangung des Grades
"Doktor der Naturwissenschaften"
am Fachbereich Physik, Mathematik und Informatik
der Johannes Gutenberg-Universität
in Mainz

März 2013

Dirk Meister: *Advanced Data Deduplication Techniques and their Application*

D77 Dissertation Johannes Gutenberg University Mainz

ABSTRACT

Data deduplication describes a class of approaches that reduce the storage capacity needed to store data or the amount of data that has to be transferred over a network. These approaches detect coarse-grained redundancies within a data set, e.g. a file system, and remove them.

One of the most important applications of data deduplication are backup storage systems where these approaches are able to reduce the storage requirements to a small fraction of the logical backup data size.

This thesis introduces multiple new extensions of so-called fingerprinting-based data deduplication. It starts with the presentation of a novel system design, which allows using a cluster of servers to perform exact data deduplication with small chunks in a scalable way.

Afterwards, a combination of compression approaches for an important, but often overlooked, data structure in data deduplication systems, so called block and file recipes, is introduced. Using these compression approaches that exploit unique properties of data deduplication systems, the size of these recipes can be reduced by more than 92% in all investigated data sets. As file recipes can occupy a significant fraction of the overall storage capacity of data deduplication systems, the compression enables significant savings.

A technique to increase the write throughput of data deduplication systems, based on the aforementioned block and file recipes, is introduced next. The novel Block Locality Caching (BLC) uses properties of block and file recipes to overcome the chunk lookup disk bottleneck of data deduplication systems. This chunk lookup disk bottleneck either limits the scalability or the throughput of data deduplication systems. The presented BLC overcomes the disk bottleneck more efficiently than existing approaches. Furthermore, it is shown that it is less prone to aging effects.

Finally, it is investigated if large HPC storage systems inhibit redundancies that can be found by fingerprinting-based data deduplication. Over 3 PB of HPC storage data from different data sets have been analyzed. In most data sets, between 20 and 30% of the data can be classified as redundant. According to these results, future work in HPC storage systems should further investigate how data deduplication can be integrated into future HPC storage systems.

This thesis presents important novel work in different area of data deduplication research.

ZUSAMMENFASSUNG

In dieser Doktorarbeit werden mehrere Forschungsergebnisse über verschiedene Aspekte von Datendeduplizierungssystemen vorgestellt. Datendeduplizierung beschreibt eine Klasse von Ansätzen, die es erlauben, die zur Speicherung von Daten notwendig Speicherkapazität zu reduzieren. Datendeduplizierungstechniken werden weiterhin eingesetzt, um die Menge der Daten, die über ein Netzwerk transferiert werden müssen, zu reduzieren. Diese Ansätze erkennen grob-körnige Redundanzen innerhalb einer Datenmenge und entfernen diese.

Ein wichtiges Anwendungsgebiet für Datendeduplizierungstechniken sind Speichersysteme zur Datensicherung, da hier die zu speichernde Datenmenge auf einen kleinen Teil der logischen Backupdaten reduziert werden kann. Dieses ermöglicht oftmals erhebliche Kosteneinsparungen im Betrieb von Datensicherungssystemen.

In dieser Arbeit werden mehrere neue Techniken für Fingerabdruck-basierende Datendeduplizierungssysteme vorgestellt. Die Arbeit startet mit der Vorstellung eines neuartigen Designs, um einen Cluster von Rechnern zu verwenden, der exakte Deduplizierung mit kleinen Chunks ermöglicht. Anschließend wird eine Kombination von Komprimierungstechniken eingeführt, die die Größe einer wichtigen und speicher-intensiven Datenstruktur von Datendeduplizierungssystemen, den sogenannten Block- und Dateirezepten, reduzieren.

Eine weitere neuartige Technik, die Block- und Dateirezepte verwendet, wird als nächstes vorgestellt. Das Blocklokalitätscaching (BLC) verwendet Eigenschaften von diesen Rezepten um den Festplattenflaschenhals von Datendeduplizierungssystemen zu überwinden. Dieser Flaschenhals limitiert entweder die Skalierbarkeit oder den Datendurchsatz von Datendeduplizierungssystemen. Der BLC-Ansatz ist dabei effizienter als existierende Ansätze. Es wird weiterhin gezeigt, dass der BLC-Ansatz weniger anfällig für Alterungserscheinungen als existierende Ansätze ist. Daher nimmt mit dem BLC-Ansatz der Durchsatz auch bei langlebigen Datendeduplizierungssystemen nicht ab.

Im letzten Teil dieser Arbeit wird untersucht, ob die Speichersysteme von großen Hochleistungsrechnern (HPC) Redundanzen von der Art beinhaltet, die von Fingerabdruck-basierender Datendeduplizierung vermieden werden können. Dazu wurden mehr als 3 PB Daten aus verschiedenen Rechenzentren analysiert. Das Ergebnis lässt darauf schließen, dass in den meisten Rechenzentren zwischen 20 und 30% der Daten als redundant klassifiziert werden können. Dieses zeigt, dass die zukünftige Forschung im Bereich der Speichersysteme von Hochleistungsrechensystemen beinhalten sollte, wie Datendeduplizierung in diesen Systemen eingesetzt werden kann.

PERSONAL PUBLICATIONS

[BBF+13] Petra Berenbrink, André Brinkmann, Tom Friedetzky, Dirk Meister, and Lars Nagel. Distributing storage in cloud environments. In *Proceedings of the High-Performance Grid and Cloud Computing Workshop (HPGC), to appear*. IEEE, May 2013.

[BGKM11] André Brinkmann, Yan Gao, Miroslaw Korzeniowski, and Dirk Meister. Request load balancing for highly skewed traffic in P2P networks. In *Proceedings of the 6th IEEE International Conference on Networking, Architecture, and Storage (NAS)*. IEEE, 2011.

[GMB10] Yan Gao, Dirk Meister, and André Brinkmann. Reliability analysis of declustered-parity RAID 6 with disk scrubbing and considering irrecoverable read errors. In *Proceedings of the 5th IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 126–134. IEEE, 2010.

[HSG+09] André Höing, Guido Scherp, Stefan Gudenkauf, Dirk Meister, and André Brinkmann. An orchestration as a service infrastructure using Grid technologies and WS-BPEL. In *Service-Oriented Computing*, volume 5900 of *Lecture Notes in Computer Science*, pages 301–315. Springer Berlin Heidelberg, 2009.

[KMB+11] Matthias Keller, Dirk Meister, André Brinkmann, Christian Terboven, and Christian Bischof. eScience cloud infrastructure. In *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 188–195. IEEE, 2011.

[KMBE12] Jürgen Kaiser, Dirk Meister, André Brinkmann, and Sascha Effert. Design of an exact data deduplication cluster. In *Proceedings of the 28th IEEE Conference on Mass Storage Systems and Technologies (MSST)*. IEEE, May 2012.

[KMHB12] Jürgen Kaiser, Dirk Meister, Tim Hartung, and André Brinkmann. ESB: Ext2 split block device. In *Proceedings of the 18th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 181–188, December 2012.

[LMB10] Paul Lensing, Dirk Meister, and André Brinkmann. hashFS: Applying hashing to optimize file systems for small file reads. In *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, pages 33–42. IEEE, 2010.

[MB09] Dirk Meister and André Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *Proceedings of the 2nd Israeli Experimental Systems Conference (SYSTOR)*, pages 8:1–8:12. ACM, 2009.

[MB10a] Dirk Meister and André Brinkmann. dedupv1: Improving deduplication throughput using solid state drives. In *Proceedings of the 26th IEEE Conference on Mass Storage Systems and Technologies (MSST)*. IEEE, May 2010.

[MBS13] Dirk Meister, André Brinkmann, and Tim Süß. File recipe compression in data deduplication systems. In *Proceedings of 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 175–182. USENIX, 2013.

[MKB+12] Dirk Meister, Jürgen Kaiser, André Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. A study on data deduplication in HPC storage systems. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 7:1–7:11. IEEE, November 2012.

ACKNOWLEDGMENTS

[The acknowledgments have been removed from the online version of the thesis due to regulations of the Johannes Gutenberg University, Mainz]

CONTENTS

1

INTRODUCTION

Data deduplication describes a class of approaches that reduce the storage capacity needed to store data or the amount of data that has to be transferred over a network. The common trait of all deduplication approaches is that they search coarse-grained redundancies within a large search scope, usually all data that has already been stored before. The search scope contrasts deduplication from classical data compression, which is usually considered to search redundancies only within a limited window, e.g., LZ77-based compression [ZL77].

Figure 1 shows an iconic visualization of data deduplication, in which multiple files are fed into the deduplication system. Each of them consists of a series of colored data blocks. In data deduplication, these data blocks are called "chunks" by convention. Chunks of the same color have the same content. The data deduplication system processes the files so that only one chunk of each color is emitted.

One important application domain for data deduplication is backup storage. The prevailing storage media for backup systems has been tape [KA01]. Over time, disk storage became less costly, but still tape storage has been considered cheaper. Disk-2-disk (D2D) backup became more cost-effective by using data deduplication techniques. Data deduplication reduces the storage requirements by a factor of around 10 and more, as large parts of a backup is identical to the previous backup [Big07].

Therefore, backup data has an exceptionally high data redundancy, which can be exploited by data deduplication. Another reason for favoring D2D backup instead of tape

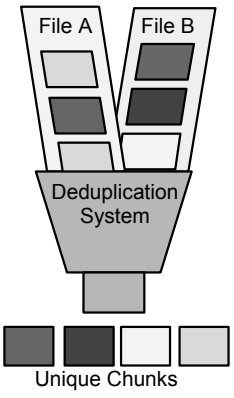


Figure 1: Iconic visualization of data deduplication (after [Big07]).

is the higher restore performance [ZLP08, LEB+09]. In addition to backup systems, data deduplication has been successfully used in various other fields as virtual machine storage, WAN replication, and primary storage [CAVL09, SHWH12, WMR12, SBGV12].

Besides its commercial success, data deduplication has also been a hot topic in the research community. Most storage conferences in the last years have featured deduplication sessions. In the recent years, researchers focused on different aspects of data deduplication:

THROUGHPUT: A major research focus are approaches to achieve a high write throughput. Especially the problem of the "chunk lookup disk bottleneck" [LEB+09] proved to be important with Zhu et al.'s work an early breakthrough [ZLP08, LEB+09, MB10a, BELL09, GE11, XJFH11, DSL10, YJF+10].

CLUSTERING: Researchers are working on improving the scalability by using a cluster of cooperating nodes for data deduplication [BELL09, WJZF10, CAVL09, DGH+09, HL04, DDL+11, YJF+10] to overcome the throughput and fault tolerance limitations of single-node systems.

OTHER TYPES OF STORAGE WORKLOAD: Beyond the usage of data deduplication in backup and archival storage systems, researchers explore how to use data deduplication for other types of storage workloads, e.g., primary storage [SBGV12, WMR12, KR10, TW11, LCGC12] and storage and migration of virtual machine images [JM09, CAVL09, RMP11, ZHMM10].

ADVANCED CHUNKING: The primary methods for splitting blocks, files, or a data stream into smaller chunks, which are then used as the unit of redundancy detection, are static chunking and content-defined chunking. Researchers work on advanced chunking approaches to improve on these primary methods [KUD10, LJD10, ET05].

This thesis presents research results, which provide a significant contribution to the first three aspects mentioned above. It also tackles research issues in the largely unexplored area of block and file recipe compression.

1.1 CONTRIBUTIONS

In this thesis, I present multiple novel contributions to the state of the art of data deduplication research.

1. I introduce the design of the di-dedupv1 data deduplication system for backup workloads. The design is able to scale exact data deduplication systems with small chunk sizes to at least a few dozen of cluster nodes. The core contributions that enable the scalability are the separation of the node that stores the chunk index entry and the node that stores the chunk data and a combination of the storage architecture and the communication protocol that allows an efficient usage of the system interconnect. A model for the communication behavior is presented that allows predicting the scalability of the design beyond the scale that has been evaluated using the prototype implementation.

2. In the research area of extending the scope of data deduplication to new workload types, I contribute a study on the data deduplication potential of HPC storage data.

This is the first study focusing on the question if it is worthwhile to apply data deduplication in HPC storage systems.

3. In the research area focusing on approaches to overcome the chunk lookup disk bottleneck, I introduce a new approach called Block Locality Caching (BLC). The approach uses the similarity between backup data and block recipes to predict future chunk requests. A trace-based simulation shows that this exact approach needs less IO operations for a backup data set than existing approaches. A prototype implementation of the new approach is used to show that it achieved a high throughput in practice.

4. Finally, I contribute novel compression approaches that compress file and block recipes. In backup situations with very high deduplication rates and long retention periods these data structures can grow to a significant fraction of the overall storage requirements of a data deduplication backup system. In these situations, recipe compression enables important savings.

1.2 STRUCTURE

The thesis is structured in five parts and the appendix:

PART I: GENERAL BACKGROUND: The first part provides the background for the thesis. After an introduction into the basics and history of data deduplication, the general related work is discussed.

Also, the part will introduce the dedupv1 data deduplication system and FS-C deduplication analysis tool suite. Both provide important foundations for the simulations and experiments presented in this thesis.

PART II: DI-DEDUPV1 DATA DEDUPLICATION CLUSTER: This part contributes a novel design for a clustered data deduplication system. The design is able to provide a scalable data deduplication while providing exact data deduplication with small chunk sizes.

PART III: BLOCK AND FILE RECIPE MANAGEMENT AND USAGE: This part introduces new techniques to manage and use block and file recipes in data deduplication systems.

First, new techniques that compress block and file recipes are introduced. Then, the recipes are used in a novel caching scheme to overcome the chunk lookup disk bottleneck. The novel caching scheme is capturing the properties of the previous backup run better than existing approaches. Therefore, it used less IO operations to perform an exact-data deduplication backup than the alternatives.

PART IV: DATA DEDUPLICATION IN HPC: Before this part, the thesis concentrated on data deduplication as a technique to enable D2D backup systems. In this part, it is shown that HPC storage systems store data with a significant amount of redundancy, which data deduplication can eliminate. This result indicates that data deduplication is also an interesting technique for HPC storage systems.

PART V: CONCLUSION: The last part summarizes the thesis and its contributions. It also presents possible directions for future research.

APPENDIX: The appendix contains additional material that completes the presentation of the evaluation of Part III and Part IV. It also explains how the trace data sets and the source code used in this thesis can be requested.

Part I

GENERAL BACKGROUND

2

BACKGROUND AND RELATED WORK

This chapter provides background information for this thesis and an overview of the general state of the art. Furthermore, it discusses techniques and tools that are used throughout the thesis.

Section 2.1 provides a brief introduction to data deduplication. Afterwards, Section 2.2 describes the general related work in more detail.

The last sections of this chapter contain the descriptions of two systems that provide the foundation for the following parts of this thesis. Section 2.3 describes the dedupv1 data deduplication system. While dedupv1 itself is not within the scope of this thesis, it is a prerequisite for some parts of this thesis (especially Part II and Part III). Section 2.4 describes the FS-C tool suite. This tool suite is also used in different parts of this thesis to analyze the deduplication ratios of data sets and to perform trace-based simulations.

2.1 INTRODUCTION TO DATA DEDUPLICATION

There are two main approaches for data deduplication storage systems: fingerprinting-based and delta-based data deduplication. Nowadays, fingerprinting-based deduplication is prevalent in practice and research and this thesis deals exclusively with this type. The baseline algorithm for fingerprinting-based data deduplication, shown in Figure 2, is the sequence of chunking, duplicate detection and storage:

CHUNKING: In the chunking step, the data is split into non-overlapping blocks, called "chunks". Each chunk is later processed independently from other chunks.

Different approaches have been proposed to generate chunks. The two most relevant chunking strategies are "static chunking" and "content-defined chunking" [PP04]. Static chunking splits data into chunks that always have equal size. Usually the chunk size is a multiple of a disk sector or the block size of the file system. Static chunking is also known as "fixed-size chunking" [JM09, MYW11, Jon10, CGC11], "fixed block chunking" [MB12, MZSU08], and "fixed chunking" [TMB+12].

Content-defined chunking is usually preferred in backup systems because it is not prone to the "boundary-shifting effect" [ET05], which reduces the redundancy found by data deduplication systems. This effect occurs when data is stored multiple times but slightly shifted, e.g., because other data was inserted into the data stream. In these cases, a system using static chunking is unable to detect the redundancies because the chunks are not identical, whereas with content-defined chunking the chunking eventually realigns with the content and the same chunks are before are

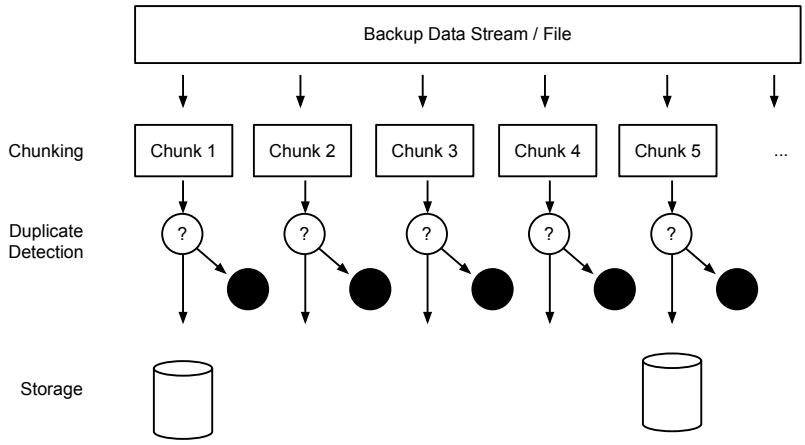


Figure 2: Illustration of the three steps of fingerprinting-based deduplication.

created. For backup workloads, content-defined chunking has been shown to produce higher deduplication ratios [MB09, PP04]. Therefore, it is used in most deduplication systems for backup workloads [ZLP08, WJZF10, MB10a, BELL09, LEB13].

The most common content-defined chunking approach calculates a hash for all sliding windows over the data stream [MCM01, CMN02]. Then, anchors are searched by fixing the positions where the equation $f \% n = c$ holds with f being the hash f over the window and c chosen appropriately. A new chunk is created at each anchor position. The expected chunk size is equal to n. In enterprise backup systems often use an expected chunk size between 4 and 16 KB[1] [ZLP08, LEB13, WJZF10, MB10a].

Content-defined chunking is usually attributed to Brin et al.'s research on copy detection for digital documents [BDGM95]. In Muthitacharoen et al.'s "Low Bandwidth File system" (LBFS) [MCM01], the approach was refined to use Rabin's fingerprinting method [Rab81, Bro93, CL01]. This is still commonly used as a rolling fingerprinting method in content-defined chunking. To my best knowledge, Policroniades and Pratt were the first to use the term "content-defined chunking" [PP04]. The approach is also called "variable-size chunking" [LEB+09, BELL09, JM09, MYW11, KUD10] and "Rabin chunking" [MB12, XJFH11] and also (incorrectly) "Rabin fingerprinting" [TKS+03, TS07, WKB+08, PBM09].

Other researchers investigated alternative chunking methods to reduce the metadata overhead while maintaining a good deduplication [KUD10, LJD10] or improve the deduplication results [FEC05].

While in most publications the term "chunks" is used, Zhu et al. call them "segments" [ZLP08].

DUPLICATE CLASSIFICATION: In this step, the deduplication system classifies if a chunk is redundant, i.e., it has already been stored. Only if a chunks is not redundant, the chunk is added to the system and its data is stored.

1 In this thesis the symbol KB is used its binary form as 2^{10} bytes when the symbol is related to storage units [iee86]. MB, GB, TB, and PB are used analogously.

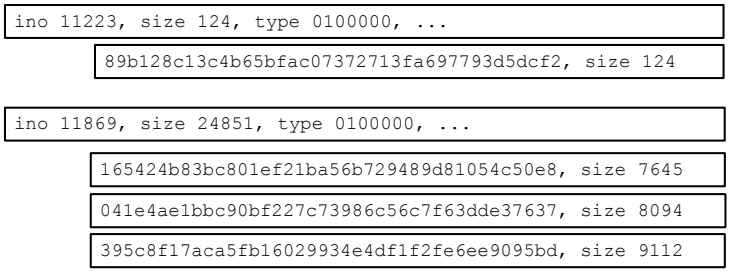


Figure 3: Illustration of file recipes.

Most fingerprinting-based deduplication systems use an approach called "compare-by-hash" [Heno3, Blao6]. Here, all chunks are fingerprinted using a cryptographic fingerprinting method. Also, an index, often called "chunk index", is maintained containing the fingerprint of all already stored chunks. If the fingerprint of a chunk can be found in the chunk index, it is assumed that the data of the already stored chunk and the currently processed chunk are identical and the chunk is classified as already stored. Other names for the chunk index are "segment index" [ZLP08], "fingerprint database" [WKB+08], and "fingerprint index" [WMR12, SHWH12].

The compare-by-hash method, which is not only used for data deduplication, has been critically discussed in the past (see [Heno3, Blao6, HH05]). One of the reasons for the critic is that if two data blocks with different contents are hashed to the same fingerprint, one of these data blocks is misclassified as redundant and a (silent) data loss occurs. However, under reasonable assumptions the probability for such a compare-by-hash induced data loss is orders of magnitude lower than other sources of data loss, e.g., undetected errors in the disk, network, or memory systems [ZLP08, QD02].

STORAGE: All chunks classified as new are stored by the deduplication system. One common approach for storing chunks is to group chunks together and store them in a "container" data structure [ZLP08, WJZF10, MB10a, DGH+09, LEB+09]. The Venti system uses a data structure similar to a container, called "arena" [QD02].

In Zhu et al.'s approach, container play a critical role as the chunks are grouped together to capture a temporal locality and to use the locality to overcome the chunk lookup disk bottleneck [ZLP08].

Besides the chunk index, any deduplication system has an additional persistent data structure to store the information that is necessary to recover the file contents (if the interface is a file system) or the block contents (if the interface is a block device).

In file-based data deduplication systems, this is called a "file recipe" [WJZF10, BELL09, TJF+11, LPB11, WDQ+12, UAA+10, ELW+07, SHWH12, LEB13, BSGH13]. The concept of file recipes was introduced by Tolia et al. [TKS+03].

A file recipe contains a list of chunk identifiers. where each chunk identifier is the cryptographic fingerprint of a chunk.

Using the file recipe, the original file contents can be reconstructed by reading the chunk data of the fingerprints and concatenating their data in the order given by the recipe. Figure 3 shows two exemplary file recipes. The file with inode number 11223 has a total

size of 124 bytes and consists of a single chunk with the given fingerprint. The content of the file with inode number 11869 is stored in three chunks and has a size of 24,851 bytes.

The fingerprints in a file recipe identify stored data similar to block pointers in standard file systems, which point to disk blocks. One difference to the file system block pointers is the size of pointer data type. Usually, the size of data block pointers does not exceed 64 bits.

Most data deduplication systems, where it is publicly known how the mapping from a file to the deduplicated contents is stored, use file recipes that store cryptographic fingerprints. In these systems the used fingerprints have a size of at least 20 bytes [WJZF10, BELL09, TJF+11, LPB11, WDQ+12, UAA+10, BSGH13, LEB13].

In block-based systems, a data structure similar to file recipes is used to map from disk blocks to chunks. This data structure is called the "block recipe" in the dedupv1 system. All block recipes are stored in the "block index".

The baseline algorithm presented above performs "inline" data deduplication [LEB13, SBGV12, DSL10]. Here, the redundancy is detected and removed while the data is written to the system and only data that has been classified as new is actually stored. An inline data deduplication system are also denoted as "online" systems [Jon10, DWG11].

Some deduplication systems store all written data and then remove the redundancies afterwards, e.g., in the background [HL04, CAVL09]. These systems are "post-processing" [YJF+10, MB12, Dut08] or "offline" [Jon10, DWG11] data deduplication systems.

Deduplication systems that remove all duplicate chunks are called "exact deduplication systems". The term "exact deduplication" was first used by Wei et al. and is defined as a type of deduplication that "completely eliminates deduplicates" [WJZF10]. Wei et al. call the class of deduplication approaches that do not detect all duplicates "approximate deduplication".

The baseline algorithm leaves important issues open that need to be resolved in concrete deduplication systems. One of these issues is the "disk bottleneck" [ZLP08] or "chunk lookup disk bottleneck" [LEB+09], which limits the throughput of deduplication systems in most situations because the baseline approach needs to lookup each chunk fingerprint in the chunk index.

The size of the chunk index usually prohibits storing it entirely in main memory. For example, the JumboStore system stores the entire chunk index in an in-memory hash table using MD5 fingerprints [ELW+07]. It needs 1,500 GB RAM to hold the chunk index for a deduplication system with 100 TB disk storage if 8 KB chunks are used [LEB+09].

The chunk index is often stored on HDD storage. However, HDD storage is not able to cope with the high random read requirements that are necessary to lookup the necessary chunk fingerprints per second. Even, for a rather low target throughput of 100 MB/s, more than 12,000s of chunk fingerprints need to be checked per second.

Different approaches have been proposed to overcome this bottleneck. One approach is the usage of Solid State Disks (SSD), which are orders of magnitude better at random read operations and which is therefore able to bypass the disk bottleneck [MB10a, DSL10, LND12].

Another research direction is based on the observation that data in backup workloads is mostly redundant and often stored in a similar order as in the week before. The data stream has *locality* based on the ordering and has *similarity* because large data regions are near-duplicates of existing data. Some approaches [ZLP08, WJZF10, RCP08, LEB+09, BELL09] use the locality and/or similarity properties of deduplication backup streams to

predict future chunk requests so that the system is able to classify the chunk fingerprints without requesting the disk-based chunk index.

Some deduplication systems, e.g. "Sparse Indexing" [LEB+09] and "Extreme Binning" [BELL09], overcome the chunk lookup disk bottleneck using an approach that is not detecting and removing all redundant chunks. These systems are "approximate deduplication systems" because they trade a part of the potential deduplication savings for a higher and more predictable throughput.

Delta-based data deduplication share the chunking step, but it is not searching similar, but necessarily identical data blocks,. When delta-based data deduplication systems write a new chunk, they search similar chunks and then store only a differential encoding [ABF+02] between the new data and chunks that have been found to be similar [YCT+12].

An example for a delta-based data deduplication system is the work by Aronovich et al. describing the design of the IBM ProtecTier backup system [AAB+09]. With exception of a recent extension by Aronovich et al. [AAH+12], there has been no major research on delta-based data deduplication in recent years. However, related techniques, especially similarity detection, are used to support fingerprinting-based data deduplication, e.g., in [SHWH12, LEB+09, BELL09].

2.2 GENERAL RELATED WORK

This chapter highlights important contributions to the field of data deduplication, which are related to the overall context of this thesis. Related work that is only relevant to a part of this thesis will be described in the related part.

The first work searching duplicate and similar files in a large file system without using pair-wise comparisons was the "sif" tool by Manber [Man94]. He proposed to fingerprint all 50-byte substrings of a file and select a specific small sample of these as approximate fingerprints. Files that have approximate fingerprints in common are considered to be similar. While, the sif tool finds similar files, it is not removing duplicates or performing any other type of data compression. Broder later formalized a related sampling approach for near-duplicate detection [Bro97, Bro00].

Bentley and Mcilroy introduced an early data reduction approach, similar to data deduplication, in 1999 as a compression method [BM99]. The method uses static chunking and Rabin's fingerprinting method [Rab81] is used to fingerprint chunks. Since this fingerprinting method is not cryptographically secure, a byte-wise comparison of the new data and the already stored chunk is performed. The approach is for example used in the BigTable data store to remove coarse-grained redundancies within the data [CDG+08].

Bolosky et al. presented an extension to the NTFS file system, called "Single Instance Storage" (SIS) in 2000 [BCGD00]. It removes full file duplicates in a post-processing step.

Another post-processing deduplication system is "DDE", which was introduced in 2004 by Hong et al. [HL04]. It removes redundant data blocks using static chunking in a post-processing step.

The first file system using inline data deduplication was introduced by Muthitacharoen et al. in the "Low Bandwidth File System" (LBFS) [MCM01]. It was also the first proposal of using Rabin's fingerprints as the rolling hash function for content-defined chunking. In LBFS deduplication techniques are not used to avoid storing redundant data. Instead, the

focus is to avoid unnecessary data transfers over the network, similar to the later usage of data deduplication for WAN replication and WAN acceleration [SHWH12].

"Venti" by Quinlan et al. and "Pastiche" by Cox et al. (both 2002) have been the first storage systems using deduplication for data reduction of archival and backup data [QD02, CMN02]. Venti uses static chunking and cryptographic fingerprinting using a disk-based chunk index. Work related to Venti mention that it uses a Bloom filter [Blo70] to reduce the number of chunk index lookups [LH08, RCP08]. Pastice is a peer-2-peer (P2P) backup system using content-defined chunking and cryptographic fingerprinting to find redundancies.

Zhu et al.'s work was one of the first focusing on avoiding the chunk lookup disk bottleneck. They presented a combination of approaches that have been used in the EMC Data Domain File System (DDFS) [ZLP08]. These approaches allow reducing the number of chunk index lookups by up to 99%. First, a Bloom filter is used to avoid the majority of chunk index lookups if the chunk has actually not been stored before. Furthermore, a container per data stream is used to store the new chunk data in the order of arrival. This is capturing temporal locality, which is used by a container-based caching scheme. Therefore, the system was able to achieve a high throughput since it could avoid the chunk lookup disk bottleneck. This approaches will be picked up again in Part III of this thesis and used for comparision with new contributions.

The "Sparse Indexing" deduplication system uses a different approach to avoid the chunk lookup disk bottleneck [LEB+09]. Here, the chunks are sequentially grouped into segments. These segments are then used to search similar existing segments using a RAM-based index, which stores only a small fraction of the already stored chunks.

In contrast to other approaches, Sparse Indexing allows to store a chunk multiple times if the similarity based system is not able to detect the segments, which already have stored the chunk. Therefore, Sparse Indexing is a member of the class of approximate data deduplication systems.

The "Extreme Binning" approach is related to Sparse Indexing in its usage of similarity detection to overcome the disk bottleneck [BELL09]. It is a cluster deduplication system targeting low locality backup environments. This means that the backup workload consists of individual files, which arrive in random order. The deduplication detection of Extreme Binning calculates the full file fingerprint is calculated and chooses a single representative fingerprint of all chunks in the file. Then, the representative fingerprint is used to locate possible matches for the full file fingerprint and to locate a bin data structure that stores chunks from a similar data stream context. Sparse Indexing and Extreme Binning are also described in Part III because variants of them are used to evaluate the new approach presented there.

Guo and Efstathopoulos [GE11] and Xia et al. [XJFH11] proposed other approximate deduplication approaches.

Some approaches use faster hardware to overcome the disk bottleneck, especially SSD storage. The dedupv1 system is an inline deduplication system that places the performance critical chunk index structure on SSD storage [MB10a]. It will be presented in more detail in the next section as it provides the basis for some parts of this thesis.

Debnath et al. propose using a fully log-structured organization in their "ChunkStash" system [DSL10]. Lu et al. propose with "BloomStore", a flash-based Bloom filter driven key/value store for the usage in data deduplication systems [LND12].

While backup and archival storage are important use cases for data deduplication, it is not its only application domain. Clements et al. presented the "DeDe" approach for post-processing data deduplication in a SAN cluster file system for the storage of virtual machine images [CAVL09]. The inline deduplication systems "iDedup" by Srinivasan et al. [SBGV12] and "HANDS' by Wildani et al. [WMR12] focus on primary data storage. Both systems identify temporal and spatial locality of the working set. The locality enables different index designs that keep the chunk index in memory.

The "I/O Deduplication' approach by Koller et al. [KR10] uses observations from typical workloads of primary data storage to develop a deduplication approach for primary data. They use a "content-based cache", a similarity-based "dynamic replica retrieval approach", and "selective duplication". The usage of selective deduplication makes the approach an approximate deduplication approach.

The "DBLK" system uses a multi-level in-memory Bloom filter to implement an inline deduplication system for primary block storage [TW11].

Similarity detection or near-duplicate detection plays a critical role in so-called delta-based data deduplication as proposed by Douglis and Ivengar [DI03]. In these systems, similar chunks or files are searched and then only the encoded difference between a file and a similar file is stored.

Kulkarni et al. proposed a scheme called "REBL", which combines fingerprinting-based and delta-based data deduplication [KDLT04].

You et al. introduced the "Deep Store" system in 2005 [YPL05]. Deep Store combines different data deduplication techniques like full file fingerprinting and delta-based deduplication.

Aronovich et al. described a delta-based data deduplication system [AAB+09], which builds the foundation for the IBM ProtecTier system. With the exception of a recent extension by Aronovich et al. [AAH+12], there has been no recent research on delta-based data deduplication.

A concept related to data deduplication is known as "content-addressable storage" (CAS) [NKO+06, DGH+09, TKS+03]. As in data deduplication, unique data blocks are only stored once. The difference is the way the data is accessed and addressed. A deduplication system usually uses conventional block-based (e.g., iSCSI) or file-based (e.g., NFS) interfaces so that the deduplication is handled transparently to the user. In a CAS system, the fingerprint is passed to the client and the fingerprint is used to later address the stored data. CAS systems also have been used as building block in data deduplication file systems [UAA+10, TKS+03].

2.3 DEDUPV1

The dedupv1 is a backup-oriented, high-throughput, inline data deduplication system, which uses a SSD-based chunk index to overcome the chunk lookup disk bottleneck. The system is presented in this section in detail, as most of the presented research has been done in the context of the dedupv1 development. The research contributions on the design and architecture of data deduplication systems fit into the overall system architecture of dedupv1. Also, most contributions are evaluated using a prototype based on dedupv1.

The work on dedupv1 started in the Master's thesis of the author [Mei08] and its design has been published before [MB10a, MB10b]. The description in this section follows

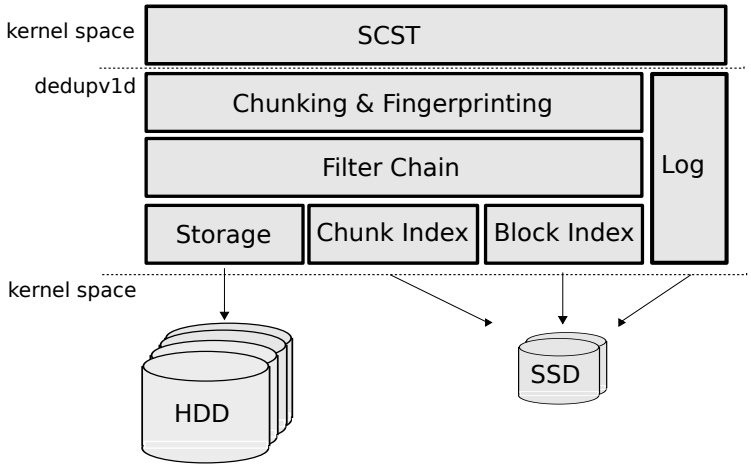


Figure 4: Architecture of the dedupv1 deduplication system (from [MB10b]).

these publications, but also updates them to reflect the most current development where necessary.

The key feature of the dedupv1 system is that it overcomes the chunk lookup disk bottleneck by storing the performance-critical chunk index on SSD storage. The architecture of dedupv1 is shown in Figure 4.

The dedupv1 system uses a SCSI target subsystem for Linux, called SCST [Bol12]. The dedupv1 user-space daemon, dedupv1d, communicates with the SCST kernel subsystem via ioctl calls. The SCST SCSI target implementation allows dedupv1 to export deduplicated volumes via iSCSI [SMS+04] or Fibre Channel (FC). The data deduplication is therefore transparent to the SCSI initiator.

After a command handling thread of dedupv1 has accepted a SCSI WRITE command, the command is transferred to the dedupv1 core engine. There, the chunking component splits the request data into chunks. The chunking is configurable per volume. Content-defined chunking [PP04] using Rabin's fingerprinting method [Rab81] is used by default. Afterwards, each chunk is fingerprinted using a cryptographic fingerprinting method. As the chunking, the fingerprinting is also configurable. A usual configuration uses the SHA-1 fingerprinting method.

The next step is the so-called filter chain. The responsibility of the filter chain is to decide if a chunk is a duplicate or not. At this time, all chunks are processed independently. In the current version of dedupv1, the filter chain is processed using a thread pool.

A filter chain consists of a series of filters. Each filter processes a chunk and tries to classify if the data of the chunk has already been stored in the system. Each filter can have one of the following results that indicates the classification and the "confidence" of the filter in the classification decision:

EXISTING: The filter has validated that a chunk with exactly identical data as the current chunk has already been stored. The only filter that is able to return this result has performed a byte-wise comparison of the chunk data with an already existing chunk. The filter chain execution is stopped after a filter returns this result.

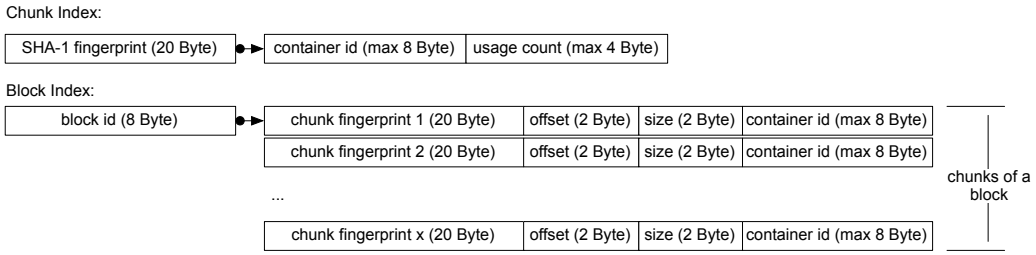


Figure 5: Data structures of the dedupv1 system. The chunk mapping is stored in the chunk index (top). The block recipe is stored in the block index (bottom) (from [MB10b]).

STRONG-MAYBY: A filter classifies a chunk with this result if the chunk has already been stored with high probability. Usually this involves comparing the chunk fingerprint with the chunk fingerprint of a chunk known to be stored already. The classical chunk index lookup is the major example for this type of filter. After a filter returned this result, only filters that are able to return an EXISTING result are executed.

WEAK-MAYBE: The filter is not able to make any classification, but it also cannot rule out the possibility that the chunk has already been stored.

NON-EXISTING: The filter classified that the chunk is not known to the system and that therefore the chunk data needs to be stored. This result is for example returned after a negative chunk index lookup. No other filter are executed after this result.

If the final result of the filter chain is NON-EXISTING, the chunk data is stored. The filter chain is executed a second time so that all filters can update their internal state after the data has been stored.

The filter chain abstraction proved to be an easy and powerful way to implement new research ideas including ways to do the chunk duplicate classification. The Master's thesis of the author included a demonstration that a Bloom filter and the container-caching based approach of Zhu et al. [ZLP08] can be integrated into dedupv1 as filters [Mei08]. The new Block Locality Cache has also been integrated into the dedupv1 system as a filter.

The chunk index is a major component of the system, which stores information about all chunks in the system. The chunk index is used by a filter to check if a chunk is already stored if no other filter was able to provide a sufficient classification.

For each chunk stored in the system, the chunk index contains a "chunk mapping". The chunk mapping stores the fingerprint, the container in which the chunk data is stored, and a usage counter (see Figure 5).

A paged disk-based hash table is used for the chunk index. The chunk index is stored on one or multiple Solid State Disks (SSD) to provide a high system throughput. One difference to existing public descriptions of dedupv1 is that the "auxiliary chunk index" has been replaced by an integrated write-back caching scheme that allows a better control of the IO traffic during write-back.

The dedupv1 system uses reference-counting-based garbage collection [Col60] to remove chunk index entires and chunk data that are no longer used. The number of references to a chunk is maintained in the chunk mapping of the chunk index. The reference

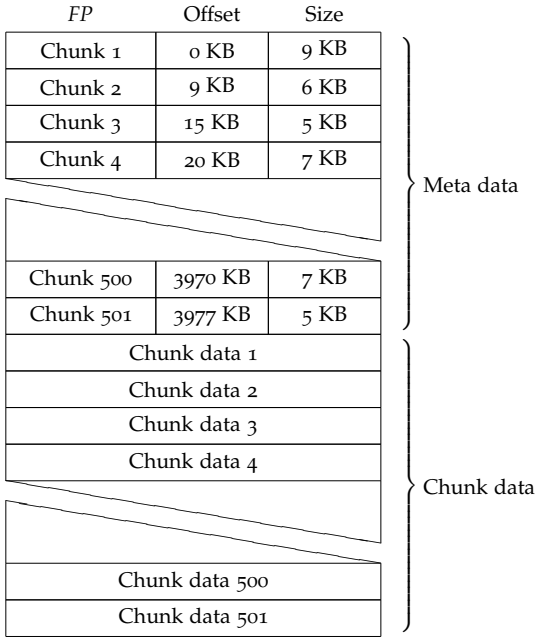


Figure 6: Structure of a (simplified) container (from [Mei08]).

count is updated in the background based on the direct log replay of the operations log, which will be explained later. However, the directly updated reference counts are not immediately persisted to disk, but maintained in-memory in the chunk index write-back cache. Eventually, the write back cache is persisted to disk in efficient sequential write-back operations.

Chunks without any references are marked as "garbage collection candidates". The chunks are not removed directly to avoid race conditions with new data that is written in parallel. These races are avoided by a Bloom filter, which maintains which chunks might have an out-dated reference count. These chunks then are called "in-combat" and they are skipped over by the background garbage collection process. Chunks without active references are marked as deleted in the chunk index and the chunk data is eventually removed.

Another index in the dedupv1 system is the block index. The block index stores block recipes, which maps from a 256 KB-sized block of the virtual device to the chunks that build the block contents. A block recipe is the equivalent to a file recipe in file-based data deduplication systems. The block index is shown in Figure 5.

Similar to other deduplication systems [ZLP08, QD02, WJZF10], dedupv1 groups multiple chunks together in a "container". A container has usually a size of 4 MB. Often, the chunk data stored in a container is compressed using standard data compression approaches, e.g., based on LZ77 compression [ZL77].

In the header of the container, all chunk fingerprints are stored together with a link to the offset of the chunk data in the body of the container and additional metadata as size of the chunk data and the compression type. The chunk data is stored in the body of the container. Figure 6 shows the structure of a container.

The containers are garbage collected. When a container item, which stores the data of a single chunk, is deleted from a container, the data is not removed immediately, but marked as deleted. Two containers can be merged into a new container when the non-deleted data of both fits into a new container. The containers are accessible using a container id. A chunk that is stored in a container will be accessible using the container id as long as the chunk is stored in the system.

To enable this invariant, the container ids are managed using a union-find data structure. A container has multiple container ids where one container id is the primary container id. If two containers with ids a and b are merged, one of the primary container ids is selected as new primary and the other ids are redirected to it.

The last component of dedupv1 is the operations log. All data modifications that are necessary for data consistency and are not immediately persisted to disk use the operations log to be able to recover a consistent state after a crash.

The log is not only used for crash recovery, but also for delaying and aggregating cleanup and IO operations even if the system is performing as expected. This allows for example to aggregate multiple changes to usage count data of the chunks into a single IO.

All events stored in the operations log are replayed multiple times during the life-time of a log event:

DIRECT LOG REPLAY: After an event is committed to the log, it is replayed directly afterwards in a background thread. It is for example used to update the usage count value of a chunk in the write-back cache.

BACKGROUND LOG REPLAY: When the system is idle or when the log is full, a background replay of the log entries is done. All data that relies on a log entry to be recoverable is persisted during the background log replay. After an event has been replayed in the background, it is removed from the log.

DIRTY START LOG REPLAY: A new log replay mode that was introduced after the previous publications is the dirty start log replay. It is performed after a crash instead of a full background log replay. It essentially recovers the in-memory state of the system directly before the crash. However, only essential information is persisted to allow a fast recovery of the system.

The operations log is realized as an on-disk ring buffer on SSD storage. All events are replayed in the different phases in a fixed, serialized order.

The dedupv1 architecture allows an exact inline data deduplication for backup workloads. It has been under active development since 2008 and has been released under an Open Source license. The current version can be found at `https://github.com/dedupv1/dedupv1`.

The single-node dedupv1 architecture is extended to form a data deduplication cluster, called di-dedupv1 in Part II. In Part III, compression schemes for the block index are presented. Furthermore, a novel approach, called Block Locality Cache (BLC), to overcome the chunk lookup disk bottleneck based on locality properties of the block index is proposed in that section. The BLC removes the need to use SSD storage for the index structures.

2.4 FS-C: DEDUPLICATION TRACING AND ANALYSIS

The FS-C is a tool suite to analyze the redundancy potential and the pattern of fingerprinting-based data deduplication. Like the dedupv1 system, the tool suite has its origins in the Master's thesis of the author [Mei08]. The tool suite has been published as open source and is available at `https://code.google.com/p/fs-c/`.

The usage of FS-C is a two-step process: a file system walk and an analysis step. During the file system walk, all files are read, chunked, and fingerprinted. Information about the files and the chunk fingerprints are stored in a trace file. Multiple chunking approaches can be used within a single file system walk.

By default, FS-C uses content-defined chunking with an expected chunk size of 8 KB. This is also the default chunk size in some backup data deduplication systems. In addition, FS-C supports static chunking, which splits the files into a series of equally sized chunks and it can be extended by other chunking approaches. All chunks are fingerprinted using SHA-1.

The output of the FS-C file system walk is a trace file containing all data that is necessary for later analysis:

- The file name, the file length, the file type, and a series of chunks for each file.
- The fingerprint and the chunk length of each chunk.

Often, a hash of the file name replaces the file name to ensure privacy. The file type is determined by the file name extension by searching for a dot character within the last five characters of the file name. If no dot can be found, the file type is stored as "Not available".

By default, the full fingerprint (20 bytes with SHA-1) is stored per chunk. Alternatively, only a prefix of the full fingerprint is stored to reduce the size of the trace file.

The trace files can then be analyzed for their data deduplication potential as it has been done in [MB09] and in Part IV of this thesis.

The overall deduplication ratio ds is defined as $1 - \left(\frac{\text{stored capacity}}{\text{total capacity}}\right)$, which is equal to $\left(\frac{\text{redundant capacity}}{\text{total capacity}}\right)$. A deduplication ratio of 25% means that 25% of the data are removed by data deduplication and 75% of the original data is actually stored. The deduplication factor is an alternative way to describe the data savings. It is defines as a factor of 1:x where x is equal to $1/ds$ [ZLP08, WDQ+12].

The analysis can be performed in three different ways:

IN-MEMORY: Based on the trace file, an in-memory chunk index is built to detect duplicate fingerprints. This approach has scalability limitations as the complete chunk index has to fit into main memory.

HADOOP: The MapReduce paradigm for distributed data is used to analyze the fingerprints [DG04, DG08]. The analysis can be modeled as series of MapReduce jobs in a natural and efficient way. The data analysis script is written in the data flow language Pig Latin [ORS+08], which is executed on a Hadoop cluster [SKRC10, Apa].

Listing 1 shows a slightly simplified version of the script that calculates the data deduplication ratio per file type.

Listing 1: Simplified Pig Latin script for calculating the deduplication ratio per file type.

```
CHUNKS = LOAD '$RUN/chunks*' AS (filename, fp, chunksize);
FILES = LOAD '$RUN/files*' AS (filename, filelength, fileext);

FILES_A = FOREACH FILES GENERATE filename, FileSizeCategory(filelength), fileext;
A = JOIN FILES_A by filename, CHUNKS by filename;
B = GROUP A BY (fp, filelength, fileext);
CL = FOREACH B GENERATE fp, fileextn, COUNT($1) as usage_count, MAX($1.chunksize);
S1 = GROUP CL BY fp;
S2 = FOREACH S1 GENERATE group, SUM(usage_count), MAX(chunksize);
S3 = FOREACH S2 GENERATE group as fp, chunksize / usage_count as storedsize;

T1 = JOIN S3 BY fp, CL BY fp;
T2 = FOREACH T1 GENERATE fileext, usage_count, chunksize as chunksize, storedsize;
T3 = FOREACH T2 GENERATE fileext, usage_count * chunksize as totalsize,
       usage_count * storedsize as storedsize;
U1 = GROUP T3 BY fileext;
U2 = FOREACH U1 GENERATE group as fileextn, SUM(totalsize), SUM(storedsize);
U3 = FOREACH U2 GENERATE fileext, totalsize, storedsize,
        totalsize - storedsize as redundancy;
U4 = ORDER U3 BY totalsize DESC;
DUMP U4;
```

In the first two lines, data is loaded from files containing chunking information and additional metadata about the fingerprinted files. The chunk data is joined with the metadata to add file type information to all entries (Alias B). Afterwards, the script calculates the usage count for each fingerprint (Alias S2) and the number of persistent bytes each chunk reference is responsible for, which is the fraction of the chunk size and the number of references to that chunk (Alias S3). This way of calculating the contribution of each reference is also used by Harnik et al. [HMN+12].

The second half of the script iterates a second time over all chunks and calculates the total size and the stored size grouped by the chunk fingerprint and the file extension. The total size and the stored size are accumulated per file extension. The difference between the total capacity and the stored capacity is the redundancy within a data set that could be avoided to store with a deduplication.

A similar script is used to calculate the deduplication ratio grouped by file sizes and the overall deduplication ratio. Hereby, the calculation results up to step CL are stored and reused to avoid expensive, unnecessary recalculations.

The advantage of the Hadoop method is that it allows analyzing nearly arbitrary sized file systems. On the other hand, a Hadoop cluster of the necessary size may not be always available.

HARNIK'S METHOD: An alternative to the first two approaches is Harnik et al.'s method to estimate the deduplication ratio [HMN+12]. This method allows an estimation within a bounded error. It works in two phases: sampling phase and scanning phase.

In the sampling phase, a pre-determined number of chunks is chosen randomly (uniformly distributed and independent). The size of the sample is denoted as m. The fingerprints of the sampled chunks are calculated and stored.

Harnik et al. propose to determine which files and which offsets within a file should be sampled during the file system directory walk. In the FS-C implementation, the already existing trace file is used. The sampled chunks are chosen using Reservoir Sampling [Vit85].

In the scanning phase, the fingerprints of all chunks are computed. For all chunks that have been chosen during the sample phase, the number of references is counted. Finally, the deduplication ratio is estimated as $\frac{1}{m}\sum_{i\in S}(\text{base}_i\,\text{count}_i)$ where S denotes the sample, base_i denotes the number of sampled occurrences of the fingerprint in the sample, and count_i is the number of occurrences of the fingerprint found during the full scan in the second phase.

If FS-C uses Harnik's method, it configures the sampling according to the recipes given by Harnik et al. [HMN+12] so that the estimation error is less than 1% with a probability of 99.9%.

The disadvantage of Harnik's method in the FS-C implementation is that it only allows estimating the overall deduplication ratio. It is (currently) not possible to calculate the deduplication ratios grouped by file sizes and file type as the Hadoop-based method allows it.

The Hadoop and Harnik's method have been used in Part IV of this thesis to analyze more than 3 PB of file system data.

Other important usages of FS-C are trace-based simulations. It is easy to develop trace-based simulations using existing or new FS-C trace files. In Part III of this thesis, simulations based on the FS-C tool suite have been used to evaluate the file recipe compression as well as the Block Locality Cache and its alternative approaches. The trace-based simulation has also been used in Part II as part of the approach to generate backup data streams to evaluate data deduplication systems.

Three different backup data sets are used throughout the thesis. All traces use content-defined chunking with an expected chunk size of 8 KB.

UPB: The first data set is based on the home directory file server of the University of Paderborn, which has been collected in 2008 [MB09]. It is a time series containing 15 weekly full backup traces of the same file system of around 440 GB (6.6 TB total). With data deduplication, the data set size can be reduced by a factor of 1:20 to 369 GB.

The trace has been collected with a single crawl thread so that the order of the files in the trace file is the same as they are returned from directory listing function. This order is stable over the weeks so that the files are in the same order very week. This order preservation is an important property for backup deduplication systems that use the order of the chunks beyond file boundaries.

JGU: The second data set contains trace data from a home directory file server at the Johannes Gutenberg University Mainz. This data set, which has been collected in 2012, consists of 13 weekly backups forming a total data set of 16.1 TB. Data deduplication compresses the backup data set to 1.4 TB (factor 1:11.5).

Table 1: Statistics about the backup data sets.

	UPB	JGU	ENG
Total File Count	53,605,162	30,744,985	256,207
Total Capacity	6.6 TB	16.1 TB	318.7 GB
Capacity After Deduplication	369.0 GB	1.4 TB	8.0 GB
Mean File Size	132.4 KB	700.3 KB	1.3 MB
Median File Size	1.0 KB	13.50 KB	31.6 KB
Cumulated Median File Size	7.7 GB	42.9 GB	1.5 GB

The data set has been collected using one directory crawl thread and one data crawl thread. Therefore, the order of the directories is preserved but the order of the files within a directory may be mangled. In a post-processing step, the file have been reordered so that the file order is preserved over the backup runs. If new files are added in a backup run, they are placed in the reordered trace behind the same files as in the original trace.

ENG: The third data set contains fingerprints of files from a file server of an engineering department of a private company backed up over 21 weeks in 2011/2012 (319 GB total). The file server seems to be barely used and data deduplication is able to shrink the physical size of this data set by a factor of 1:40 to 8.0 GB.

This data set has been collected using multiple directory crawl and data crawl threads. Therefore, the order of the files in the trace is not deterministic between the weeks. However, the order of the chunks within the file are preserved, so that this data set can be used in settings where the chunk order is not important at all or if only the order within files is important.

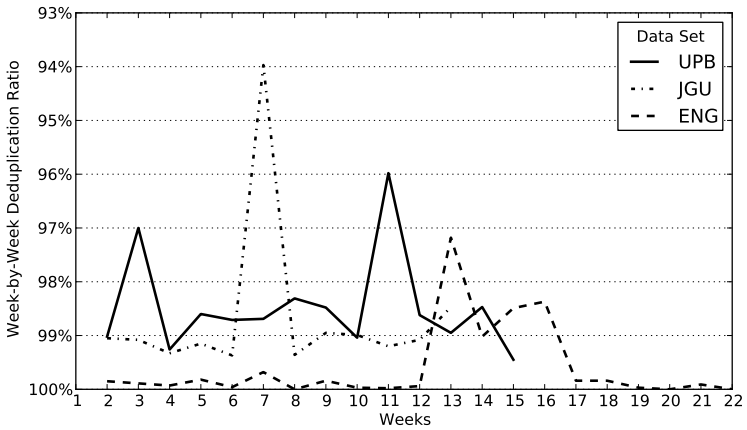


Figure 7: Week-by-week inter-backup deduplication ratio using content-defined chunking with an expected chunk size of 8 KB of different backup data sets.

Table 1 summarizes some elementary statistics about the data sets. All values are the total over all backup runs. The capacity after deduplication denotes the data capacity of the complete backup data set after is has been reduced using data deduplication with content-defined chunking and an expected chunk size of 8 KB. The cumulated median file size denotes the, the file size for which 50% of the capacity is occupied by smaller files and 50% by larger files.

Figure 7 shows the week-by-week inter-backup data deduplication ratio on the different data sets. The deduplication ratio here denotes the percentage of the capacity of a backup run that is stored after data deduplication using the inter-backup redundancy of the previous backup run.

Part II

DI-DEDUPV1 DATA DEDUPLICATION CLUSTER

3

INTRODUCTION

In this part, the focus is on a novel architecture to scale data deduplication by using multiple cooperating nodes. In addition, a model that predicts the communication behavior is developed and used to explain the limitations of the architecture.

With the real deployment of deduplication systems, it became clear that a single-node deduplication system is not able to fulfill all requirements with respect to throughput and fault tolerance.

This can partly be bypassed by using multiple independent deduplication systems. However, such setups are difficult to manage. Furthermore, the efficiency of the redundancy removal decreases, as identical data is stored on multiple of these deduplication instances. While Douglis et al. proposed a scheme to assign data to multiple independent deduplication systems so that the loss in deduplication is reduced [DBQS11], other approaches are worth investigating. Hence, one of the goals of this thesis is to develop and evaluate an exact deduplication system architecture that can be scaled to a cluster of 16-32 nodes.

The definition of exact data deduplication does not state how chunks are built and which size they have. This is another property affecting the trade-off between deduplication and throughput. Larger chunks usually lead to a higher throughput, but deduplication potential is lost [WDQ+12, MB09, MB12]. The chunk sizes of deployed deduplication storage systems are usually between 4 KB and 16 KB [QD02, ZLP08, LEB+09, BELL09]. Some systems,e.g. the HYDRAstor/HydraFS combination [UAA+10, DGH+09], use large chunks (64 KB and larger) and achieve a high throughput needing only an eighth eighth part of the chunk lookups than a system with 8 KB chunks. Therefore, another requirement is added is that the expected chunk size should be between 4 KB and 16 KB. A chunk size of 8 KB enables a comparison with the standard configuration of dedupv1 and other systems.

The challenge of designing an exact deduplication cluster with small chunk sizes and the reason why it has been asserted (compare e.g. [BELL09]) that such a design is not feasible, as nearly all chunk index lookups require a network lookup. The contribution of the architecture, called "di-dedupv1" (for distributed dedupv1), is that it is shows the possibility to combine exact deduplication with small chunk sizes and scalability in one environment.

In di-dedupv1, all major data structures are spread across all nodes. The design separates the node on which the chunk fingerprint can be found in the chunk index from the node on which the chunk data is stored in a container. However, the protocol design avoids sending the actual deduplicated data over the network in the write-path. Usually, only a small message containing the fingerprint of the data is exchanged between nodes.

The architecture is implemented in a prototype based on the dedupv1 system. The evaluation of the prototype shows that the architecture scales up to at least 24 cluster nodes even with a commodity GBit Ethernet interconnect. In addition, a communication model is developed and verified against the prototype. The model can be used to predict the performance of other configurations and also shows the limitations of the architecture.

This research shows that it is possible and feasible to combine exact deduplication with small chunk sizes and scalability within one environment. There are different ways to increase the throughput of a data deduplication system by avoiding a chunk index lookup for the majority of found chunks [ZLP08, WJZF10]. One novel approach to use and "capture" locality will be presented in another part of this thesis. The design presented here is not assuming locality. It can therefore be used in situations where it is not be possible to rely on locality.

The research presented in this part has been published before in "Design of an Exact Data Deduplication Cluster" [KMBE12]. The di-dedupv1 prototype, which extends the dedupv1 single-node system, has mainly been implemented by Jürgen Kaiser. An early version of the design has been presented in Jürgen Kaiser's Master's thesis [Kai11].

This part starts by a look on the related work specific to the scaling of deduplication systems in Chapter 4. Afterwards, Chapter 5 describes the system design by providing an overview of the system design in Section 5.1. This is followed by a detailed look on the storage organization, the communication protocols, and the fault tolerance and load balancing properties. Chapter 6 describes the methodology of the evaluation and the evaluation results. Finally, Chapter 7 summarizes the contribution presented in this part.

4

RELATED WORK

While general related work has been presented in Chapter 2, this section adds related work focusing on distributed and clustered data deduplication.

"Extreme Binning" by Bhagwat et al. is an approximate deduplication cluster for file-based backup workloads. The approach is optimized for situations where the locality properties used e.g. by Zhu et al. are not available [ZLP08]. In that approach, a representative chunk is chosen per file [BELL09]. The representative chunks build a sparse index, which is striped over all nodes of the system. The distributed sparse index is then used to find a "bin", whose content is likely to be similar to the file content. Due to the sparse index with a single representative fingerprint per file, the approach has a low communication overhead. The authors do not analyze the throughput of their system.

Dong et al.'s extension of Zhu et al.'s work on the EMC Data Domain file system is based on similar ideas, but uses so-called "super chunks" for the data routing instead using a single chunk fingerprint per file [DDL+11, ZLP08].

The "SiLo" system by Xia et al. is another proposed design for an approximate deduplication cluster [XJFH11]. It uses locality and similarity to overcome the disk bottleneck. The throughput is only estimated based on the processing rate of chunk fingerprints. Also, the evaluation is only based on a single node.

Dubnicki et al. presented "HYDRAstor", a content-addressable storage cluster that uses large chunks of 64 KB [DGH+09]. HYDRAstor is used as a building block for a distributed file system called "HydraFS" [UAA+10]. The chunk fingerprint data is distributed using a modified variant of the "Fixed Prefix Network" (FPN) [DUK04], a kind of "distributed hash table" (DHT). The data is distributed to multiple nodes using "erasure coding" for resiliency. Using 64 KB chunks and holding the chunk index in memory bypasses the chunk lookup disk bottleneck. While the design is that of an exact deduplication system, the choice of the chunk size favors throughput over deduplication ratio [DGH+09, UAA+10]. They report scalability results up to 12 nodes with a system throughput of 600 MB/s for non-deduplicating data with 0% redundant chunks and 1200 MB/s with 100% duplicate chunks [DGH+09].

The "MAD2" system by Wei et al. is another approach for a distributed deduplication system applying exact deduplication [WJZF10]. The chunk data is distributed using a DHT using a Bloom filter scheme and a locality preserving caching scheme similar to Zhu et al.'s [ZLP08]. The system uses local storage, but does not replicate chunk fingerprints to multiple nodes so that the system is unable to tolerate node crashes. Additionally, the prototype has been demonstrated only using two nodes. The authors neither try to evaluate further scalability nor do they consider the impact of the underlying network.

"DEBAR" is a design for a post-processing deduplication system cluster [YJF+10]. Similarly, "DeDe" and "DDE" are post-processing deduplication clusters using a storage area network (SAN) [CAVL09, HL04].

Some commercial clustered storage systems with deduplication or similar techniques are available. Due to the proprietary nature of them often only little is known about the clustering and data deduplication approaches. Septon provides a clustered deduplication backup system using a content metadata aware deduplication scheme with a byte-wise comparison of the data. Based on the information available it is hard to classify the Septon system, but it is most-likely not fingerprinting-based deduplication [Sep10].

The commercial Permabit system is an inline, fingerprint-based deduplication system using a DHT-based approach using local storage at the nodes [Per12]. Permabit may use a scheme storing metadata about the fingerprints in RAM using a scheme that allows membership-testing with a certain false-positive rate [MOS+05]. The data structure uses the property that the keys are cryptographic fingerprints for a scheme that they are claiming is more efficient than Bloom filter [Blo70]. Little is known about its approaches for data distribution, load balancing, and fault tolerance.

An alternative to distributed deduplication systems is the usage of multiple separate deduplication appliances that are not cooperating. However, this reduces the possible savings since overlapping backup data, which is assigned to different machines, is stored multiple times. Additionally, moving backups between machines, e.g., after adding new appliances, requires to store already deduplicated data redundantly. Thus, the overall deduplication setup has only approximate data deduplication capabilities if the deduplication systems themselves perform exact data deduplication.

Douglis et al. present a load balancing/backup assignment scheme so that backups are assigned to deduplication instances so that the overlap and affinity between backups runs is increased [DBQS11]. However, a clustered deduplication system where the data is deduplicated between all deduplication nodes makes the management easier, enables an easier load balancing, and increases the storage savings of data deduplication.

The discussion between "shared disk" and "stored nothing" storage systems is not new. There are several examples for both concepts. IBM's GPFS, IBM Storage Tank, RedHat's GFS, and VMFS are shared disk storage systems [SH02, MPR+03, PBB+99, VMw09]. On the other hand, Ceph, Lustre, and PanFS use local disks to which other nodes do not have block-level access [WBM+06, Bra02, NSM04].

A similar approach to crash recovery, although not in the context of distributed data deduplication systems, has been proposed by Devarakonda et al. in the "Calypso file system" [DKM96]. They also use shared storage, called there multi-ported disks, to recover the storage system state after crashes while only a single server accesses a disk at any time. They also relied on RAID [PGK88] or mirroring approaches to provide disk fault tolerance. In addition, they used client state to recover from node failures. A contrary position is taken by Welch et al., who propose using replicate state in main memory for recovery [WBD+89].

The property that it is sufficient if nodes can only access a subset of the available storage is related to the storage pool concept of V:Drive [BHM+04] or the view concept in the consistent hashing scheme [KLL+97].

The communication and the network interconnect between storage peers have been analyzed before by Brinkmann and Effert [BE07]. They conclude that the interconnect can become the bottleneck, but the authors concentrate on bandwidth-limited traffic. This trap

is bypassed in this work by avoiding that the actual data is sent over the internal communication network. Similar observations have been presented by Nagle et al. [NSM04].

5

SYSTEM DESIGN

This chapter describes the system design of di-dedupv1. The first section provides an overview of the di-dedupv1 system design, describing the overall components of the system, the interactions, and the major design considerations. Later sections cover the most important components in more detail. The organization of the storage is discussed in Section 5.2. The communication protocol is covered in Section 5.3. The crosscutting properties of fault handling and the load balancing are explained in Section 5.4.

5.1 SYSTEM DESIGN OVERVIEW

As discussed earlier, one of the explicit targets of this work has been to explore exact-deduplication clusters. This contrasts to most clustered deduplication systems, which are approximate deduplication systems.

Another design requirement is that the deduplication system should offer the standard interfaces for block based storage as iSCSI and Fibre Channel, so that clients do not have to install new drivers. This is different from other solutions as, e.g., presented by Dong et al. for a single-node deduplication system [DDL+11].

The di-dedupv1 deduplication cluster is based on dedupv1 and inherits most of its properties. However, all storage components are split and distributed amoung all nodes. The design consists of the following components:

DEDUPLICATION NODES: Deduplication nodes are extensions of the single-node variant. They export volumes over iSCSI and FC, accept incoming requests, perform chunking and fingerprinting, reply to chunk mapping requests of other nodes, and write new chunks to the container storage. Figure 8 shows the internal components of a deduplication node. All deduplication nodes are responsible for a part of the chunk index, the block index, and the container storage.

The node that is responsible for the chunk index entry of a chunk is separated from the node that is responsible for the container that stored the data of a chunk. This separation allows designing a communication protocol that avoids exchanging the chunk data in the performance-critical write-path.

In the single-node dedupv1 system, the operations log is shared between all system components. This changes in di-dedupv1 where the log is split up for the different partitions (which will be introduced later) as a partition may be assigned to different nodes in cases of failure recovery and load balancing.

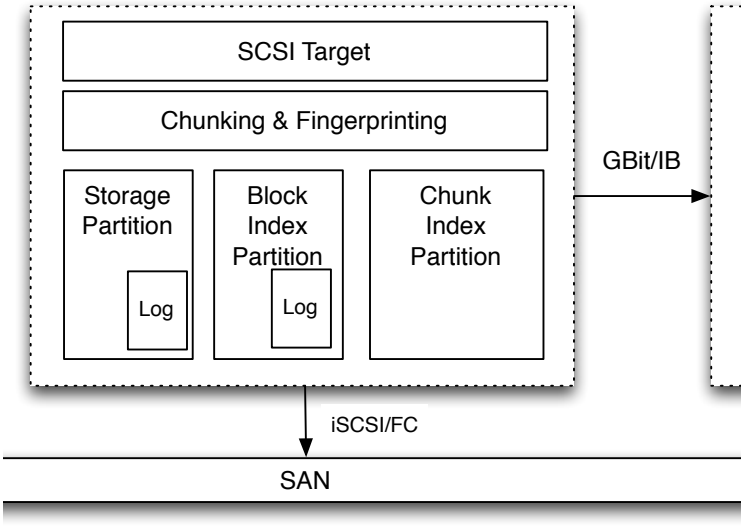


Figure 8: Single deduplication node. Incoming SCSI requests are chunked and fingerprinted, and then chunk lookup requests are sent to the responsible nodes of the cluster.

At all times, one of the deduplication nodes is elected as distinguished "master node". The master node is responsible to coordinate the load balancing and a small set of other decisions necessary for cluster-wide coordination.

SHARED STORAGE: Deduplication nodes have access to hard disks and SSD storage via one or more storage area networks. The available storage is split into many partitions. Each partition contains its own local file system and is only accessed by a single deduplication node at a time. If a node crashes, a surviving node takes over the responsibility for a partition and can restart operating with that partition. A similar approach is used for load balancing, where the system moves the responsibility of a partition to a different node. In both cases, there is no need to move any data. Instead of moving the data, the responsibility for the data is moved.

One assumption is that the shared storage back-end provides a reliable access to the stored data, e.g., by using RAID-6 on hard disks as well as SSD storage. This frees the system from explicitly managing the reliability of the already stored data.

Another assumption is that it is possible to scale the back-end storage with the deduplication cluster. This is achieved by being able to use multiple storage systems as the back-end. The data distribution and communication policies allow configurations where a deduplication node can only access a subset of the storage back-end partitions. The only requirement is that all storage partitions are accessible by sufficent deduplication nodes to tolerate node failures and to enable load balancing.

The system distinguishes between different types of partitions, some using HDDs, some SSD-based storage. The types of partitions and the role in the system are described later.

INTERCONNECT BETWEEN DEDUPLICATION NODES: As the chunk index is distributed over all nodes, it is necessary in an exact deduplication cluster to communicate with other nodes when checking if a chunk is already known or not. However, around 120,000 chunk index lookups per second are necessary for each GB/s throughput, assuming a chunk size of 8 KB.

The design of the communication pattern always tries to avoid sending chunk data over the internal network in the write-path. This is a key feature that enables a high throughput. The network might be bandwidth-limited if chunk data would be transferred.

CLIENTS: The clients of the system communicate with the cluster using standard block level protocols as iSCSI or FC. The approach therefore does not require any modifications on the clients.

DISTRIBUTED COORDINATION: The system requires that certain critical information are held consistent between all nodes. This includes the system configuration and the currently elected master node.

The distributed coordination service also stores the assignment of storage partitions to deduplication nodes. This information is also cached locally so that the information might be outdated on the nodes. Such a situation is detected and the local state is corrected by loading the data from the distribution coordination service.

The "ZooKeeper" system is used for the coordination service [HKJR10]. A ZooKeeper daemon is running on all or a subset of the deduplication nodes. All deduplication nodes connect to the ZooKeeper service for the distributed coordination.

5.2 STORAGE ORGANIZATION

An important element of the system design is the storage organization, which describes the hardware and software architecture of the storage system and how persistent information is stored.

The di-dedupv1 storage architecture using shared storage is illustrated in Figure 9.

Central to the overall design is the usage of shared disk storage for all persistent data. In a shared disk storage architecture the storage is not located at the nodes directly, but connected using a separate storage network. In princple, the storage can be accessed by multiple or even all nodes over the network. It is the responsibility of the nodes to ensure that data integrity constraints are met, e.g. by using external synchronization schemes as e.g. GPFS and other storage systems use [SH02, TML97, LT96, PBB+99, VMw09].

The current prototype assumes block-based storage as back-end storage system. The block-based storage for example could be accessed over iSCSI or FC. However, the general approach could as well use a distributed file system or other ways to share storage.

The deduplicated data is organized in containers. This data is stored on HDD-based storage back-ends. The container storage is not using a central organization. All nodes manage the container storage independently for all other nodes. Container data garbage collection and possible de-fragmentation is performed locally and in the background.

The performance critical index data, especially the chunk index, is stored on SSD storage. This is similar to the design of the single-node dedupv1 system. However, here the SSD storage is accessed over a storage area network.

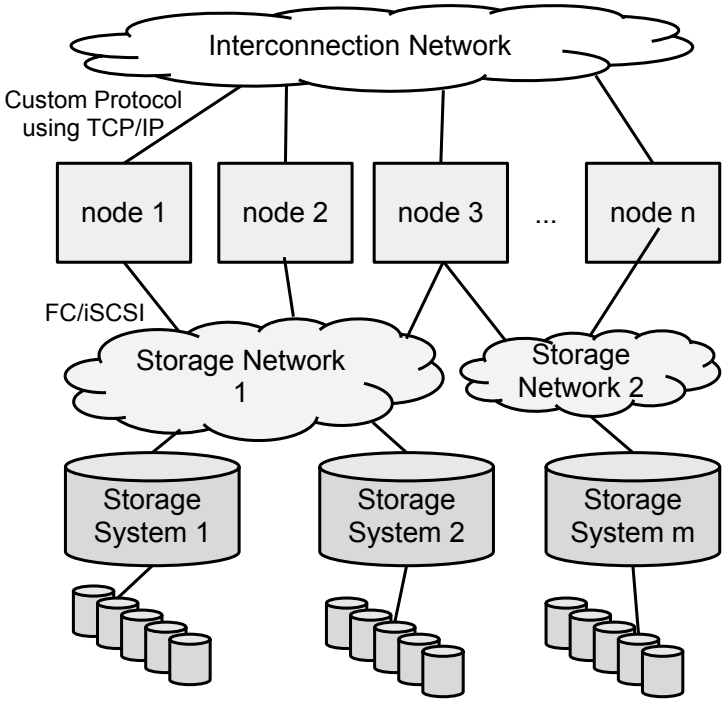


Figure 9: di-dedupv1's storage architecture.

5.2.1 *Comparison with Shared Nothing Approaches*

An alternative to the shared storage model is a "shared nothing" approach [Sto86]. In the shared nothing approach, the storage is directly attached to the cluster nodes and the only way that nodes can exchange data is over the internal network. The chunk index might also be distributed over all nodes, e.g., by hashing the chunk fingerprint to a node similar to a "distributed hash table". This architectural model for clustered data duplication is, e.g., used by Dubnicki et al. and Wei et al. [WJZF10, DGH+09].

The shared storage is the foundation for fault tolerance of the architecture. A different node can work on the same data immediately after a node has crashed without any data copy operations as the other node can see the same data over the storage network. The data will be stored redundantly on the storage system itself, e.g. using a RAID scheme. As reliability is handled locally within the back-end storage, there is no network traffic to replicate data.

If data is only stored locally on direct-attached storage (DAS) on a node and this node goes down, the data is unavailable until the node gets online again (e.g., after a hardware replacement) or after disks are physically moved to a different node. However, replication of written data also means that the chunk data has to be copied over the internal communication network. Furthermore, besides the chunk data also the index data structures would be stored on multiple nodes. This would further increase the amount of messages sent to keep the replications in sync.

Another difference between the chosen storage model and a distributed hash table model using shared nothing storage is handling of cluster expansions by adding new cluster nodes. If a new node joins the cluster, the load balancing process will immediately assign partitions to the node. Using this storage model, this can be done nearly instantaneous without copying data between nodes. For example, the consistent hashing scheme [KLL+97] would require to copy $O(1/d)$ of the data when a new node joins the network with d nodes.

Most new scalable storage systems use the "shared nothing" approach as traditionally shared disk systems need complex communication schemes so that multiple nodes can access the same storage concurrently. The distributed dedupv1 architecture avoids this and allows only a single node to access a partition. The shared storage systems are used as they were local in the normal processing. The property that the storage is shared between nodes is only used for crash recovery and load balancing. This avoids the complexity and the possible inefficiencies of typical shared storage approaches.

5.2.2 *Storage Partitioning*

Each storage partition is realized using separate SCSI logical units and/or using block device partitions. Each is formatted with a local file system as ext4 [CBT07]. Each partition is assigned to one and only one node. Only this node is allowed to mount the file system and to access the data. In certain situations, e.g., when a node crash is detected, the responsibility for a partition is moved from one node to another. The current partition assignment mapping is maintained using the distributed coordination service.

All persistent data structures of dedupv1, especially the chunk index, block index, and the container storage are split into several parts. Each part of these structures is stored on a different partition.

The partitions are differentiated between types of partitions. Each type stores different information and each type may be handled differently for crash recovery and load balancing. Also, the type of a partition determines if it is stored on HDD or SSD storage.

CHUNK INDEX PARTITION: Each chunk index partition contains a part of the overall chunk index. All chunk index partitions together are equivalent to the traditional chunk index, as it stores every chunk fingerprint stored in the deduplication system. Figure 10 illustrates the shared chunk index with several chunk index partitions. The system assigns a chunk to a partition by hashing the chunk fingerprint.

A fixed number of partitions, usually much larger than the expected number of nodes in the system is created during initialization. The system ensures that each partition is assigned to a node at all times.

If a partition is not assigned to an alive node, e.g., directly after a node crash, write operations cannot complete, as it is not possible to check if a chunk is stored in the unserved chunk index partition. Therefore the fault tolerance mechanisms, which will be described later, ensure that each partition is re-assigned as soon as possible.

The chunk index is a performance-critical persistent data structure. Therefore, it is stored on SSD storage as in the single-node dedupv1 system.

BLOCK INDEX PARTITION: Each block index partition contains a part of the block index data. However, the partitioning is done differently then for the chunk index partition.

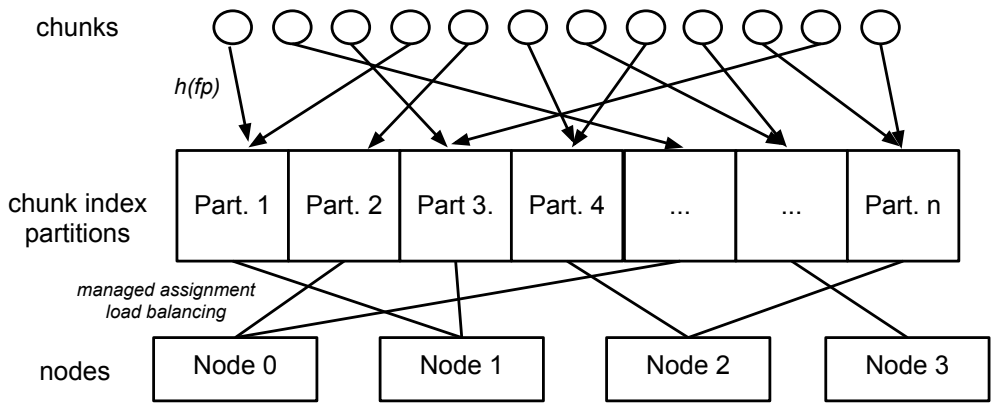


Figure 10: Shared chunk index. The system hashes chunk fingerprints to one of the equal-sized chunk index partitions. When a chunk is mapped, the deduplication node that mounts the corresponding partition becomes the responsible node for the chunk and adds an entry in the chunk index.

Each exported virtual disk (SCSI Logical Unit) is assigned to one and only one block index partition. Only the node to which a partition is assigned exports the disk over iSCSI or FC.

This design has limitations, e.g. the chunking and fingerprinting is always performed on the node where the write request arrived. If only a single virtual disk is used in a di-dedupv1 cluster, this introduces a load imbalance.

The load imbalance can be avoided by using multiple virtual disks on different machines and the handling is not as performance critical as the chunk index handling. One approach would be to implement SCSI Referral support [Eva10], which allows exporting different parts of a virtual disk on different nodes. With referral support, a single virtual disk would be exported to the clients, which is distributed over all nodes so that the resources of all nodes is used for chunking and fingerprinting.

Because chunks are automatically distributed over all chunk index partitions nearly equally, all chunk index partitions have to be served to allow successful writes. This is not necessary for block index partitions. A partition without an assigned and active volume does not need to be served. This way, it is possible to create a large amount of empty partitions that are not actually served by a node. These partitions then might be used as a reserve to scale the system as more nodes are added.

The block index partitions are also stored on SSD storage.

CONTAINER PARTITION: A container partition contains a part of the container storage. The post-deduplication chunk data is stored and managed in a container-data structure on a container partition. The container partition is stored on HDD storage.

If a node runs out of free space in its assigned container partitions, a new container partition is assigned to that node (if possible). If a node observes an above average load on one of its container partitions, the load is balanced by moving the responsibility for some of the partitions to a different node. In general, it is possible to

continue serving write requests during the re-balancing phase. However, read requests might be delayed when the read requests needs data from a container that is stored on a re-balanced partition.

CONTAINER METADATA PARTITION: A container metadata partition stores performance-sensitive metadata about a container partition on SSD storage and supports the container partition. This includes the current on-disk location of a container or the garbage collection state of a container. There is always a one-to-one mapping between container partitions and their metadata partition. The metadata partitions are always assigned to the same node as the container partition itself.

Multiple partitions of each type can be assigned to each node. The assignment is stored in ZooKeeper and is managed by the elected master node.

A partition has not to be accessible from all deduplication nodes, as long as every partition is accessible from enough nodes to tolerate node failures and to ensure an even load distribution. This feature makes it easier and also more cost efficient to extend the back-end HDD and SSD storage.

The partition setup separates the node that is responsible for the chunk index partition of a chunk from the node that serves the container partitions storing the chunk data. This separation enables the communication protocol, which will be discussed in the next chapter.

5.3 INTER-NODE COMMUNICATION

The inter-node communication is critical for the design of the di-dedupv1 system. The focus of the communication protocol design is reducing the number of messages, which are is send over the network and processed by the nodes. Especially the raw data should not be sent over the internal network if it can be avoided. This section describes the communication in more detail.

5.3.1 *Write Request Communication*

A write request is processed as follows: A client sends a write request with a block of data to a deduplication node. Internally, the data is processed in blocks of 256 KB. All SCSI requests are processed concurrently facilitating current multi-core CPUs.

The data of the request is chunked and fingerprinted on the deduplication node. After chunking and fingerprinting, it is checked for each chunk if it is already known (chunk lookup). For this, the node first hashes the fingerprint of each chunk to a chunk index partition. Next, the node determines the nodes that are currently responsible for the chunk index partitions. The partitioning information is managed using the distributed coordination service. The partition assignment changes infrequently and is small in size, so it is cached on all nodes.

The ZooKeeper's callback mechanism is used to invalidate caches. Even if a partition assignment is outdated in a node's cache, in the worst case the request is rejected and the requesting node has to update its data from ZooKeeper. In most situations, the node serving a partition can be determined without network communication.

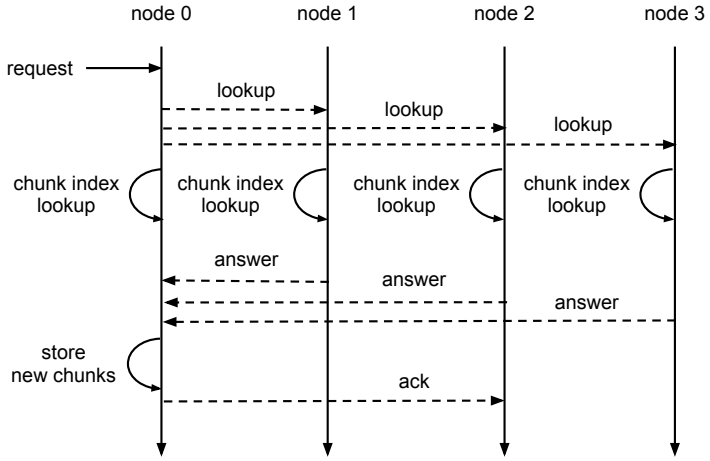


Figure 11: Communication pattern. In this example, only node 2 has a non-existing chunk index entry and thus receives an additional acknowledgment at the end.

After resolving the nodes that are responsible for the current chunks, chunk lookup request messages are sent to the nodes. Each chunk lookup message to a node consists of all chunk lookup requests to that node as well as a request id for reference.

When receiving the chunk lookup request, the requested node performs a chunk index lookup for all fingerprints in the request and answers in a single message. If there is a chunk index entry for a chunk, the node answers with the id of the partition and the id of the container in that partition that holds the chunk data. If the chunk is not found in the chunk index, the chunk has appeared for the first time and the requested node answer contains the chunk fingerprint and an "is-new" flag set. In addition, it locks the index entry, delays any further request for that chunk, and waits for an answer of the requesting node. A conflict-resolving scheme ensures that the system makes progress even when multiple nodes try to write the same data concurrently and some of the nodes crash.

When the requesting node receives an answer, it stores new chunks in a locally mounted container partition and sends an acknowledgment message back. The acknowledgment message contains the identifier of the partition and container id of the container now holding the chunk data. The requested node then updates the chunk index and answers all delayed requests based on the new information. If no acknowledgment message arrives within a given time bound, the requested node unlocks the entry and takes no further actions. Otherwise, if there are delayed requests, the requested node answers the first delayed lookup request as if it were the first and proceeds as described.

After all chunk lookup requests are answered and possible acknowledgment messages are sent, the node updates its block index. Figure 11 illustrates the communication pattern.

5.3.2 *Read Request Communication*

A read request is processed similarly to a write request. If a client sends a read request, the request is internally processed in blocks of 256 KB.

The di-dedupv1 fetches the block recipe from a local block index partition. The block recipe in dedupv1 contains the list of fingerprints that form the most recent version of the block. In addition to a chunk fingerprint, the container id, and the container partition id is stored in it.

The block data is re-assemble by loading the container data from the locally mounted container partitions. If a container partition is not locally mounted, a request is sent to the remote partition. The remote partition then fetches the container from disk and sends the container data of the requested fingerprint back to the requesting node. Thus, chunk data is transferred over the internal network in the read-path of the system.

This design is suitable for backup-oriented data deduplication systems, which are often write-optimized instead of read-optimized (see e.g., the read throughput presented in Zhu et al. [ZLP08]). Only recently, the read performance of backup data deduplication systems received more attention [KBKD12, LEB13].

Furthermore, it is assumed that read operations for data restore are usually not overlapping with write operations for data backup. Otherwise, the write throughput would suffer when the data transfers necessary for the read operations interfere with the chunk lookup request messages of the write operations.

5.3.3 *Refinements*

The deduplication nodes apply further refinements:

INTER-NODE CHUNK CACHE: All nodes have a cache containing least recently used fingerprints it has send chunk lookup requests for. It has been shown that there exists a temporal chunk locality, which is used by the cache to avoid some chunk index lookups. However, this locality alone is not strong enough to overcome the chunk lookup disk bottleneck [QD02, MB09].

LOCAL CHUNK CACHE: Also, all nodes use a page cache before the chunk index to avoid going to the SSD storage if the index page has been requested recently. This local cache is used as a write-back cache. This allows various optimizations when the cache data is written back to the underlying storage. It is ensured that the data is written sequentially instead of writing-back random pages and often multiple page updates are aggregated into a single IO operation.

This optimization is also used by the single-node dedupv1 systems.

ZERO-CHUNK HANDLING: Furthermore, the zero-chunk, which is the chunk containing only zeros, is known to occur often in real workloads [JM09, WJZF10]. This chunk is always processed locally. This special handling avoids unnecessary communication.

In the single-node dedupv1 system, a special handling of the zero-chunk is used to avoid chunk index lookup operations on it.

DATA LOCAL CONTAINER WRITES: The chunk data is always stored on a locally assigned container partition. This avoids communication at the time the chunk data is written and it creates a data locality that is used to optimize the read performance. The data of all chunks that have been written to that volume first are locally available and does not have to be fetched from other nodes.

5.3.4 *Communication Model*

After describing the communication scheme, now the expected numbers of messages per write request is estimated. The estimation will later be validated using the prototype performance evaluation.

The system may query another node for each chunk that is written to the system. With a default expected chunk size of 8 KB, this means 120,000 chunk lookup requests per second are necessary to reach a system throughput of 1 GB/s.

Here, the focus lies on estimating the number of the network messages that need to be exchanged and processed.

For this, the expected number of messages m exchanged for a write request served by a node i is computed. It is the sum of lookup messages (phase 1), response messages (phase 2), and address notification messages exchanged about new chunks (phase 3). The cluster size is n and the expected number of chunks per write is $c = \frac{\text{request size}}{\text{average chunk size}}$. The probability of a chunk to be new is p_n. The set of chunks C of size c is assigned to nodes using a randomly selected hash function. The subset of new chunks is C_n. The expected size of C_n is $p_n \cdot c$. The set of chunks assigned to a node j is denoted with $H(C, j)$. $P[i \rightarrow_p j]$ denotes the probability that node i sends a message to node j in phase p. The probability mass function of the binomial distribution is denotes as "binom" with the parameters k, n, and p. Combining these notions, the expected number of messages m is:

$$
\begin{aligned}
m &= \sum_{j \neq i} P[i \rightarrow_1 j] + \sum_{j \neq i} P[j \rightarrow_2 i] + \sum_{j \neq i} P[i \rightarrow_3 j] \\
&= 2 \cdot \sum_{j \neq i} P[i \rightarrow_1 j] + \sum_{j \neq i} P[i \rightarrow_3 j] \\
&= 2 \cdot \sum_{j \neq i} P[|H(C, j)| > 0] + \sum_{j \neq i} P[|H(C_n, j)| > 0] \\
&= 2 \cdot \sum_{j \neq i} (1 - P[|H(C, j)| = 0]) + \sum_{j \neq i} (1 - P[|H(C_n, j)| = 0]) \\
&\approx 2 \cdot (n-1)(1 - \text{binom}(k = 0, n = c, p = \frac{1}{n})) + \\
&\qquad (n-1)(1 - \text{binom}(k = 0, n = c \cdot p_n, p = \frac{1}{n})) \\
&\approx 2 \cdot (n-1)\left(1 - \left(\frac{n-1}{n}\right)^{c}\right) + (n-1)\left(1 - \left(\frac{n-1}{n}\right)^{c \cdot p_n}\right) \qquad (1)
\end{aligned}
$$

This estimation does not model the refinements discussed above, especially not the chunk caching.

The expected message size shrinks with increasing node count as few chunk requests are folded into a single message. However, the messages will still use a single Ethernet frame. Only in small clusters of two to four nodes, some request messages might not fit into a single Ethernet frame.

Figure 12 illustrates the equation (1) for different average chunk sizes and cluster sizes for a 256 KB write request and probability of 2% of new chunks, which is a realistic probability for backup runs. In general, the number of necessary messages increases faster than

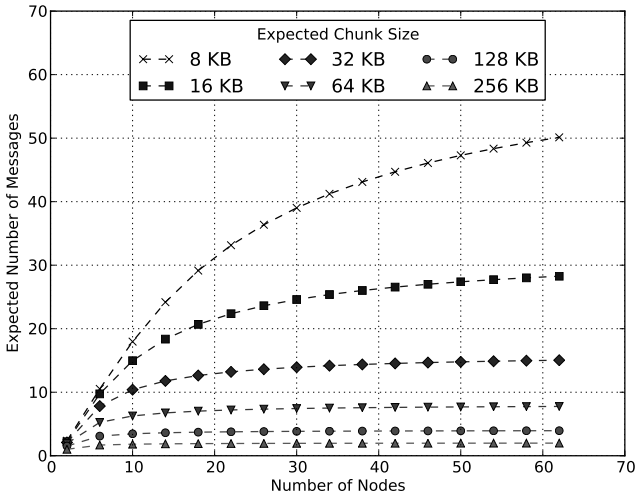


Figure 12: Expected number of messages per SCSI write request of size 256 KB.

linear when the cluster is smaller than the number of chunks per request. I call this a "relative small cluster configuration".

For example, in a cluster with 4 nodes and 32 chunks per request, the probability that a node is not responsible for a single chunk is less than 0.1%. Therefore, a message is sent to all nodes with a high probability.

If the cluster is larger, the number of messages per node stays largely constant regardless of the cluster size. For example, in a cluster with 64 nodes and 32 chunks per request, at most 32 chunk lookup messages are sent to other nodes.

This illustrates a property of the communication scheme: In a single-node deduplication system, doubling the chunk size usually increases the throughput by a factor of two if the chunk index is the bottleneck. However, doubling the chunk size is not reducing the number of messages exchanged by 50% if the cluster has still a relative small cluster configuration. If the communication instead of the actual chunk index lookup is the bottleneck, this implies that increasing the chunk size is not increasing the throughput as much as it does in a chunk lookup bottlenecked system.

In Chapter 6, the architecture is evaluated based on a prototype implementation. The limits on the message rates the communication system is able to process are also investigated as they determine the achievable throughput and scalability.

5.3.5 *Node Communication Center*

As tens of thousands of messages per second are exchanged between the nodes, the cluster needs a well-engineered inter-node communication system that minimizes the amount of information exchanged and is optimized for very high messages per second rates. Other properties as latency and data throughput are only of secondary importance for the communication sub-system.

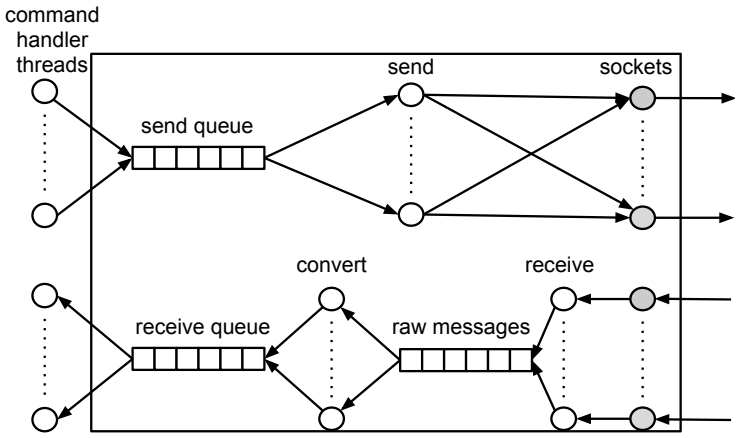


Figure 13: Architecture of the communication system.

All messages are received and sent through a highly parallelized central communication system (Figure 13) within each node, called "communication center". The communication center keeps TCP/IP connections open to all known cluster nodes.

All incoming messages are first delegated to a series of converter threads that convert the raw messages to higher-level objects. The converter also checks the message checksums. The objects are then delegated to a thread pool using a concurrent queue where the messages are handled according to the type of the message.

Outgoing messages are placed in a concurrent queue, which is part of the node's communication center. The messages are consumed from a series of sender threads that serialize the messages and send them over the outgoing sockets.

The communication center is engineered to optimize the processed number of messages per second and was the result of a comparison with other designs, e.g., using SCTP messages [SXM+00] or using a design using a single thread for sending and receiving messages before passing the messages to a thread pool.

5.4 FAULT TOLERANCE AND LOAD BALANCING

This section discusses the design of the fault tolerance of di-dedupv1 and the load balancing approaches build into the design of di-dedupv1.

5.4.1 *Fault Tolerance*

Fault tolerance is an important property of storage systems or backup storage systems as it is critical to be able to continue operations within a short time frame after a failure because of the limited size of backup windows. However, di-dedupv1 is not designed as a highly-available system that continues operations even if components fail. The system is designed to handle any hardware faults without corrupting the data and be able to restart its operations within a sufficiently short time span.

Fault tolerance needs to handle all sub-components of the system. One assumption is that the back-end storage (HDD/SSD) is an enterprise-class storage system using appro-

priate RAID schemes, disk scrubbing, or even intra-disk redundancy [PGK88, SXM+04, SDG10, GMB10]. In the following, the assumption is that what is stored on the shared storage system is stored there reliable and highly available.

Nevertheless, CRC checksums [PB61] are calculated for all persisted data (container, chunk index entries, block recipes, log entries). Therefore, most unintentional modifications of the persisted data, e.g., due to silent data corruption [NCL+09, BADAD+08], can be detected and the system is stopped, but it is not possible to correct errors. The CRC checksums build an additional level to detect errors, but still the storage hardware needs to provide a sufficient level of reliability.

Other important assumption for the handling of faults is that di-dedupv1 consists of at most a few dozens of cooperating nodes that are managed by the same organization. Therefore, the system is assumed to not have to deal with adversarial behavior [PSL80]. Instead the "fail stop" model has been the driving principle [SS83].

The system relies on the failure detector of the ZooKeeper system, which uses a heartbeat approach to discover node crashes. While such a failure detector has drawbacks (see [CTA02]), it works well in practice.

When the current master node is informed about a node crash, the master node reassigns all partitions to other nodes that also see the storage of the partitions. After all partitions are mounted on the new assignees and the necessary recovery operations are complete, the operations can continue. When the current master node fails, a new master node is elected and the same operations are performed.

For the following description, it is assumed that only one node crashes at a time. The recovery operations for the partition types differ:

CHUNK INDEX PARTITION: The chunk index partition is especially important because the complete system halts if it is not available. Therefore the operations must resume after at most a few seconds. This is achieved in di-dedupv1 by assigning all chunk index partitions of the crashed node to other nodes. The new node immediately mounts the partition and starts processing chunk index lookup messages.

Chunk fingerprints that are already stored in the back-end SSD storage are answered correctly. However, as write-back caches are used for the chunk index, the back-end SSD storage might not have information about all chunks that have already been stored. This leads to a situation where chunks are stored in containers, but the chunk index has no entries for these chunks. Chunks that have been stored previously in the write-back cache, but whose entries have not yet been persisted to disk are classified as new chunks. They are "false negatives" as these chunks have actually already been stored.

The exact chunk index can be recovered by replaying the operations log, but this might take hours. Therefore, the exact state of the chunk index is not recovered immediately. Instead, the system goes into a recovering state. Chunk entries that are not found in the back-end storage in the recovering state are assumed to be new and stored on disk. However, they are marked in the chunk index that they are stored during a recovery.

Eventually, the operations logs of the container partitions are replayed. At this point, the system notices the mismatch between the container id and the id stored in the chunk index, which is pointed to the newer container. Because the recovering flag is set, this mismatch is accepted and corrected.

The system is not fully consistent directly after the crash recovery, but it is working correctly and is able to eventually reach a fully consistent state with low overhead.

BLOCK INDEX PARTITION: The block index partition is subject to a different value proposition. As the system is used for backups, a downtime of up to a minute is acceptable for the block index partition. However, delivering outdated results is not.

Therefore, when another node takes over the block partition, it reads all un-replayed log entries of the block index partition's operation log and rebuilds the exact state of the block index.

All operations on other volumes can continue while the partition is being rebuild. As only relatively small number of blocks is stored in the write-back cache of the block index, a rebuilding process completes within the set time limit.

CONTAINER PARTITION: The recovery of a container partition and its metadata partition consists only of mounting the partition and the corresponding management partition at another node and recovering some state of the container partition. This state is small as it consists only of the current redirections of container ids and the used/free bitmap of the container on-disk locations. Therefore, the recovery process finishes within a few seconds.

Placing partitions on the shared storage allows a fast recovery from node failures while avoiding explicit replication in the communication protocol.

5.4.2 *Load Balancing*

A topic related to the fault tolerance is load balancing as in di-dedupv1 similar techniques are used for both. Again, the load balancing also relies on the storage design. Similar to the handling of crashes for fault tolerance, each partition type uses different load balancing mechanism.

CHUNK INDEX PARTITION: The system always tries to distribute the chunk index partitions uniformly among the nodes. Given a fixed amount of nodes, re-balancing is not necessary because the direct hashing ensures a near-uniform distribution of chunks (and thus chunk lookups) to the equal-sized chunk index partitions. This follows from the basic theorem of balls-into-bins games stating with a high probability that if m items are distributed independently and uniformly at random to n nodes with $n \log n << m$, no nodes stores more than

$$\frac{m}{n} + O\left(\sqrt{\frac{m}{n} \log n}\right)$$

items [RS98, Eff11]. Here, the static scheme with equal-sized partitions ensures that the data distribution is not introducing an imbalance. A more dynamic scheme would require methods like consistent hashing, in which the partitions with a high probability differ by a factor of $O(\log n)$ in size. In the basic form, this would introduce an imbalance [KLL+97].

However, different chunks are used with different probabilities leading a chunk usage skew [MB09]. This may lead to an uneven distribution of chunk lookup requests

to partitions and nodes. Due to the special handling of the zero-chunk as well as the inter-node chunk cache, the chunk skew does not lead to a significant imbalance on the partitions.

BLOCK INDEX PARTITION: It becomes necessary to re-balance a block index partition if a deduplication node exports heavily used SCSI volumes. The re-balancing moves a block index partition from one node to another node. Without further action, this would require a reconfiguration of all iSCSI clients to change the IP address manually.

The alternative chosen for the design of di-dedupv1 may use iSCSI Active/Passive Multipathing. Using multipathing, a partition can be re-assigned from one node to a limited, pre-determined set of other nodes. Even the limited choices for block index reconfigurations allow a suitable load balancing unless the load to a single virtual disk is very high. A related load-balancing scheme has been proposed by Brinkmann, Gao, Korzeniowski, and Meister for the skewCCC peer-to-peer (P2P) system [BGKM11].

The re-balancing requires that all incoming traffic is stopped while the rebalancing is in process. Therefore, the latency during a re-balancing process is exceptionally high, but latency is not a concern in backup deduplication system as long as the SCSI connection does not time-out. To avoid this, the system evicts all dirty pages from the block index write back cache before moving the partition. The new node can then immediately provide the SCSI volumes after the remount.

CONTAINER PARTITION: The container partitions are the easiest to balance. The load is measured in terms of the fill ratio and the write and read operations. The load balancing ensures that all nodes have a container partition mounted with sufficent free space if this is possible.

A re-balancing invalidates the partition-to-node caches mentioned in Section 5.3. If a node receives a request that requires a partition that the node is no longer mounting, the node answers with an error code. The requesting node then updates its cache and resends the request to the correct node. Using this lazy scheme, there is no need for a consistent cache management.

It should be noted that the prototype evaluated in Chapter 6 writes all operation log information that is necessary for the recovery process, but the recovery and the load balancing are currently not implemented.

6

EVALUATION

This chapter presents the evaluation results using a prototype based on the dedupv1 single-node system. The evaluation uses a cluster of up to 60 nodes. Of these 60 nodes, up to 24 nodes have been used as deduplication cluster nodes.

The most important properties of interest in this chapter are the throughput and the scalability properties. Also, the performance results are related to the communication model and it is shown that the number of processed messages per node is the limiting bottleneck of the prototype. Based on this, the limitations of this approach and its scalability on even larger clusters of machines are predicted.

The chapter is organized as follows: In Section 6.1, the evaluation methodology is explained. The throughput and scalability results are presented and discussed in Section 6.2. It can be shown that the scalability results can be explained by the communication pattern and the ability of the nodes to process messages. Section 6.3 explores the throughput and scalability limitation that is induced by the communication system. It also predicts the scalability for larger cluster sizes and other chunk configurations.

6.1 EVALUATION METHODOLOGY

This section describes how the distributed dedupv1 approach is evaluated using a prototype. At first, the data stream generation model is presented. Afterwards, the concrete system setup and configuration is described.

6.1.1 *Data Stream Generation*

The evaluation of file and storage systems is a hard task [TZJW08]. Good benchmarks for evaluating file and storage systems are available, e.g., filebench [Wil08] for (local) file systems and IOMeter for network block storage [Sie06]. However, these benchmarks fail to evaluate the throughput of deduplication storage systems. While the concrete written data pattern is not significant for storage systems without data deduplication, it is performance critical in deduplication systems.

Still to this day, there is no agreed upon public benchmark for deduplication systems (see, e.g., [TBZS11, TMB+12]). Some systems are evaluated using non-public data sets, which are either gathered ad-hoc or which are based on customer data [ZLP08, YJF+10, LLS+08]. In addition, the throughput of other systems is not evaluated using a prototype or a real implementation at all, e.g., [WJZF10, BELL09].

One of the descriptions of a deduplication system where the synthetic data used for the evaluation is explained is HYDRAstor [DGH+09]. The data generator has a configurable deduplication ratio and distributes duplicates evenly in the data stream. However, in real systems duplicate chunks cluster together forming runs of consecutive redundant or unique chunks. As pointed out by Tarasov et al. the run lengths affect the throughput [TMB+12]. It is likely that the benchmark fails to capture important characteristics of duplication workloads. It can be assumed that the same synthetic benchmark is also used to evaluate HydraFS [UAA+10].

DEDISbench is a block-based benchmark for data deduplication systems [PRPS12]. The chunk usage distribution models realistic pattern, but the important run lengths distribution is not modeled. Redundant and unique chunks are distributed uniformly. Also the benchmarking tool is not targeted for backup workloads, as it is not modeling that data is only slightly changing from backup run to backup run.

Tarasov et al. presented an approach for deduplication-aware data generation on the file system level [TMB+12]. They use an existing file system as the basis. A Markov model is used to decide which files are newly created, modified, or deleted for each mutation. Other not further specified statistical distributions are used to model in-file changes.

For this evaluation, an artificially generated data stream is used to evaluate the data deduplication system. The data stream generator is designed to closely resemble a block-based backup over multiple weeks. The generation model is based on data deduplication traces of the *UPB* data set introduced in Section 2.4.

The source code and other necessary data of the data generator are available at `https://github.com/dmeister/dedupbenchmark/tree/online`. The public availability of the data stream generator can help other researches to perform similar tests and to provide comparable evaluation results.

The basic idea of the data generation is that a backup data stream consists of data that is new to that backup generation, data that has not been stored in the previous backup run, but is redundant within the same backup (intra-backup redundancy), and data that has already been stored in previous backup runs (inter-backup redundancy).

Following this general model, a probability distribution of the run lengths for three types of data is extracted from the *UPB* data set.

In data deduplication systems, the first backup generation has different properties than later backup generations. The system is empty before the first generation, so that no inter-backup redundancy can be used. In the second generation, the deduplication system can use chunk fingerprints already stored in the index. Later backup generations are then similar to the second one. Therefore, these two important cases are considered while generating the data streams for the evaluation.

Furthermore, Tarasov et al. [TMB+12] as well as Park and Lilja [PL10] point out unique and redundant chunks are not distributed uniformly in the data stream, but they come in "runs". If a chunk is unique, there is a high probability that the next chunk is unique, too. Besides the basic property of the deduplication ratio, this distribution of chunk run lengths is a significant property to generate artificial data for the evaluation of deduplication systems. Park and Lilja observed that the average run length varies little between the different data sets they have analyzed [PL10]. However, the maximal run length and the skewness varies, which has an impact on the throughput observed in their simulations.

The data for the first backup generation written to the deduplication system is generated as follows: As long as the backup generation is not finished, the next from two states

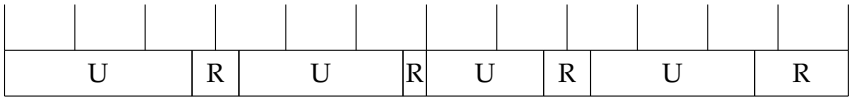


Figure 14: Illustration of the data pattern of the first backup generation. R = redundant data, U = unique data. The length of the blocks is randomly chosen according to a probability distribution extracted from real world data. The marker in the upper half denotes the chunks.

is picked (U for unique data or R for redundant data) and chosen independently and randomly how much data is continuously generated from the chosen state. The length is chosen based on the empirical distribution extracted from these existing traces. It is not possible to fit the empirical distributions to well-known probability distributions. The probability distributions of U and R are highly skewed.

The run lengths of redundant data observed in the trace file used for the data generation have a mean of 143 KB, a median of 16 KB, and a maximum of 3,498 MB. The run lengths of unique data have a mean of 337 KB, a median of 19 KB, and a maximum of 6,582 MB.

The data for the unique state U is generated using a pseudo random number generator with a sufficient sequence length [MN98]. To generate redundant data (R), a number of redundant blocks is pre-generated and then one of these blocks is picked independent at random via an exponential distribution, as an exponential distribution is a good fit for the reuse pattern of redundant chunks.

The second data generation is generated similarly to the first one. Figure 16 shows an illustration of the data generation for the second generation.

The empirical distributions for new unique data (U), intra-backup redundant data (R), and inter-backup redundant data (BR) are extracted from the *UPB* data set. The probability distributions are also extracted from the same real-world traces. The run lengths of inter-backup (temporal) redundant data observed in the trace file used for the data generation have a mean of 17.84 MB, a median of 15 KB, and a maximum of 30.1 GB.

Given that the complete data set has a size of 440 GB, the largest run of data, which already existed in the previous backup run, covers around 6.8% of the complete data set. Not a single chunk that is unique or only redundant within the same backup run breaks this 30 GB run. The mean run length of intra-backup redundant data is with 325.4 KB (median 13.3 KB, max 160.9 MB). New unique data has a run length distribution with a mean of 162.4 KB, a median of 16.0 KB, and a maximal run length of 271.4 MB.

In addition, the probability distribution of state switches between the states (U, R, BR) is extracted. The most common state switch is to the BR state, as up to 98% of the data is redundant data. It is assumed that the Markov condition holds for state switches, which means that the probability for a state switch does not depend on previous state switches. The Markov condition to switch data generation states is also assumed by Tarasov et al. [TMB+12]. Figure 15 shows the switch probabilities that have been extracted from the real-world traces.

The generation for U and R is the same as in the first generation. The intra-backup redundant data is generated to contain the same data as in the first run.

All generated run lengths are rounded up to the next full 4 KB block to closer resemble a block-based backup of a file system.

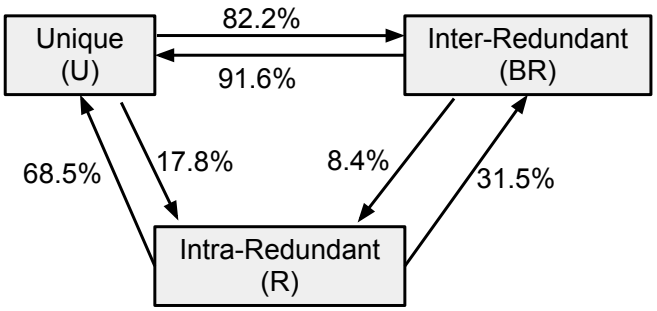


Figure 15: Switch probabilities between second-generation states, extracted from real-world traces.

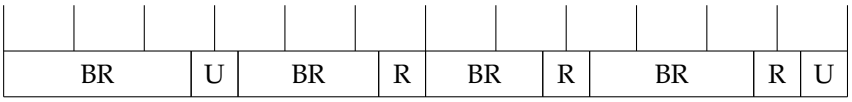


Figure 16: Illustration of the data pattern of the second backup generation. BR = inter-backup redundant data, R = intra-backup redundant data, U = unique data. The length of the blocks is randomly chosen according to a probability distribution extracted from real world data. The marker in upper half denotes the chunks.

In the following evaluation, only the first and the second backup generation are simulated. Therefore, the benchmark is not able to capture long-term effects. However, this is acceptable because these effects are not the focus of this research.

Another limitation of this evaluation approach is that different clients do not share data in this model as the underlying trace data has no information about data sharing and the traffic characteristics of cross-client data sharing in a backup system. Without such information, is is not be possible to model data sharing with confidence. The data generation on the different backup clients is started with different random seeds so that the generated data is unique per client. This is a worst-case assumption for the deduplication system. Any data sharing between clients would increase the data deduplication ratio and throughput.

Furthermore, the generation model is not sufficient for other types of data deduplication. For example, it is insufficient for delta-based deduplication because the sub-chunk data is not considered. Another aspect not modeled here is the compressibility of the generated chunks – the generated data is not compressible at all.

The data stream generation model has been validated against a real data set from workgroup computers with a similar data deduplication ratio as the trace data using the single-node dedupv1 system. The throughput results using real data and artificially generated data differs within a margin of 5%. Therefore, the data stream generation captures the essence of the data pattern and localities that are needed for the evaluation of the data deduplication prototype.

The approach of the workload generation allows an evaluation of a fingerprinting-based deduplication system without using proprietary ad-hoc datasets. It allows to re-assemble

backup runs of arbitrary size. It generates backup data with similar deduplication ratios and similar data layout of redundant and non-redundant parts of the data stream.

An early version of the scheme has been described in previous work of the author [MB10a]. The two most important differences are:

1. The data is now generated during the test run itself instead of pre-generating data before the test run. The previous version had to read the pre-generated data from a storage system during the tests. Now, the benchmark environment therefore does not have to include large and fast disk storage on the workload generating clients and allows the generation of arbitrary large data sets.

2. The generation of redundant data is now modeled more accurate. It has been observed that some redundant chunks are referenced more often than other redundant chunks [MB09]. The skew in the usage of redundant chunks is incorporated into the data generation algorithm.

6.1.2 *Evaluation Setup and Configuration*

After the data generation for the evaluations has been explained, more details about the evaluation setup and configuration are presented in following.

For the purpose of the evaluation, a 60-node cluster has been available. The cluster is usually used for HPC scientific computation and it is not possible to modify the hardware to closer resemble a storage platform. All nodes are identical 1U Fujitsu RX200S6 servers with two 6-core Intel X4650 CPUs running at 2.67 GHz and having 36 GB RAM. All nodes have GBit Ethernet and InfiniBand 4xSDR HCAs. Both networks are fully switched. The CentOS 5.4 distribution is used with a custom 2.6.18 Linux Kernel. The kernel uses the performance-related kernel patches of SCST for optimal performance.

The di-dedupv1 system has designed to work with less-powerful deduplication nodes and to scale horizontally. Most CPU power and the RAM available in the evaluation cluster have been unused.

The 60-node cluster has been partitioned as follows. Up to 24 nodes built the deduplication cluster. Up to 24 nodes are used as clients for workload generation. Up to 12 nodes are used as iSCSI targets to provide the shared storage emulation (6 SSD nodes, 6 HDD nodes). There is always a 1:1 correspondence between deduplication nodes and clients and a 4:1 ratio between deduplication nodes and storage emulation nodes if more than 4 deduplication nodes are used.

The internal communication of the deduplication cluster can be switched between GBit Ethernet and IP over InfiniBand (IPoIB), which is a protocol to tunnel IP packets over InfiniBand. Unless otherwise noted, the GBit interface is used. The system is configured to use 8 KB content-defined chunking using Rabin's fingerprinting method. Chunk data is stored in 4 MB containers.

The system is configured not to compress the deduplicated chunks as it is usually done in deduplication systems [ZLP08, MB10a, QD02, WDQ+12]. Given enough CPU power, a fast compression may increase the throughput by reducing the amount of data written to the back-end storage [KTSZ09]. The inter-node chunk cache is set to 2 MB per deduplication node.

The client nodes generate backup data using the data generation scheme described above and write the data to volumes exported by the deduplication system via iSCSI.

Each client writes 256 GB data per backup generation. Therefore, the evaluation is scaled up to 12 TB of written data in the largest test configuration. The iSCSI traffic uses IP over InfiniBand (IPoIB) [CK06] to ensure that the GBit Ethernet network is free for the internal communication. A real deduplication system would instead have two or more separate Ethernet connections. Additionally, this ensures that the iSCSI transfer throughput is not the bottleneck.

The cluster has no shared storage available that would have been appropriate for this use case. As hardware changes were not possible, compute nodes are used to provide the shared storage via a software iSCSI target. The SSD storage is simulated by modifying the SCST iSCSI target framework so that a RAM-disk-based volume has the latency characteristics of SSD storage. The SSD performance model proposed by El Maghraoui et al. [EKJP10] is used to simulate the SSD latency characteristics. The extra latency for a 4 KB read is 245 µs, the write latency is 791 µs. The general processing overhead, e.g., of iSCSI, adds additional latency.

The shared disk storage is simulated in a similar way. As only sequential write operations (4 MB container commit) are performed in the tests, the latency is adjusted accordingly to 1,700 µs. The data itself has not been stored, as the data is never accessed again in the evaluation.

The storage emulation and latency modeling is validated using the single-node dedupv1 system using real hardware. The emulation has a similar performance than the real hardware. The differences can be explained, e.g., as a different SSD model is used than it was used by El Maghraoui et al. to generate the SSD performance model.

The deduplication nodes are stopped and restarted between the first and second data generation as well as all operating system caches are cleared after the system initialization and the simulated backup runs.

6.2 SCALABILITY AND WRITE THROUGHPUT

The write throughput of a deduplication system targeted at backup workloads is the most important performance characteristic besides the storage savings as a backup usually has to complete within a specific time frame, called "backup window" [ZLP08].

Since the design of this deduplication scheme ensures that the same redundancy is found and eliminated as in an equivalent single node deduplication system, the evaluation focuses on the throughput of the cluster.

For the evaluation, the cluster size is scaled from 2 to 24 nodes using a GBit Ethernet interconnect. The cluster throughput for the different cluster sizes is shown in Figure 17. The highest throughput is measured with 24 nodes reaching 3.0 GB/s for the first data generation and 5.2 GB/s for the second data generation. The figure also shows dashed bars, which denotes the expected performance for linear scaling. As it can be seen there, the throughput is not scaling linearly. In fact, the throughput scales exponentially with a factor of at least 0.86 for the range of cluster sizes evaluated. This means that when backup size and the cluster size are doubled, the throughput increases by at least 86%. The reasons why linear scalability is not achieved with the chosen design will be explored later.

Figure 18 shows a breakdown of latency factors for a write command and their relative mean timings in a setup with 24 cluster nodes. A write command consists of 5 separate

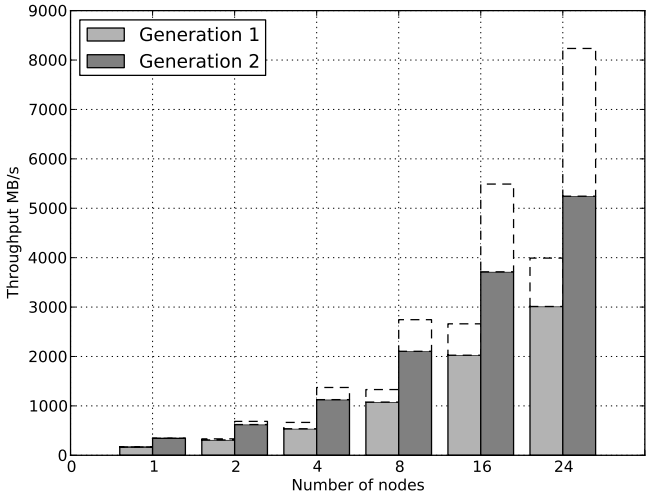


Figure 17: Throughput using Gigabit Ethernet interconnect for different cluster sizes. The dashed bars show a linear scaling.

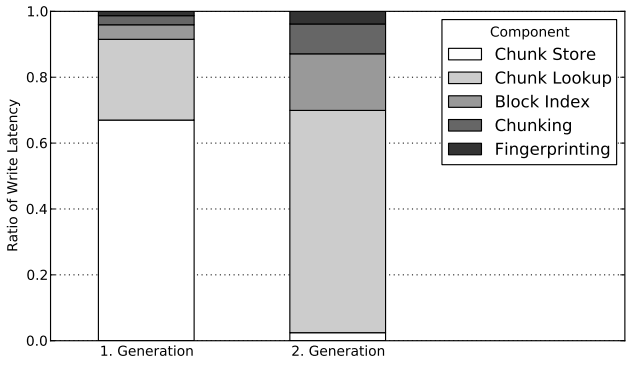


Figure 18: Relative latency of different components for a write operation in a 24-node cluster.

Table 2: Average per-node/total throughput in MB/s using different intra-node interconnects.

Node Count	Gigabit				InfiniBand (IPoIB)			
	1. Generation		2. Generation		1. Generation		2. Generation	
1	166	166	343	343	166	166	342	342
2	155	310	311	622	156	312	304	608
4	134	536	281	1,124	134	536	278	1,112
8	134	1,072	263	2,104	133	1,064	255	2,040
16	126	2,016	232	3,712	127	2,032	229	3,664
24	125	3,000	218	5,232	124	2,976	209	5,016

phases: The chunking and fingerprint, the chunk lookup, which is executed in parallel for all chunks, the chunk write, and the handling of the block index.

In the first generation, the chunk writing and the chunk lookups dominate with a total share of 91.9%. The high chunk write share of 66.9% is caused by the high percentage of new data. The next larger component is the chunk lookup with 24.5%, which represents the time to collect necessary chunk information from all nodes. The remaining components (block index 4.4%, chunking 2.7%, fingerprinting 1.3%) play a smaller role.

In the second generation, the chunk write share (2.4%) nearly vanishes, because most data is now known and does not need to be stored. As a consequence, the fractions of chunk lookup (67.4%), block index (17.1%), chunking (9.0%) and fingerprinting (3.8%) rise. The chunk lookup includes the chunk index lookup itself as well as the communication with other cluster nodes for the remote chunk lookup. The breakdown of the latency factors shows that the chunk lookup phase becomes important in later backup generations to achieve a high throughput.

For all results until now the chunk lookup messages have been exchanged over a separated GBit Ethernet network. While GBit Ethernet is still commonly used, it is not high-end hardware. To see if a high-end network makes a difference on the throughput and the scalability results, some experiments have been repeated using the InfiniBand network of the HPC cluster. Here, an InfiniBand network is used as a drop-in replacement using IPoIB for the intra-node communication. The results are shown in Table 2. The throughput is similar to the GBit Ethernet results.

Besides the overhead of IPoIB instead of "raw" InfiniBand, another factor involved in this result is that the InfiniBand network has two usages in this setup. The incoming SCSI traffic is also using the InfiniBand network. Either the InfiniBand network is bottlenecked with the two usages or this is another indication that the internal processing capabilities are the bottleneck in the evaluation, and not the network itself. The additional evaluations focused on the network, which will be presented in the next section, support the second assumption.

6.3 COMMUNICATION-INDUCED LIMITS ON THROUGHPUT

The chunking and fingerprinting scales with the number of CPU resources and therefore should inhabit a linear scalability. Also, the storage of the new chunk data is scaling with the size of the back-end storage and also can be scaled by using multiple independent

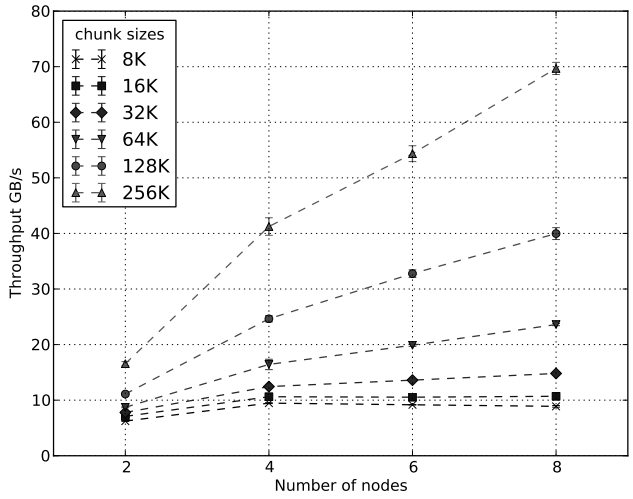


Figure 19: Theoretical limits on cluster throughput based on network communication using GBit ethernet.

back-end storage systems. In addition, Figure 18 shows that the chunk write is not the bottleneck. A similar argumentation holds for the SSD-based chunk index lookup. The chunk index lookups are assumed to be scalable with the number of nodes by adding more storage servers with increasing cluster sizes. Therefore, this section focuses on an analysis of the communication and the interconnect.

The central factors are logged during the evaluation runs whose results have been presented above. In the evaluation with 4 nodes, each node sends on average 6,573 messages per second with an average message size of 118 bytes. Each chunk lookup request message contains on average requests for 3.6 chunks (see Section 5.3 for the underlying mechanisms).

The aggregation of multiple chunk requests into a single chunk lookup request message reduces the load in smaller clusters, but the effect vanished for larger cluster sizes. In the case of a 16-node deduplication cluster, the average message size decreases to 58 bytes and each node has to send 19,528 messages per second on average. Each message includes on average requests for 2.3 chunks. This drops even to 1.5 chunks on average for clusters with 24 nodes. This validates the communication model established in Section 5.3.

The important observation is that the number of generated messages per node based on incoming SCSI writes cannot increase to a number higher than the number of chunks per request. In this case, the number of chunks per message is around 1.0. The system only sends a lookup request messages to a node if at least one chunk can be found there. For each incoming 256 KB data block, a node generates at most as much messages as there are chunks, regardless of the number of nodes in the network.

This explains the sub-linear scalability observed in the evaluation measurements for the cluster with up to 24 nodes. In that node range, each increase in the node count increases the number of messages a node has to process in two ways:

1. More nodes result in more write requests, which create more chunk lookup requests.

2. The average number of chunks per chunk lookup request is reduced, which means more messages per write request. However, with each increase in the number of nodes the ability to process messages increases only linearly.

This opens up a new question. Assuming that a linear scaling of the actual chunk index lookups on the SSD storage using faster and/or more SSDs, what is the performance limit caused by the communication overhead.

This can be formulated as the communication-induced limit on the throughput. The focus on the impact of the communication and networking aspects allows deriving a limit using the current setup and architecture.

This is evaluated using a prototype that simulates the communication pattern of di-dedupv1 system using the same communication infrastructure on the same hardware as the full prototype evaluation. However, it is only possible to repeat the evaluation on the same hardware with up to 8 nodes.

Each node generates one million requests that represent writes of size 256 KB. For each request, a node randomly maps chunks to nodes and sends the fingerprints to the chosen ones. All chunks to the same node are aggregated together and sent as a single chunk lookup message. A receiving node declares each chunk as new with a probability of 2% to reflect the rate of new data in consecutive backup runs. This models the communication pattern of the chunk index lookups in di-dedupv1, while eliminating all other factors.

Figure 19 shows the theoretical achievable maximum throughput with different cluster and chunk sizes when focusing on the chunk lookup network traffic (with 95% confidence intervals). The cluster throughput for 8 KB chunks starts at 6.2 GB/s and rises to 9.4 GB/s for 4 nodes after which it slowly degrades to 8.8 GB/s for 8 nodes. This shows that the message-processing rate of the nodes is the bottleneck rather than the network itself, because the total number of messages sent during the experiment has no impact on the performance. In a 4-node cluster with an average chunk size of 8 KB each request results in expected 32 chunks such that each node has to answer expected 8 chunks. Hence, a request generates at least 3 messages with high probability. In an 8 nodes cluster each node has to answer expected 4 chunks, which results in at least 7 messages per request. The slight performance decrease comes from the increased waiting time, as the nodes must process more messages.

The results indicate that the nodes' ability to process messages is the bottleneck in these experiments. However, as explained before, the number of generated messages only grows linearly after a certain cluster size is reached and the configuration is no longer "relative small". The experiments reveal this even for such small clusters with only up to 8 nodes if large chunk sizes are used. The turning point where the communication system would stop to be a scalability problem is approximately 32 nodes for 8 KB chunks and 16 nodes for 16 KB chunks.

Another source of scalability problems could be the network switch. It would be possible to reach a point where the switch is not able to deliver the messages per second. The available results show no sign of saturation in the GBit Ethernet network switch. As long as all nodes are connected to a single HPC-class switch, this may not be an issue.

The overall throughput improves if the message-processing load per node decreases. This can be achieved by either adding more nodes while fixing the number of requests or by increasing the expected chunk size. The latter can be seen in Figure 19 and reduces the number of chunks per request and thus, the maximum number of messages for collecting

the chunk information. The best performance is reached for 256 KB chunks, where each request only contains one chunk.

As the actual chunk index lookup on the SSD storage has been ignored in this setting, these limits are also valid for deduplication systems that hold the complete chunk index in RAM instead on an SSD. It does not hold for other types of deduplication system like approximate deduplication systems or systems, where it is not necessary to lookup most chunks over the network, e.g., by using a locality-preserving caching scheme [ZLP08].

Another way to increase the throughput in a communication-limited system is to further optimize the communication system. For example, decreasing lock contentions would actually have an impact on the throughput. While such optimizations are certainly possible, they are probably not trivial as the communication system is already carefully engineered, profiled, and optimized.

The major conclusion of this section is that the network will be able to scale further. Factors like chunking and fingerprinting scale linearly with the number of nodes. If also the storage components are scaled with the number of cluster nodes, the overall system is scalable.

7

CONTRIBUTION

In this part, a novel design for an exact inline-deduplication cluster has been introduced. The cluster deduplication system detects the same redundant chunks as a single-node deduplication system, while still using small chunk sizes. Key elements of the design are combination of the communication protocol and the storage design to avoid sending large amounts of data over the network. In the write-path, only small messages containing only the fingerprints of the data are exchanged over the internal network.

A prototype based on the dedupv1 system has been used to evaluate the performance of the design. The evaluation shows that it is possible to combine exact deduplication, small chunk sizes, and scalability within one environment while using only commodity GBit Ethernet interconnections.

A combination of the prototype-based evaluation, a modeling of the communication pattern, and simulations has been used to predict the scalability for larger cluster sizes and to explore the limits of the scalability. The results show that the scalability of the design is limited when the cluster is small. However, for larger chunk sizes and larger cluster sizes, the scalability improves.

Part III

BLOCK AND FILE RECIPE MANAGEMENT AND USAGE

8

INTRODUCTION

The last part presented an architecture to scale a deduplication system by using multiple cooperating nodes as a cluster. The key to achieve this was the design of the storage layer together with the communication protocol that enables the necessary exchange of messages between cluster nodes. However, the organization of each node, the data structures, and how the data is stored and retrieved from persistent storage has essentially not been changed in comparison to dedupv1. For example, the chunk lookup disk bottleneck is still avoided by using SSD storage.

In this part, the focus is different. The focus here lies on two novel techniques related to the usage and management of block and file recipes.

The first technique is a combination of deduplication-aware compression schemes to reduce the size of block and file recipes by more than 92% in all investigated data sets. In a backup deduplication system with very high deduplication ratios and long data retention periods, these savings significantly reduce the overall storage requirements.

The second technique uses a novel caching scheme to overcome the chunk lookup disk bottleneck. It will be shown that this scheme captures the locality of the previous backup run better than the state-of-the-art alternatives, e.g., the locality-preserving container caching schemes proposed by Zhu et al. [ZLP08]. Therefore, new approach requires less chunk-lookup-related disk IO operation to perform a backup than the existing approaches. Especially with techniques that offload the chunking and fingerprinting to the clients [LGLZ12, EMC12], the number of IO operations on the backup system is still an important factor for the ingest performance.

Previous research often focused on the chunk index. The storage of the block and file recipes is largely being ignored because they are not as critical for the system throughput as the chunk index: While per throughput of 100 MB/s around 12,000 chunk index lookups are necessary, the file or block recipes require practically no disk IO in a backup deduplication system.

Under the assumption of fully sequential writes, a system with a throughput of tp, an expected chunk size of cs bytes, and a block size of bs bytes (or an average file size of bs bytes) need to process $bps = {}^{tp}/_{bs}$ blocks per second. By default, the dedupv1 system uses a block size bs of 256 KB. Therefore around 400 blocks need to be processed per second per throughput of 100 MB/s. Only the contents of these blocks need to be fetched from disks.

Each block consists of ${}^{bs}/_{cs}$ chunks on average. In the default configuration, the data of a block recipe can be estimated with ${}^{bs}/_{cs} \cdot f = 640$ bytes with f as the fingerprint size of 20 bytes if SHA-1 is used. Under the assumption of sequential write operations, the access to the block recipes is also sequential. If an ordered persistent data structure as a

B-Tree [BM72] or a Log-Structured Merge-Tree (LSM-Tree) [OCGO96] is used to store the recipes, then the disk accesses are also likely to be sequential. In these cases, a transfer rate of $2 \cdot {}^{tp}/_{bs} \cdot {}^{bs}/_{cs} \cdot fps = 2 \cdot {}^{tp}/_{cs} \cdot f = 480\,KB$ per 100 MB/s of read/write-throughput is sufficient to read and write the block recipes. Such a transfer rate is unlikely to be a bottleneck even in high-performance backup systems.

Nevertheless, solely focusing on the chunk index is misleading. The file and block recipes are an important element of the system architecture.

One reason is the storage that is needed to store the index itself: A file or block recipe stores f bytes per referenced chunk. Therefore, the recipes grow with the size of the logically stored data. Per TB of logically written data using an average chunk size of cs bytes, at least ${}^{2^{40}}/_{cs} \cdot f$ bytes are required to store the recipes. With a chunk size of 8 KB [ZLP08, MB10a], a TB of logically written data accounts for 2.5 GB recipe data. This stands in contrast to the chunk index, which stores one entry per unique chunk fingerprint, and therefore grows only with the physical capacity of the system.

The used physical capacity is only growing slowly over time. For example, Zhu et al. report a daily deduplication ratio of 0.98 [ZLP08]. The data sets introduced in Section 2.4 have a weekly deduplication ratio of 98% or higher in most weeks.

To estimate the storage usage of block and file recipe data, I focus on the recipes and the post-deduplication data storage as major usage of back-end storage in a data deduplication system. Assuming an internal data deduplication of $ir = 0.35$, a temporal deduplication between weekly backups of $tr = 0.95$, a weekly data growth rate of 1%, and a data compression ratio of 0.5 for one-year of weekly backups. In this setting, the file recipes need 3.2 TB storage and the post-deduplication data needs 32.9 TB storage. Here, the file recipes occupy 9.7% of the overall back-end storage.

More and more users of data deduplication backup systems use daily backups and/or longer retention periods. For a daily backups with a retention period of a year are used, a slightly higher temporal deduplication between the daily backups of $tr = 0.99$ and a daily growth rate of 0.2% is assumed. Then the file recipes have an estimated size of 26.1 TB and the post-deduplicated data has an estimated size 53.7 TB for a 20 TB backup. Therefore, the recipes occupy around 32.7% of the disk space the deduplicated data requires to be stored.

An approach to reduce the size of the recipes plays therefore an important role on the overall storage usage of a data deduplication system, especially if considering daily backups in a high deduplication ratio environment.

Figure 20 shows the estimated ratios of block and file recipes on the overall storage of the system for different backup-by-backup deduplication ratios (90%, 95%, and 99%) and number of backup runs. The growth rate is always set to a fifth of the backup-by-backup deduplication ratio.

Another factor that drives the recipe size up is that the currently commonly used hash function for data deduplication systems, SHA-1, will eventually be replaced by another hash function due to concerns about second pre-image resistance of SHA-1 [DCR08]. The possible alternative cryptographic hash functions, e.g., from the SHA-2 family (SHA-256, SHA-512), have larger digest length from up to 256 and 512 bits. Without recipe compression, this increases the size of recipes by a factor of 1.6 (SHA-256) or 3.2 (SHA-512). After a year of weekly full backups with an initial data size of 1 TB, a deduplication ratio of 95%, and a growth rate of 1%, the file recipes with SHA-512 hashes occupy 20.9% of the overall storage capacity.

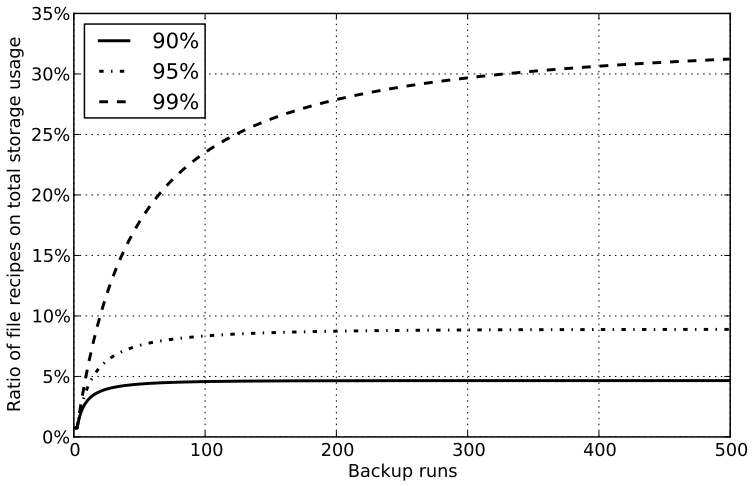


Figure 20: Estimated ratio of the block and file recipes on total storage usage in data deduplication. Backup-by-Backup deduplication ratios are 90%, 95%, and 99%. The growth rate is always set to a fifth of the backup-by-backup deduplication ratio.

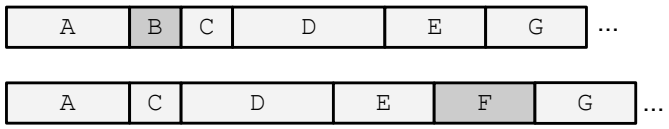


Figure 21: Illustration of ordering of chunk groups in two consecutive backup streams. Chunk group B is no longer part of the backup. Chunk group F is a group of chunks occurring in the later backup for the first time.

Another reason to consider at block and file recipes is that they inhibit similar properties as the data stream itself. Assuming a full backup using a block-based deduplication system, millions of blocks build the sequence of chunks for a backup run. Usually, the next backup run consists of data where only a small percentage of the data has changed. Again, the sequence of chunks forming the data of the backup run is stored in millions of consecutive blocks. Only a small percentage of these chunks changes due to modifications in the data stream. The order of the chunks, also, highly resembles the order within the previous chunk sequence. This is locality property is one of the central assumptions behind Zhu et al.'s work [ZLP08].

Figure 21 illustrates the locality property with an example. The two lines show the order of the chunks in a backup data stream in two consecutive backup runs. Besides that most of the chunks have already been stored, the order in which the chunks occur is similar.

Furthermore, another common pattern in backup data streams are shifts, e.g., due to newly inserted data. This is the reason why for backup workloads content-defined chunking is considered superior to static chunking [MB09]. These shift patterns are also reflected

in the recipes. In the *UPB* data set, less than 50% of the 256 KB blocks are identical between backup runs.

In this part, two novel techniques based on block and file recipes are introduced. First, a combination of compression approaches to reduce the size of the recipes and then a new caching approach that uses the relation between the backup data and the recipes to overcome the chunk lookup disk bottleneck.

One way to reduce the size of the file or block recipes is to use larger chunk sizes, e.g., 16 KB instead of 8 KB. This reduces the number of fingerprints stored in recipes and therefore the overall size of the recipes. However, previous studies have shown that increased chunk sizes decrease the deduplication results [WDQ+12, MB09, MB12].

Another way would be to use shorter cryptographic fingerprints. This is only possible if it is combined with a performance-intensive byte-wise comparison as otherwise the probability of data loss because of hash collisions increases.

Therefore, alternatives are favorable. The idea of block and file recipe compression is to assign (shorter) code words to fingerprints. The code word is then stored instead of the fingerprint in the recipe. Here, four compression approaches are proposed. Each uses unique properties of block and file recipes in data deduplication systems.

The first approach suppresses the fingerprint of the chunk filled with zeros (zero-chunk). Wei et al. handle the zero-chunk as a special case in the MAD2 system [WJZF10]. They may also already use "zero-chunk suppression".

The other three approaches are novel contributions: The second approach assigns each chunk fingerprint a shorter code word based on the number of stored fingerprints. It uses the chunk index to derive the code words, which form an alternative key for the chunk index entry.

Thus, the code word can be used as a replacement for the full fingerprints throughout the system without causing additional overhead. This approach is called "chunk index page-oriented compression".

A third approach called "statistical dictionary approach" uses the skew in the fingerprint usage. Certain fingerprints are more likely than others. A small set of common fingerprints get a short code word assigned using an explicit dictionary.

The fourth approach, called "statistical prediction approach", uses the observation that it is possible to predict the next fingerprint in a backup data stream if the previous fingerprint is known. If the next fingerprint can be correctly predicted using the previous fingerprint as context, a special code word is stored as a flag instead of the original fingerprint.

All compression approaches are practical, efficient, and scalable for the usage in data deduplication systems. They are evaluated using trace-based simulations using the *UPB*, *JGU*, and *ENG* data sets.

The second contribution, the novel approach to overcome the chunk lookup disk bottleneck, is called "Block Locality Caching" (BLC). Here, the relation between the backup data and the recipes are used to search an alignment between the recipes of the previous backup runs and incoming backup data. This alignment is then used to predict and cache blocks that contain chunks that are likely be accessed next. If an accessed chunk is found in the cache, it is not necessary to lookup the chunk in the on-disk chunk index.

Trace-based simulations show that this exact approach is able to find such an alignment in noisy real world data sets and save more than 99% of the chunk lookup related IO operations. The simulations show that the BLC approach needs less IO operations for a

backup data set than existing approaches for backup systems. Furthermore, the approach always uses the up-to-date locality information and is therefore less prone to aging.

The block and file recipe compression schemes have been published as "File Recipe Compression in Data Deduplication Systems" [MBS13]. The work has been supported by Tim Süß and André Brinkmann. The presentation here differs from the published version mainly in the usage of a new, larger JGU data set that also covers more weekly backups.

The unpublished research of the BLC has been supported by Jürgen Kaiser and André Brinkmann.

Chapter 9 describes related work with regards to compression in data deduplication systems, approaches to overcome to chunk lookup disk bottleneck, and the management of block and file recipes. In Chapter 10, the work on file recipe compression is described and evaluated. Afterwards, Chapter 11 describes the Block Locality Cache and its evaluation. Limitations on the compression approaches and the BLC approach are discussed in Chapter 12. Finally, the contributions presented in this part are summarized in Chapter 13.

9

RELATED WORK

The storage research community largely ignored the management and usage of block and file recipes. The reasons for this have been explained in the previous section.

This section presents specific related work with regards to the focus of this section: Compression within data deduplication systems and approaches to overcome the chunk lookup disk bottleneck.

9.1 COMPRESSION IN DATA DEDUPLICATION SYSTEMS

Data compression as LZ77 [ZL77] is used by most deduplication systems to reduce the size of chunks after the duplication identification step [ZLP08, MB10a, QD02, WDQ+12]. The most common approaches are the individual compression of each chunk [ZLP08, MB10a] or the compression of larger regions [QD02, WDQ+12]. In general, the compression ratio is better when larger regions are compressed together. Only a few deduplication systems do not compress chunk data [BELL09].

Multiple ways to reduce the size of the chunk index have been proposed. Sparse indexing reduces the size of the chunk index by only storing a subset of the chunks [LEB+09]. Another approach is to store only short fingerprints in an in-memory chunk index and to additionally compare the chunk data byte-by-byte to be safe from hash collisions [Alv11], which are likely in this setting.

Balachandran and Constantinescu propose to use runs of hashes for file recipe compression [BC08]. If a run of hashes occurs twice in a data stream, they replace it with the fingerprint of the first chunk and the length of the repeated sequence. Constantinescu et al. propose to combine repeated sequences of adjacent chunks to "super-chunks" [CPL09]. Any chunk is then either merged into a super chunk or it is not repeated. However, both works do not describe how to find the runs or super chunks in an efficient manner.

The "JumboStore" data deduplication system reduces the size of file recipes by chunking them into "indirection nodes" [ELW+07]. These nodes can be reused for identical and similar files. Based on the assumption of repeated runs of chunk hashes in the file recipes this indirection approach reduces the overall size of the file recipes.

The skew in the chunk usage distribution and the popularity of the zero-chunk has been noted before [WJZF10, KMBE12, MB09, HL04, JM09]. Wei et al. propose the special handling of the zero-chunk [WJZF10].

Patterson uses two separate index structures: One mapping from the fingerprint to a sequentially increased code word and one from the code word to the on-disk location [Pat06]. The code word length is in the order of the logarithm of the number of stored chunks. Therefore, it produces code words of similar length than the page-based approach.

The data model of the HYDRAstor content-addressable storage system consists a tree of immutable blocks, where each entry in a block consists either of data or a pointer to a different block [DGH+09]. The identifier of a block is a hash of the content and the pointers of the block. If a new block is written to the system that contains the same pointers as an existing block, the identifier of the existing block is used. This can be seen to applying data deduplication to these blocks. In the HydraFS file system [UAA+10], this data model is used to automatically remove identical segments of an "imap" entry, which is a data structure that is equivalent to a file recipe. However, the size of an imap segment is unclear as is the effectiveness of the approach in real data sets. The larger the imap segment, the larger the possible compression benefits, but also the smaller the likelihood to find identical segments.

Also, the page-based approach is similar to dictionary encoding in databases (see, e.g., Abadi et al. [AMF06]). However, database dictionary encoding is usually only applied to columns with a limited, fixed cardinality and it uses a separate lookup index. The page-based approach works for billions of chunks without additional index lookups.

The statistical dictionary compression is related to Huffman codes [Huf52] in that both aim to reduce the code word size based on the order-0 statistic of the data. The statistical dictionary approach differs in the way the code words are assigned. Furthermore, it relaxes the compression for the usage as file recipe compression.

The compression of index structures is state-of-the-art in areas like databases (e.g., [HRSD07, AMF06]) or information retrieval (e.g., [MRS09, WMB99]). The techniques used there, e.g., page-local compression, run-length encoding, and delta-compression within pages, are not directly applicable to compress block and file recipes.

9.2 OVERCOMING THE CHUNK LOOKUP DISK BOTTLENECK

One of the research challenges for data deduplication systems is to overcome the disk bottleneck of the chunk index lookups.

Zhu et al. presented a combination of three approaches, used in the EMC Data Domain File System (DDFS), to overcome the chunk lookup disk bottleneck [ZLP08]. A Bloom filter [Blo70] is maintained as a probabilistic summary of all fingerprints already stored in the system. Using the Bloom filter, the lookups for the majority of the chunks can be avoided if the chunk has not been stored yet. A Bloom filter might return a false positive answer and it can happen that a chunk that is classified as possibly redundant has not been stored before. Therefore, an additional chunk index lookup is necessary to verify the result for a positive answer.

A "stream-informed segment layout" stores the incoming new chunks in a container data structure so that in each container only the data from one stream is stored in arrival order. This captures the temporal locality of the data at the time the chunks are written for the first time. During backup runs, the fingerprints stored in the container are fetched into memory and stored in a cache. Assuming that the order of the chunk requests in the current backup run is similar to the order when the data has been written to the containers, it is very likely that many of the following chunks can be found in the container chunk cache after the container fingerprints have been loaded.

The "locality-preserving container caching scheme" works together with the stream-informed segment layout to capture the data order when the data is written for the first time and using this captured locality to accelerate the duplicate detection.

Similar ideas to accelerate the duplicate detection are used by Wei et al. [WJZF10] and Rhea et al. [RCP08].

Another approach to overcome the disk bottleneck is "Sparse Indexing", an approximate approach proposed by Lillibridge et al. [LEB+09]. Here, the chunks are sequentially grouped together into segments. A small set of chunks from each segment is chosen as "hooks", which are stored in a RAM-based chunk index. Lillibridge et al. propose choosing all hashes from a segment as hooks whose n least-significant bits are zero, leading to a "sampling rate" of $\frac{1}{2^n}$.

This index is called sparse because only the small set of hook chunks is stored. Therefore, the chunk index easily fits into main memory. For each incoming segments, similar existing segments are located by searching segments that are most similar to the incoming segment. These similar segments are called "champions". The champion segments are then used to determine for the incoming segment if the chunks are already stored.

New chunks that have not been stored before are stored using a container data structure, as it is common in data deduplication systems. Afterwards, the segment is stored and new hooks are inserted into the sparse index.

"Extreme Binning" is related to Sparse Indexing as both use near-duplicate detection to overcome the disk bottleneck [BELL09]. It is a cluster deduplication system targeted at backup systems with low temporal locality. This means that the backup workload consists of individual files, which arrive in random order.

Extreme Binning detects duplicates by calculating a fingerprint over the full file content and then selecting a single chunk fingerprint as representative fingerprint from all chunks of the file. The representative fingerprints of all files are maintained in an in-memory index, called "primary index".

If the representative fingerprint of a file has not been stored before, all unique chunk fingerprints of the file are added to a data structure called "bin". The deduplication systems stores the bin to disk and the adds the representative fingerprint to the primary index together with a pointer to the bin and the full file fingerprint.

If the representative fingerprint of a file has been stored in the primary index, the full file fingerprint of the current file is compared with a full file fingerprint previously used together with the representative fingerprint. If the full file fingerprint does not match, the bin is loaded from disk. Then, for all remaining chunks of the file, the deduplication systems checks if the chunk is already stored in the bin. If a chunk is not stored in the bin, the chunk is added to it. After the checks are completed, the updated bin is stored to disk.

Guo and Efstathopoulos propose another approximate approach to overcome the chunk lookup disk bottleneck [GE11]. They create a sparse index keeping on average every 1 in T chunks by using the fingerprint modulo $T = 0$ as anchor. This sparse index is then used to fetch the container into a cache. The approach combines a sparse index with a locality-preserving container-caching scheme.

Similar to Block Locality Caching, "CABdedupe" aims to use the backup data of the previous backup run to guide the deduplication [TJF+11]. They use explicit causality information like information which files have been modified, renamed, or removed. If the file content has changed, the new file recipe is compared to the file recipe of the same file

in the previous backup run to detect identical chunks. However, such explicit causality information is often not available in backup storage systems.

The "SiLo" approach uses similarity and locality to overcome the bottleneck [XJFH11]. The data stream is segmented by grouping multiple small files into one segment and by splitting a large file into multiple segments. These segments are then used to detect similar segments using a representative fingerprint as in Extreme Binning. In addition, multiple segments are grouped into a larger unit called "blocks" (segment size ~2 MB, block size ~100 segments). If a segment is determined to be similar, the fingerprints of the complete block are fetched into a cache.

The approaches "iDedup" [SBGV12], "HANDS" [WMR12] and "I/O Deduplication system" [KR10] use different workload properties to enable the inline deduplication of primary storage data. The "DBLK" system uses a multi-level in-memory Bloom filter to implement an inline deduplication system for primary block storage [TW11].

Other approaches use faster hardware to overcome the disk bottleneck, especially Solid State Disks (SSD) [MB10a, DSL10, LND12].

9.3 OTHER USAGES OF BLOCK AND FILE RECIPES

Kaczmarczyk et al. researched how data fragmentation causes a reduced restore performance and propose rewriting some disk content based on the backup stream context to reduce the fragmentation [KBKD12]. While the term "recipe" is not used in the publication, the "backup stream context", as they call it, can be determined by the block and file recipes.

Another usage of file recipes has been presented by Lillibridge et al. [LEB13] to improve the restore speed of deduplication systems. On the one hand, they limit the number of old containers a recipe can refer to. This approach called container capping trades some deduplication against a higher restore performance. On the other hand, they use the following file recipes of the backup data that should be restored to build a forward-looking cache of container data, which is called "forward assembly area".

10

INDEX COMPRESSION

The idea of block and file recipe compression is to assign smaller code words to fingerprints. The code word is then stored instead of the fingerprint in the recipe. Different approaches to select the code words are explored in this chapter.

Also, it is often necessary to be able to resolve a code word back to the matching full fingerprint, e.g., during a read operation if the code word alone does not identify the container and the chunk within a container. Another operation for which resolving a code word is necessary is the update of the usage count if the system uses a reference-counting-based garbage collection.

The three different backup data sets, which have been described in Section 2.4, are used in this section to motivate the approaches based on observations from the data sets. The same data sets will later be used for the evaluation.

The compression approaches are described in the next sections. The evaluation follows in Section 10.4.

10.1 ZERO-CHUNK SUPPRESSION

The first approach is called "zero-chunk suppression". Several authors have noted that a few chunks are responsible for a high number of duplicates [WJZF10, KMBE12, MB09, HL04, JM09]. Especially noteworthy is the chunk completely filled with zeros ("zero-chunk"). Jin et al. show that zero-chunks are common in VM disk images [JM09].

A special handling of the zero-chunk has been proposed by Wei et al. [WJZF10]. While it is not clear if the zero-chunk is actually replaced by a short fingerprint in Wei et al.'s MAD2 system, it is assumed here.

A 1-byte special fingerprint replaces the zero-chunk in the dedupv1 system. This replacement was never mentioned in a publication as it was solely done to make the read path easier. The compression savings of this have been accidental. In di-dedupv1, the zero-chunk receives also a special handling to avoid a remote chunk lookup.

If the special code word assigned is small, e.g., 1 byte, the size of the recipe is reduced. Alternatively, it is possible to store a bit mask in the recipe similar to the way missing values are handled in some databases [SHWK76].

This technique is not only free in terms of IO, CPU, and storage capacity, but also allows saving IO operations.

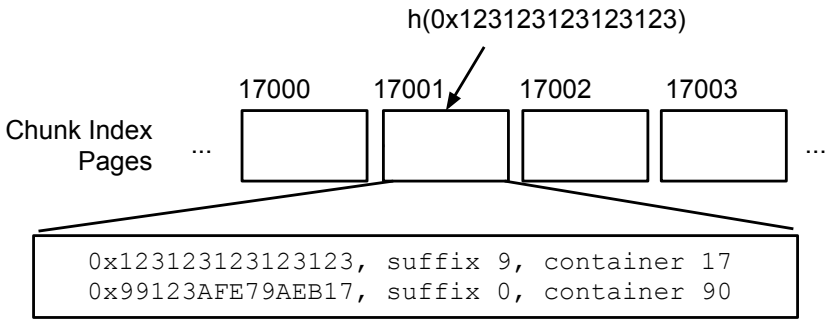


Figure 22: Illustration of the page-oriented approach. A fingerprint is hashed to an index page. A page-unique code word prefix is assigned to each fingerprint. The complete code word is `17001||9`.

10.2 CHUNK INDEX PAGE-ORIENTED APPROACH

This approach aims to assign a code word to all chunks that is not significantly longer than necessary to have a unique code word for each chunk in the system. The approach is called "chunk index page-oriented" because it uses chunk index pages to assign code words.

The approach assumes that the chunk index is implemented using a paged disk-based hash table and fingerprints are hashed to on-disk pages.

The code word consists of two parts: prefix and suffix. The suffix is chosen using the least number of bits possible, which are still unique within the page. The code word is aligned to byte boundaries allowing a faster processing. The prefix of the code word denotes the page number. The combination of suffix and prefix together uniquely identify the fingerprint (see Figure 22).

The code word is an alternative key for the full fingerprint in the chunk index. The prefix is used to identify the index on-disk page where the fingerprint is stored on. Given the page contents, the fingerprint and its chunk index entry data can be found by searching in the page for the fingerprint with the matching suffix. Therefore, it is possible to lookup the chunk index entry corresponding to the fingerprint without an additional index lookup.

Alternatively, the suffix can be used as an index within the page if the chunk index entries are of a fixed size.

This approach assigns code words when the fingerprint is stored in the system for the first time. The complete fingerprint/code pair is also stored in the container to allow reading an item from a container using only the code word. Depending on the crash recovery scheme this might be required. For example in the dedupv1 system not-persisted parts of the chunk index are restored from the container data after a crash. It is therefore possible to ensure that a fingerprint gets exactly the same code word assigned as before the crash.

This chunk index page-oriented approach uses code words that are as long or slightly longer than necessary to assign a unique number to all chunk fingerprints. The two-index approach presented by Patterson uses fingerprints of a similar length [Pato6].

Table 3: Order-0 statistic for the *ENG* data set after 3 backups.

Fingerprint	Usage count	Probability	Entropy
`0x518843...`	380,508	0.108765	3.2
`0x04F90E...`	7,227	0.002066	8.9
`0x4a9b36...`	4800	0.001372	9.5
...	...	...	...
`0x435123...`	1000	0.001	10.0
`0xF7290B...`	128	0.000055	14.2
`0x5BC1B1...`	27	0.000008	17.0
...	...	...	...
`0x6B4D0A...`	3	≈ 0	20.1

10.3 STATISTICAL APPROACHES

The zero-chunk suppression assigns short fingerprints to a certain, well-known fingerprint, while the chunk index page-oriented approach provides a basic code word for all fingerprints in the system.

The statistical approaches presented in this section generalize the zero-chunk suppression. They use statistics to assign shorter code words to fingerprints based on the estimated probabilities of the chunk usage.

The first approach relies on the observation that the number of references to chunks is highly skewed [MB09]. The assigned code words are stored in an additional in-memory dictionary to allow a lookup from the code word to the fingerprint.

The approach uses the fingerprint's "order-0" statistic. This statistical model relies on a fingerprint's usage without looking at its context, e.g., at previous fingerprints in the data stream. A second statistical approach using the "order-1" statistic, which conditions the probabilities of a fingerprint on the previous fingerprint. These are standard statistical models used for data compression [Say00, Nel95].

10.3.1 *Statistical Dictionary Approach*

The order-0 statistic is the statistical model relying on a fingerprint's usage without looking at its context, e.g., at previous fingerprints in the data stream. Information theory provides a lower bound on the code word length that would be achievable with this statistical model.

The information content is the sum of the fingerprint entropies. The entropy of a fingerprint h is $-\log_2\left(\frac{\text{usage count of } h}{\text{total chunk usage}}\right)$. The higher the probability of a fingerprint, the lower is the entropy and the shorter the optimal code word length.

This skew can be observed in all three data sets introduced before. Table 3 shows examples of the entropy statistics of the *ENG* data set. A fingerprint that is used 7,227 times has an information entropy of 8.9 bits given the overall state of the recipes at that time. The average entropy (and therefore optimal average code word length) is 17.3 bits per chunk using this statistical model and an optimal compression scheme.

The approach is similar to Huffman compression codes [Huf52]. However, the compression of recipes has properties that render it impossible to use Huffman codes and other classic compression approaches directly:

SIZE OF ALPHABET: The size of the code alphabet is 2^{160} instead of 2^8. All schemes relying on holding complete statistics about the full context of a fingerprint are impractical without intensive pruning. It is not even feasible to hold an index of all stored file fingerprints in memory. Therefore, it is also not feasible to create a full Huffman tree in memory.

RANDOM ACCESS: Data deduplication storage systems have to support random accesses. Adaptive text compression decoders rely on parsing the full history of a compressed data stream to build up the same model as seen by the encoder. This is, e.g., the reason `.tar.gz` files do not provide random access. However, in storage systems, it is mandatory to provide random access to files and disk blocks.

PAYLOAD PATTERNS CHANGES: A heavily pattern or fingerprint is not necessarily a heavily used pattern years later. It is necessary for a practical recipe compression scheme to adapt to these changes.

Since it is not practically possible to build a full Huffman tree, a relaxed approach to use the skew of the chunks for recipe compression was chosen. In this called "statistical dictionary approach", fixed size code word is assigned to the fingerprint if the entropy of a chunk is below a certain threshold. The check can either be performed in a background process or alternatively when a write command accesses a chunk.

However, the entropy is not known in advance and can only be estimated after some data has been written. Therefore, any code word assignment is delayed until the chunk index is filled beyond a minimal threshold or until some backup runs have been stored. This is done to avoid preemptive assignments before reaching a stable estimate of the probabilities. The intuition is that the probabilities based on the usage counts are stable in a backup environment after the first backups of a backup stream have been written. This is only a rough estimation as, e.g., long-term shifts in the data or new backup streams may invalidate this assumption.

Another important consequence of the delayed code word assignment is that either full fingerprints are stored in some recipes or it would be necessary to rewrite recipes.

If idle time is available, the recipes can be re-written in the background. If no idle time is available or rewriting the recipes is not feasible, compressibility for these already written recipes is lost. The compression then would only be used for newly written recipes.

Once a code word is assigned, it can never be revoked as long as a recipe uses that code word. If at some point a chunk usage falls below a threshold, the code word is declared as inactive and all new appearances again use the full fingerprint. Older backup runs are eventually evicted from the backup storage system. If there is no longer a reference to the code word stored in any recipe, the code word is garbage collected and the code word becomes free again, e.g., it is possible to maintain a separate code word usage counter in addition to the chunk usage counter in the chunk index.

For most of the chunks, the entropy is nearly identical to the length of the code word assigned by the page-oriented method. It is, therefore, not worthwhile to assign an entropy-based code word to more than a small fraction of the chunks.

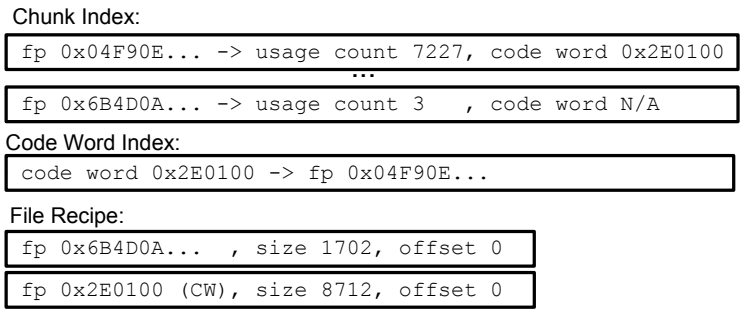


Figure 23: Illustration of the statistical dictionary approach. The chunk with the fingerprint `0x04F90E...` has, based on its high usage count, a short code word. The code word is then used in the recipe instead of the fingerprint. The recipe shrinks from 40 bytes to 23 bytes in this example.

One important assumption is that it is effectively possible to estimate the overall probability of a chunk. The reference count is often collected for the garbage collection e.g., HYDRAstor [SAHI+13] and dedupv1 maintain reference counts. By putting this usage count in relation to the total number of references, the probability of a chunk can be easily calculated.

In deduplication systems without reference-counting based garbage collection, e.g. using a mark-and-sweep garbage collection [McC60, EG10, GE11, BSGH13], an approximation of the reference counts needs to be estimated. The estimation could use algorithms as approximate counting [Mor78] or frequent item set estimation [MG82]. Nevertheless, extra effort is needed to estimate chunk probabilities if no reference count is maintained.

In contrast to the previous approaches, a separate index is needed to store the mapping from the code word to the fingerprint. This opens the question where to store the reverse code word index: in RAM or on a persistent storage. Holding the index in memory allows fast access, but limits the number of code words. Using an on-disk index, the number of code words is virtually unlimited, but this requires additional IO operations to resolve full fingerprints.

10.3.2 *Statistical Prediction Approach*

The statistical dictionary approach uses the probability that a chunk fingerprint is referenced without using context information (order-0 statistic) to build a code word.

The order-1 statistic looks at the previous chunk and calculates the probability of the fingerprint based on that context information.

The entropy of the order-1 model again provides a theoretical lower bound. In the *ENG* data set, the entropy using the order-1 statistical model is 0.13 bits. The statistics in the other data sets are similar (0.14 bits in *UPB*, 0.21 bits in *JGU*).

This shows that there is only little uncertainty about the next fingerprint if the previous fingerprint is known. Capturing this low uncertainty in a practical scheme, called "statistical prediction approach" promises significant compression.

Algorithm 1 Misra-Gries: Update fingerprint f:

```
1: if H contains f then
2:   c_f += 1
3: else
4:   Add pair (f, c_f) with c_f = 1
5: end if
6: if |H| > k then
7:   for all pairs (f', c_f') do
8:     c_f' -= 1
9:     if c_f' = 0 then
10:      Remove (f', c_f')
11:    end if
12:  end for
13: end if
```

The low entropy using this statistical model in backup data streams has two main reasons: The redundant data found by the deduplication process is usually clustered together and forms larger sequences of redundant and unique chunks, called "runs".

In the *UPB* data set's last backup the average run length of redundant chunks without a single unknown chunk breaking the series is 449.8. The average run length in the *JGU* data set is shorter (262.9), and it is larger in the *ENG* data set (901.5).

In all cases, there is a significant clustering, which forms long runs of redundant chunks. The backup runs change only slightly from one backup run to another and usually the backup is written sequentially. This is one of the assumptions that empowers several techniques for backup workloads, e.g., the locality preserving caching approach of Zhu et al. [ZLP08].

The information to store the order-1 model grows quadratic in the number of chunks and is therefore not usable in practical deduplication systems. The approach presented here relaxes the order-1 model and reduces the information kept per chunk to a constant size. Considering the low entropy of the order-1 model in the backup data streams, even a relaxed approach should be able to predict fingerprints with high accuracy.

The data stream algorithm "Misra-Gries" is used to approximate the detection of frequent fingerprint pairs [MG82]. The algorithm maintains a collection H of pairs (f, c_f) and is configured with a parameter k. The parameter determines the number of pairs stored in the collection.

Algorithm 1 shows how the data structures are updated when a fingerprint f is used. The fingerprint f with the highest c_f is estimated to be the most frequent fingerprint in the data structure.

Thus, up to k fingerprint pairs are stored in the chunk index entry. It is also possible to store the page-based code words instead of fingerprints in the chunk index entry.

The parameter k denoting the number of possible frequent fingerprints is used to trade accuracy against memory overhead. Different choices for k are later compared with the full order-1 statistics.

When a recipe is updated, the frequent item data structure of the previous fingerprint is updated with this fingerprint in the recipe. Based on this, the data structure can be queried to receive the estimation for the fingerprint that most frequently followed the

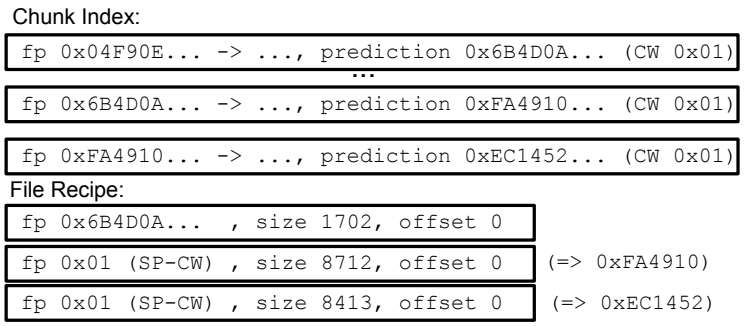


Figure 24: Illustration of the statistical prediction approach. If the fingerprint 0x6B4D0A... occurs after 0x04F90E..., the fingerprint is replaced by the code word 0x01. One full fingerprint is used as an anchor for the compression. The code word is then used in the recipe instead of the fingerprint. The recipe shrinks from 60 bytes to 22 bytes in this example.

previous fingerprint. This fingerprint as the prediction candidate is selected and a short 1-byte code word is assigned to it. Similar to the statistical dictionary approach, a code word is not changed as long as a recipe uses it. Instead, if eventually a different prediction candidate is chosen, another free code word is used.

When data is written to the system, the prediction candidate based on the previous chunk is checked. If a fingerprint matches the prediction candidate based on the previous fingerprint, the code word is stored in the recipe. Figure 24 shows an example for this approach.

Every one in b fingerprints in a recipe is stored as full fingerprint (or a different kind of code word) instead of using the code word determined by the statistical prediction approach. This provides an anchor so that random access is possible. For example, the dedupv1 deduplication system uses a block size of 256 KB, so on average every one in 32 chunks a full fingerprint is stored.

10.4 EVALUATION

The compression approaches are evaluated using the following criteria:

STORAGE: Additional information that is stored per chunk in the chunk index or other indexes.

MEMORY: Amount of main memory per chunk used for compression. The memory cost does not include temporarily used memory, e.g., for paging on-disk data.

ASSIGNMENT: Time when code word is assigned.

BACKGROUND PROCESSING: Is an additional background processing, e.g., in idle times, necessary.

RECIPE COMPRESSION: The compression ratio of the recipes, evaluated based on the trace data sets presented before. The setting assumed for this evaluation is a backup-deduplication environment with weekly full backups. The size of the original block

Table 4: Overview of recipe compression's properties and results. Numeric values in bytes per chunk.

	Storage	Memory	Assignm.	Backgr.
Zero-Chunks (ZC)	None	None	Direct	No
Page-based (PB)	≈ 1	None	Direct	No
Directory (SD)	≈ 0.24	≈ 0.24	Delayed	Yes
Prediction (SP)	$\approx$ 20-68	None	Delayed	No

and file recipes, assuming 20 bytes per chunk reference, is compared to the size after compression.

The source code of the simulation has been released under an Open Source license to enable an independent review and to foster further research. The source code is available at `https://github.com/dmeister/fs-c-analytics`.

The compression methods are first evaluated individually. Afterwards, the compression methods are applied together and present the combined compression ratios. Table 4 summarizes the properties 1-4.

The compression ratios have been evaluated using a trace-based simulation based on the FS-C tool suite, which has been described in Section 2.4. The compression ratios are shown in Table 5.

10.4.1 *Zero-Chunk Suppression*

The zero-chunk suppression (ZC) method does not cause any additional IO operations or increases the storage requirements. In contrast, it saves a significant amount of chunk index lookup operations if the backup stream contains a high amount of zero-chunks and has therefore a positive performance impact.

The compression results depend on the data set: In the *ENG* data set, 14.9% of the chunks reference the zero-chunk. The zero-chunk suppression reduces the recipes by 11.8%. On the other hand, the *UPB* data set contains only 0.6% zero-chunks and the *JGU* data set contains only 0.3%. Here, the recipes shrink by 0.5%, resp. 0.26%.

10.4.2 *Chunk Index Page-based Approach*

The code word length using the chunk index page-oriented approach (PB) depends on the scale of the system. In an 8 TB configuration with 2^{24} chunk index pages and a page size of 4 KB, each chunk index page will hold 64 chunks on average. The balls-into-bins theory [RS98] shows that with high probability no page has to store more than 97 chunks. Therefore, a suffix length of 7 bits is sufficient to enumerate all chunks in a page and to assign a unique suffix to each of them. The prefix would account for 24 bits, so that the final code word has 31 bits or 4 bytes when aligned to full bytes. This 4-bytes code word would not be sufficient for a large configuration with, e.g., 256 TB back-end storage, where 2^{29} chunk index pages are needed. Here, 5 bytes would be sufficient.

The extra storage costs consist of the prefix that is stored in each chunk entry. With 4 bytes code words, this method reduces the recipes by 80% (75% with 5 bytes). This

method's advantage is that the savings are predictable and do not depend on the concrete data set.

This approach has no impact on the system throughput if the deduplication requests each chunk in the chunk index as dedupv1 does. The approach is also easy to modify for environments where the chunk index is not requested at all, e.g. for Zhu et al.'s caching approach by storing the code word in the container.

When combined with other methods, the page-oriented approach provides a base compression for all fingerprints, while certain other approaches use even smaller code words.

10.4.3 *Statistical Dictionary Approach*

Similar to the zero-chunk approach, the savings of the statistical dictionary approach (SD) depend on the concrete data set. The higher the skew of the chunk usages, the better the dictionary compression performs.

This approach is simulated by assigning a 24-bit code word to each chunk with an entropy smaller than 85% of $\log_2(n)$ where n denotes the number of chunks. Once a code word is assigned in the simulation, it is never deactivated or revoked.

Using this method, 0.36% of the chunks in the *ENG* data set have an assigned code word after 20 backup runs. The size of the recipes is reduced by 14.2%. In the *UPB* data set, 0.25% of the chunks have a code word assigned by this method resulting in a compression ratio of 9.0%. In the *JGU*, 0.07% of the chunks have a code word, which compresses the recipes by 3.6%.

An additional index needs to be stored to resolve the code word back to its fingerprint. The index needs to store the code word and the corresponding 20-bytes fingerprint. Assuming that 0.4% ($\approx 2^{-8}$) of the fingerprints are shortened, the reverse index can be stored in main memory. For an 8 TB configuration with around 2^{30} chunks, 4 million chunks get a code word assigned. Therefore, the code word index has a size of approximately 128 MB if full fingerprints are used or 32 MB if this approach is combined with the page-based approach.

10.4.4 *Statistical Prediction Approach*

The statistical prediction approach (SP) is simulated by assigning predictions after each backup run. Once a prediction is assigned, it is not changed. The compression ratios with $k = 2$ fields for the Misra-Gries algorithms are promising. The approach alone leads to a compression between 69.2% (*JGU*) and 82.2% (*ENG*).

Increasing the Misra-Gries parameter k results in a better estimation of the most frequent following fingerprint and therefore in better predictions for the compression. With $k = 4$, the compression improves slightly: to 82.5% (*ENG*), 77.6% (*UPB*), and 69.3% (*JGU*). Larger values for k do not increase the compression substantially. Even if the full order-1 statistic is maintained (instead of using the Misra-Gries estimation) the compression only increases to 82.6%, 77.6%, and 69.5%.

The statistical prediction approach stores additional data in each chunk index entry: The k fingerprints/counter pairs for the Misra-Gries algorithm and a selected prediction fingerprint. Therefore, around 68 bytes need to be added for $k = 2$. Depending on the data

Table 5: Compression ratios (for all combinations).

ZC	PB	SD	SP	UPB	JGU	ENG
Yes				5.4%	0.2%	12.6%
	Yes			80.0%	80.0%	80.0%
		Yes		9.0%	4.6%	14.2%
			Yes	77.3%	78.3%	82.2%
Yes	Yes			80.1%	80.0%	80.2%
Yes		Yes		9.1%	4.6%	15.9%
Yes			Yes	77.3%	78.4%	84.1%
	Yes	Yes		80.5%	80.3%	80.8%
	Yes		Yes	92.2%	91.9%	93.0%
		Yes	Yes	79.0%	78.7%	83.7%
Yes		Yes	Yes	79,1%	69.5%	84.3%
Yes	Yes		Yes	92.2%	91.9%	93.3%
Yes	Yes	Yes		80.6%	80.3%	82.2%
	Yes	Yes	Yes	92.2%	92.0%	93.1%
Yes	Yes	Yes	Yes	92.3%	92.0%	93.3%

stored in the chunk index, this can lead to a chunk index that is up to a factor of 4 larger. This overhead can be reduced to $\approx$ 20 bytes if combined with the page-oriented approach.

The statistical approaches can be sensitive to long-term aging effects. Such effects usually arise slowly from changes in the backup data pattern, e.g., a fingerprint is no longer used in new backups and its entropy is increasing so that is would not get a code word assigned.

For the vast majority of the fingerprints, the dictionary and the prediction remain constant for a long time. In the *UPB* data set, covering 15 weeks, on average 0.7% of the dictionary code words are deactivated per backup run, assuming that a code word is deactivated immediately after it crosses the entropy threshold. If the prediction is changed as soon as a different fingerprint is estimated as more common successor, on average 0.04% of the chunks change their prediction per week.

10.4.5 *Combination of Compression Approaches*

The approaches can be combined: A fingerprint is always replaced by its shortest code word and, thus, can achieve the highest compression. All possible combinations of approaches are reported in Table 5. If all approaches are applied, the recipes of all data sets are compressed by more than 92%.

Most savings are created by the statistical prediction and the page-oriented approach. These two combined achieve a compression that is within 0.3% of the compression if all four approaches are combined. The combination is also advantageous because it reduces the size of the prediction information that need to be stored in each chunk index entry. Around 12 bytes for need to be stored in the chunk index entry for a Misra-Gries data

structure with $k = 2$ and a prediction fingerprint. With full fingerprint, at least 60 bytes are added to each chunk index entry.

Still, the size of each chunk index entry increases. This, however, does not change the estimation of the size of the page-based code words. Assuming that the size of each chunk index entry doubles, each page stores less chunks, e.g., 32 chunks on average. With a high probability not more than 64 chunks need to be stored in any chunk index, leading to a suffix length of 6 bits. As the number of chunks in the system is not changing, 2^{25} chunk index pages are now necessary. In total, the code word can still be estimated with 31 bits.

The already established approach of zero-chunk suppression has only a small effect if it is used in combination with other compression methods. Similarly, the method with significant main memory overhead, the statistical dictionary approach, does not provide a substantial additional compression if it is used in combination.

None of the approaches causes any additional IO operations or requires extensive locking in the critical path. The computations, e.g. comparing the fingerprints with the fingerprint of the zero-chunk, are fast and parallelizable. It is unlikely that the computations are a bottleneck for the system throughput.

All approaches except the zero-chunk suppression have to store additional information, e.g., in the chunk index. The compression savings amortize the overhead, which grows with the physical storage capacity.

The conclusion of these combinations is that a practical system should skip the order-0 compression because of the maintenance of an additional in-memory index. While in the combination with other approaches, the zero-suppression is not necessary for a high compression, a practical system should use it because it benefits system throughput, too. Therefore, the zero-chunk suppression, the chunk-index based approach as well as the statistical prediction approach are implemented as prototype in the dedupv1 data deduplication system.

11

BLOCK LOCALITY CACHING

The similarities between the block recipes and the backup data stream are used to devise a novel approach to overcome the chunk lookup disk bottleneck. The "Block Locality Cache" (BLC) uses a heuristic to find an alignment between the previous backup and the current position in the current backup run. Under the locality assumption the chunks with the current data stream are likely to be found at (or near) the aligned position within the latest backup run. An estimation for such a matching alignment is maintained over the course of the backup run.

Figure 25 illustrates the idea. At the start of the second backup run, the block locality finds the beginning of the previous backup run. The vertical line indicates the alignment with regards to the block of the current write and the beginning of the previous backup run. As data from chunk group A is written, the block recipes of the old backup run with the matching alignment are loaded and cached. Ideally, all known chunks can be found in the cache without asking the chunk index. When the backup client writes the chunk group C, the cache fails to correctly predict the chunks.

According to the estimated alignment the Block Locality Cache loads parts of chunk group B. Therefore, the approach fails to predict the chunks and the chunk index is asked. Conceptually, the BLC learns the new correct alignment – denoted by line 2). When the data stream reaches the group F, the cache is not able to correctly predict the chunks because the chunks in the group F are unknown. This is detected by a chunk index lookup or a lookup in a Bloom filter [Blo70]. Finally, also the first chunks of the group G are not predicted, but, again, the alignment between the current backup stream and the last backup stream is learned.

With such an estimated alignment, the Block Locality Cache always caches the block recipes at the aligned position. When a chunk is deduplicated, the cache is checked first.

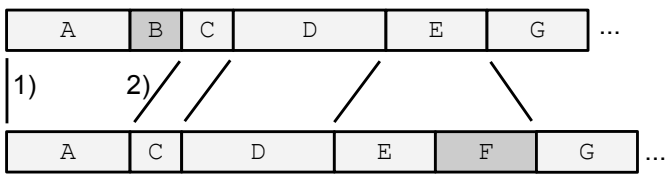


Figure 25: Illustration of ordering of chunk groups in two consecutive backup streams. The vertical lines illustrate a good estimate for the offset between the backup streams.

If the chunk fingerprint has been stored before at or near the alignment, the chunk can be found in the cache. Then, the chunk is declared as known and the chunk data is not stored. The expectation is that the costs of loading the block recipe are amortized by finding multiple chunks in the cache. Therefore, multiple chunk index lookup operations are avoided.

The BLC approach predicts future chunk requests better than existing approaches resulting in significantly less remaining IO operations for a backup run. Furthermore, the approach is designed to always use up-to-date locality information, which makes it less likely to a degenerated efficiency on aged systems.

The basic idea given in this section is extended to a fully function design in Section 11.1. Especially, the concept to detect and maintain an estimation for a suitable alignment in an efficient, practical way is explained. Afterwards, the approach is evaluated in relation to different existing approaches in a trace-based simulation. Finally, the throughput is evaluated using a prototype implementation based on the dedupv1 system. The trace-based simulation and the performance evaluation based on the prototype can be found in Section 11.2 and Section 11.3.

11.1 DESIGN

The Block Locality Cache approach uses a heuristic to align the data stream of the latest backup run with the current backup data stream. If such an alignment is available, it is easy to fetch the next block recipe in the latest backup stream and to compare incoming fingerprints with the fingerprint in the cache. If the backup streams are aligned and the locality assumption holds, the fetched recipes are good predictions for the next incoming fingerprints.

The alignment is found by estimating the difference between the block locations in the virtual device between the current backup stream and the latest backup stream with related data. The difference estimates are maintained over the complete backup and automatically updated to reflect the changes in the data pattern to keep the alignment up-to-date.

If an alignment is found on the first data access, which is then used for the remaining backup run, a single difference value would be sufficient. However, due to changes in the backup data, a single alignment is not able to predict future chunk requests well enough. The simulation will show that a single alignment is not even sufficient if it is allowed that the alignment changes during the backup run. Therefore, a small set of difference values is maintained in a "difference cache". The cache therefore contains a small set of possible data stream alignments.

The difference values are obtained heuristically. The chunk index entry of a fingerprint is extended so that it also stores a new value called the "block hint". The block hint is the block identifier of the last block that has accessed the chunk. The block hint is later used as an anchor that helps finding the correct offset of the new backup data in the most recent previous backup. A simplified example for the usage of the block hint field is shown in Figure 26.

When the system checks if a chunk fingerprint f in a current block b is already known, it first checks the cache containing the fingerprints of the prefetched blocks. This "block chunk cache" is a small cache that is maintained using a least-recently used (LRU) strategy

Block Index

```
block 104 -> [0x32342344, 0xFE744193]
block 105 -> [0xAA13B112, 0xBC3713A1]
```

Chunk Index

```
0x32342344 | block hint 104 | container id 4
0xAA13B112 | block hint 105 | container id 4
0xBC3713A1 | block hint 105 | container id 5
0xFE744193 | block hint 104 | container id 37
```

Figure 26: Simplified example of chunk index and block recipes.

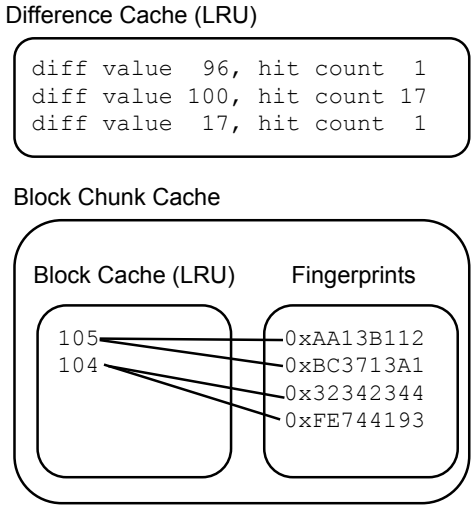


Figure 27: Simplified example of the cache data structures of the BLC approach.

of the cached blocks. If the block cache contains the fingerprint f ("block cache hit"), the system can classify the fingerprint as redundant and avoids the costly chunk index lookup for the fingerprint. Furthermore, the difference between the cached fingerprint's block hint value and the current block is calculated and the difference cache is updated.

If the fingerprint f cannot be found in the block chunk cache, it is a "block cache miss". In this case, the values in a difference cache are processed in the order of the most recent success. One of the fundamental idea of the BLC is that often the last diff value that was able to predict a chunk successfully is also able to predict the next chunk. The last diff value that succeeded is called the "leading" diff value and it is tried first.

Based on the difference value d and the current block b, the predicted block $b - d$ is calculated and the block recipe is loaded into the cache. After fetching the block fingerprints into the cache, the block chunk cache is checked again. If the difference value is a good prediction for the alignment between the backup runs, the fingerprint is now expected to be available in the cache. If the fingerprint is still not found, the next difference in the cache's sorted order value is tried.

When all difference values have been processed and the fingerprint has not been found, the chunk index is requested. If the chunk is found there, the last block hint value from the chunk entry is used to update the difference value with the difference between the current block and the hint value. In all cases, the block hint values of all chunk entries are updated with the current block id. The pseudo code of the fingerprint check using the block locality caching is shown in Algorithm 2.

For example, the current block id is 200 and this block contains chunks with the fingerprints (`0x32342344`, `0xFE744193`). The difference value cache contains 100 and the block and chunk index have the state shown in Figure 26. Assume that the fingerprint `0x32342344` is not found in the chunk cache. Then, the block $200 - 100 = 100$ is fetched from disk based on the cached diff value. The newly fetched block is inserted into the block chunk cache. The expectation is that in most cases the chunk is either found in the cache in the beginning or it is at least found in the cache after the fetch operation.

Here it is assumed that the block 100 does not contain the requested fingerprint, which leads to a misprediction. Then, the chunk index is requested to check if the fingerprint `0x032342344` is known. The fingerprint exists and the block 104 has most recently used it. This indicates a hole in the data stream, e.g., because a file in the file system has been deleted. The diff value $200 - 104 = 96$ is then added to the cache and 96 is a new estimate for the difference between the current position in the backup stream and a position with previously stored, related data.

Next, the fingerprint `0xFE744193` is processed. There, the block 104 is fetched from disk based on the most recently added or successfully used value in the diff cache. The block 104 stored the fingerprint `0x32342344` and under the locality assumption it is also likely to contain the fingerprint `0xFE744193`. The following block 201 contains the fingerprints `0xAA13B112` and `0xBC3713A1`. Based on the most recently used diff value 96, the block 105 is (correctly) predicted to likely contain these fingerprints. Figure 27 shows the state of the BLC data structures after these blocks have been processed.

A larger block chunk cache helps to reduce chunk index lookups. However, there is a trade-off with the main memory consumption that limits the size of the block cache. Also, after growing beyond some cache size, the additional gain by a larger cache vanishes.

Algorithm 2 Check fingerprint f in block b:

```
1: if block chunk cache contains f then
2:   Move all cached blocks containing f to end of LRU list
3:   Move all matching cached difference values to end of LRU list
4:   return f is redundant
5: else
6:   for all diff values d in diff value cache (ordered by recent usage) do
7:     p ← b − d
8:     Load block p into block chunk cache
9:     {Evict least recently used block with fingerprint f′ from cache}
10:    {Remove fingerprint f′ from the cache if no cached block contains f′}
11:    if block chunk cache contains f then
12:      Add d to end of difference cache LRU list
13:      return f is redundant
14:    end if
15:  end for
16:  if chunk index contains f then
17:    h = last block that used f (from chunk index entry)
18:    Load block h into block chunk cache
19:    d ← b − h
20:    Add d to end of difference cache LRU list
21:    return f is redundant
22:  else
23:    return f is new
24:  end if
25: end if
```

A larger difference cache is a trade-off between the quality of the prediction and the costs of block recipe fetches if the prediction is wrong. In the worst case, a single missed block cache check results in one IO operation per value in the difference value cache.

The prediction-based BLC is not able avoid the chunk index lookups for chunks that have not been stored before. Therefore, the BLC needs to be combined in exact data deduplication systems with another approach targeting new chunks. One approach is to use a Bloom filter is used as a filter step before the block locality cache as it is also used in the exact data deduplication systems presented by Zhu et al. [ZLP08] and at least some versions of the Venti system [QD02, RCP08, LH08]. When a Bloom filter is used, a chunk the BLC is asked for exists with a high probability in the chunk index. The Bloom filter especially helps to overcome the disk bottleneck in the first backup generation, as most chunks cannot be found in the chunk index. The Bloom filter also is important in later generations as often even the amount of new chunks is high enough to create a disk bottleneck, even if the BLC is able to predict the IO accesses of all already stored chunks.

While the block chunk cache might be shared between data streams, it is important that the difference cache is private per incoming data stream. Since the difference value cache only contains a few values, the overhead is negligible.

11.1.1 *Refinements*

There are two refinements of the strategy presented above: Minimal difference counter values and prefetching.

It is possible to store an additional counter for each cached difference value. The counter is incremented when the difference value was used to create a cache hit. A difference value in the cache is then only used to fetch blocks into the cache if the counter exceeds a threshold. The intuition is that a cached difference value has to prove itself before it is acted on.

By ignoring the difference values that have not yet proven to be helpful, fewer blocks are fetched into the cache. Especially if some difference values are the result of noisy fingerprints and the diff cache is large, it may be better to wait and observe for a while. On the other hand, if the new difference value actually reflects a permanent change of the alignment, ignoring the difference value will cause some mispredictions that have been avoidable.

Another refinement is the prefetching of blocks: When a block is loaded into the cache, the following n blocks are also fetched from disk. The value n is the prefetching window. Usually, prefetching does not cause additional disk IO. The block index is often organized in an ordered persistent data structure as a B-Tree or an LSM-tree. Then, multiple consecutive block recipes stored within the same data page. Therefore, the data of the prefetched blocks is often stored in the same, already cached, page. An alternative, requiring a deeper integration with the data structure implementation, is to load all blocks that are stored in a data page.

The pages are cached by the B-Tree or LSM-tree implementation. If locality exists, the prefetched block recipes will probably be used in the near future. On the other hand, prefetching causes the eviction of some cached blocks from the cache.

11.1.2 *Design Comparison*

The novel Block Locality Caching scheme uses a different approach to overcome the disk bottleneck than existing ones. Related approaches for backup environments are the Locality-Preserving Caching Scheme of Zhu et al. [ZLP08], Sparse Indexing [LEB+09], and Extreme Binning [BELL09].

These approaches share the general scheme. The data is chunked and fingerprinted. Then the approaches decide if a chunk has already been stored or if the chunk is new using a prediction based on different properties like similarity or locality. If the approaches are able to classify a chunk based on the prediction scheme, it is not necessary to perform a chunk index lookup, which in most cases will cause disk IO operations.

To enable the chunk prediction, all approaches load some kind of meta data into main memory. These load operations cause disk IO operations. However, the costs are expected to be amortized by using the information in the data structure to process the current or future chunks.

The Locality-Preserving Container-Caching approach and the Block Locality Cache use a cache to keep loaded fingerprints for future reference assuming future chunks can probably be found in the cache. Sparse Indexing and Extreme Binning also use a small cache to avoid reloading recently loaded segments or bins. However, the cache is an optimization

and not essential to these approaches. The main memory in these approaches is used to hold necessary, additional index structures.

In general, there is a trade-off between the prediction quality (ratio of chunk fingerprints that can be processed using only the cache) and the disk IO for loading data structures or other resources. A misprediction means that the approach was not able to classify a chunk as unique based on the current state.

In exact deduplication systems, a mispredicted chunk causes a lookup operation in the disk-based chunk index. In approximate deduplication systems, a misprediction does not cause additional disk IOs, but may result in a reduced deduplication ratio because a chunk that actually has been stored before is stored again.

However, the prediction approach and the exact/approximate property are not inherently coupled. The two approximate approaches can be seen to be a combination of an approach for fingerprint prediction and the approximate property, which trades deduplication quality for throughput for mispredicted chunks by not holding a full chunk index. The approximate approaches can be made exact by introducing a full chunk index lookup after the prediction fails. Similarly, it is possible to create approximate variants of the Container-Caching and the Block Locality Cache approaches. The approach proposed by Guo et al. [GE11] can actually be interpreted as an approximate variant of the Container-Caching approach.

The Block Locality Cache uses significantly smaller caches than necessary in a container-caching based scheme. The reason is that in general the backup is moving through the blocks of the previous generation, making the caching of old blocks less likely to be effective. However, the Bloom filter data structure to avoid most chunk index lookups on new chunks dominates the overall memory consumption. While Lillibridge et al. point out that approximate deduplication approaches using a sparse index are likely to be less memory efficient [LEB+09], a Bloom filter size of 25 GB for a post-deduplication storage size of 100 TB is feasible nowadays.

The segments in the Sparse Indexing approach store data that is similar to block recipes, a logical sequential list of fingerprints in the order they arrived. However, the approaches differ in that Sparse Indexing segments are additional data while the block recipes are needed for the basic functionality of the block-based data deduplication. The approaches also differ in the way this data structure is used for prediction. Sparse Indexing uses segment similarity, the Block Locality Cache uses the analogy between the data stream and the block recipes to capture temporal locality.

From the point of duplicate detection and overcoming the disk bottleneck, the biggest difference between Sparse Indexing and Extreme Binning is that segments are always a temporarily ordered sequential list of the stored chunks, while the bins only contain unique chunks and are more likely to fragment. Furthermore, while both approaches uses similarity detection, the similarity detection approach of Sparse Indexing is a better predictor for the similarity. Sparse Indexing uses multiple hooks per segment and searches the existing segments with a high overlap while Extreme Binning uses similar bins based on a single hook fingerprint.

The sizes of the IO requests issued by the approaches differ, too. A block in the Block Locality Cache usually covers 256 KB of data with 32 8 KB chunks. Assuming additional 4-bytes metadata per chunk, the block recipe has a size of around 800 bytes. Multiple block recipes are, therefore, stored in a single index page. The block index is organized using an ordered persistent data structure to make sequential scans efficient. In the Container

Caching approach, the fingerprints of a container are fetched from disk. Each container stores around 1,000 chunks. Therefore, assuming a 20 bytes fingerprint with 4 bytes additional data, the metadata of a container has a size of around 64 KB. The segments in Sparse Indexing usually are larger than containers with values between 1 and 20 MB [LEB+09].

In Extreme Binning, the bins are fetched from disks. In contrast to the other approaches where the fetched data structure has a fixed size (block, container, segment), bins are growing over time. While a bin in the beginning contains only the unique chunk fingerprints of the initial block, over time, more and more unique chunks may be added to the same bin.

11.2 EVALUATION: TRACE-BASED SIMULATION

The Block Locality Cache is evaluated using trace-based simulations and a prototype implementation to verify the simulation results.

The trace-based simulation uses the *UPB* data set and *JGU* data set. The *ENG* data set has not been used because it does not reflect the locality pattern as it has been collected using a high number of concurrent threads. The source code of the trace-based simulation is available at `https://github.com/dmeister/fs-c-analytics`.

The simulated setting is a single-stream backup for exact block-based deduplication systems using a fingerprinting-approach with a container-based storage. The backup data is stored consecutively in a single backup file.

The Block Locality Cache is compared to three existing approaches:

- a Locality-Preserving Container-Caching scheme,
- an exact variant of Sparse Indexing,
- a block-based, single-node, exact variant of Extreme Binning.

These are all variants of major approaches to avoid the chunk index bottleneck using a prediction for backup based deduplication systems.

The simulation of the existing concept serves two purposes. It provides a baseline to compare the novel approach against. In addition, it is a "Repeated Research" serving as an independent validation of previous results [Mat04].

It deserves some discussion why exact variants of approximate deduplication approaches are used for comparison.

The focus in this evaluation is to compare the prediction approaches and, as discussed before, the exact/approximate property is not inherently coupled with the prediction approach. However, comparing approximate and exact approaches at the same time is difficult as there is no clear way to weight the trade-off between deduplication ratio and reduced disk IO. This difficult trade-off does not exist for exact systems where the only goal of the prediction approaches is to maximize overall throughput by minimizing overall disk IO operations. To enable a comparison of the prediction quality of the approaches exact variants are used.

The BLC approach is simulated using a block size of 256 KB, diff cache values between 1 and 16, and block chunk cache sizes of up to 2048 blocks. Minimal difference value is explored with values between 0 and 8. Prefetching simulates that when a B-tree page is read from disk, all blocks stored on the page are loaded into the cache.

The Container-Caching approach is based on Zhu et al.'s work [ZLP08]. It is simulated by sequentially building containers up to 4 MB and caching the chunk fingerprints of a number of containers in RAM. The container cache is maintained using a LRU cache replacement policy. A small chunk cache is used to avoid holding all chunks from the container of highly referenced chunks in the container cache and occupying cache space. These highly used chunks are probably often stored in the chunk cache. The configuration space of the Container-Caching approach is small. All configurations use 4 MB container containing around 1,000 chunks. The container cache is evaluated for 32, 64, 128, 256, 512, and 1,024 containers. The chunk cache is evaluated with 512 and 1,024 chunks.

An exact variant of Sparse Indexing [LEB+09] is implemented by forming segments using variable-sized segmenting. Then, hooks are found by hashing the fingerprints and selecting a fingerprint as hook if and only if the correct number of most-significant bits is set. Champion segments are searched as described in [LEB+09]. If a chunk cannot be correctly predicted by selected segments, the chunk index is queried instead of storing the chunk a second time.

The configuration parameter space is chosen similar to the explored parameter set in [LEB+09]. The explored configuration space consists of all combinations of the following parameters:

SEGMENT SIZE: 2 MB, 4 MB, 8 MB, 10 MB, and 20 MB

SEGMENT CACHE SIZE: 4, 16, and 64 segments

SAMPLING RATE: $1/2^5$, $1/2^6$, $1/2^7$, and $1/2^8$,

MAXIMAL NUMBER OF SEGMENTS PER HOOK IN SPARSE INDEX: 2, 4, and 8

MAXIMAL HOOKS PER SEGMENT: 1, 2, 4, and 8

Variable-sized segmenting was used by hashing the chunk fingerprints and creating a new segment after a chunk with the appropriate probability.

One difference, which is unavoidable due to the data sets, is that the expected chunk size is 8 KB, while Lillibridge et al. argue that with Sparse Indexing 4 KB chunks are feasible. The cache size is chosen significantly smaller than for the Container-Caching approach to accommodate for the RAM usage by the sparse index.

An approach based on Extreme Binning [BELL09] is also simulated. It was modified to be exact and to use blocks of data instead of files, as this is more suitable for a setting with a single large backup file. The approach is called "Exact Block-based Extreme Binning". Instead of files, blocks up to a configurable size are used as the unit for the representative fingerprint selection. Block sizes between 256 KB - 20 MB have been explored.

If the representative fingerprint of a block is not known, a new bin is created and all unique chunk fingerprints are added to the new bin. If the representative fingerprint is known, the bin of the representative fingerprint is loaded (or fetched from a LRU cache). For all chunk fingerprints not found in the bin, the chunk index is queried. If the chunk is not found in the chunk index, the chunk is added to the bin.

The full file fingerprint has been modified to calculate the hash of a block instead of the complete file. The system looks the chunk up in a full chunk index if the full-block fingerprint comparison and the bin level prediction fail.

It is important to notice that the state-of-the-art approaches are simulated and compared here. However, this is not a direct comparison with the concrete implementations of the

approaches in the form of the EMC Data Domain product line or the HP StorageWorks D2D backup systems.

All approaches are assumed to use a Bloom filter to avoid chunk index lookups on chunks that are not yet stored. Especially the Bloom filter guards against a significant slowdown in the first backup generation, as during the first backup run most chunks have not been stored yet. In the simulation, new chunks are only "counted" as chunk index lookup with a probability of 1%.

The main evaluation factor is the number of IO operations needed by the different approaches, as the main purpose is to overcome the disk bottleneck. Therefore, only the IO operations issued to perform or to accelerate chunk lookups are counted.

Container writes, chunk index updates, or bin updates are not counted against the approaches as writes can be delayed, aggregated and sequentialized using standard techniques such as write-ahead logs. These techniques migrate the costs of the update operations and reduce the costs in the critical data path.

For the Block Locality Cache, the IO load is the sum of all loaded blocks as well as the remaining chunk index lookups. For the Container-Caching approach, all loads of containers into the cache and all remaining chunk index lookups are counted. The IO load of the Exact Sparse Indexing approach is the combination of all segment load operations as well as the remaining chunk index lookups. For the modified Extreme Binning approach, the loads of all bins are counted.

The number of lookup-related IO operations serves as a proxy for the system throughput. Especially with techniques that offload the chunking and fingerprinting to the clients [LGLZ12, EMC12], the number of IO operations on the backup system is still an important factor for the ingest performance. However, without a full and optimized implementation of all approaches, a throughput comparison is not possible.

This mode of evaluation assumes that IO operations involve significant seek operations. During the evaluation of the IO pattern of each approach, it will also be discussed if this assumption has been valid or if it was too conservative.

The goal of these approaches is to overcome the chunk lookup disk bottleneck. For the purpose of this evaluation, I define that an approach is able to overcome the disk bottleneck if the number of chunk lookup related IO operations is reduced by more than 98%. If 98% of the IO operations can be avoided, a single hard disk (dedicated to chunk index lookups) can provide a sufficient random read rate to process 100 MB/s, which is also approximately the sequential write throughput of a single disk.

The discussion solely focuses on the second and later backup generations as they represent the steady-state of data deduplication systems. The first data generation is a special situation as a significant lower potential for data deduplication can be used and furthermore the locality and similarity properties used by all approaches are not yet available in the data set.

This simulation is the first time that these approaches are compared using the same data sets. When the results of the simulations are reported, the results of the best configuration found are stated for each approach and data set until stated otherwise.

First, an overview of the results is provided. Then, the results of the different approaches are investigated in more detail. Such detailed discussions provide significant insights into their behavior and properties even of the existing approaches.

Table 6: IO operations per non-fresh backup run using the best configurations.

Approach	Average IO	Average Improvement
Block Locality Caching (BLC)	66,917	99.9%
Container-Caching	270,844	99.5%
Exact Sparse Indexing	158,625	99.7%
Exact Block-Based Extreme Binning	318,545	99.4%
Baseline	55,738,553	

(a) *UPB* data set.

Approach	Average IO	Average Improvement
Block Locality Caching (BLC)	289,300	99.8%
Container-Caching	485,257	99.6%
Exact Sparse Indexing	613,338	99.6%
Exact Block-Based Extreme Binning	7,693,733	94.4%
Baseline	137,183,532	

(b) *JGU* data set.

11.2.1 *Simulation Result Overview*

The first column of Table 6 shows the average number of IO operations of all non-fresh backup runs for both data sets using the best configuration found for each approach. The baseline row shows the average number of IO operations using a disk-based chunk index, assuming no cache or Bloom filter.

The tables show that the approaches are able to avoid most chunk index lookups and reduce the IO requirements significantly. With savings of over 99.8% in both data sets, the BLC approach as well as Container-Caching and Exact Sparse Indexing overcome the chunk lookup disk bottleneck. However, the differences between the approaches are significant.

In both data sets, the novel BLC approach needs the least number of IO operations. It needs 58% less IO operations than the second best approach, Exact Sparse Indexing, in the *UPB* data set and 40% less than the second best approach in the *JGU* data set.

The similarity-based approaches, Exact Sparse Indexing and Exact Block-based Extreme Binning, perform better, compared to the Container-Caching approach, in the *UPB* data set than in the *JGU* data set. In the latter, Exact Sparse Indexing needs more IO operations than the Container-Caching approach and the Extreme Binning variant drops to a level where it no longer fully overcomes the disk bottleneck.

The average values of the IO operations in Table 6 do not contain the IO operations for the first backup generation, which is a special case. For completeness, the results for the first backup generation are stated here. Even in the first generation, the BLC and Container-Caching enable a total saving of 98.8%. The Container-Caching needs the least amount with 536,996 (*UPB*) and 1,686,339 (*JGU*) IO operations. The BLC needs 9.8% (UPB) and 4.5% (JGU) more IO operations than the Container-Caching. Exact Sparse Indexing and Exact Block-based Extreme Binning are more expensive with 5.5 respectively 9.9 times

Table 7: Best Block Locality Caching configurations found.

Parameter	Value	Description
Diff Cache Size	4 (*UPB*) 8 (*JGU*)	4 or 8 most-recent diff values are probed
Min Diff Value	0	A diff value is used even if new
Block Cache Size	2,048	2,048 blocks are cached
Chunk Cache Size	1,024	1,024 chunks are stored in a first-level LRU cache
Block Index Page Size	32 KB	The block index uses 32 KB pages

the number of IO operations in the *UPB* data set and 4.6 respectively 7.0 times the number of IO operations in the *JGU* data set compared to the Container-Caching approach. Therefore, in the first backup generation, these approaches are not able to overcome the disk bottleneck.

11.2.2 *Results: Block Locality Caching*

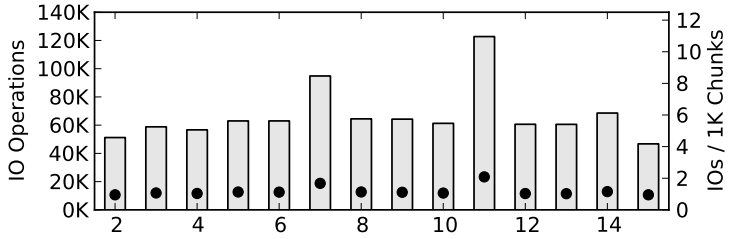
The best configuration for the Block Locality Caching approach in the *UPB* data set uses a block chunk cache containing 2,048 blocks, a difference cache with 4 values with no minimal difference counter. This is denoted as the 4/0 configuration. The best configuration in the *JGU* data set is the 8/0 configuration. Here, the 4/0 configuration needs on average 0.5% more IO operations (257,631 instead of 254,338). Table 7 shows the full set of configuration variables.

Figure 28a shows the number of IO operations per weekly backup run for the *UPB* data set (bar). The dot values shown on the second y-axis normalize the number of IO operations per 1,000 chunks. The outliers in week 7 and 11 are caused by an under average prediction quality leading to additional chunk index lookup operations. In week 7, 3.6 times more chunks are mispredicted than in an average week (40,204 chunks). In week 11, this number increases to a factor of 3.9 compared to an average week (43,734 chunks). This is significant as usually the number of mispredictions is stable with a standard derivation of 1,698 when these two weeks are excluded. As week 11 also has the highest number of new chunks of all non-empty backup generations, the workload in week 11 has been significantly different from the other weeks.

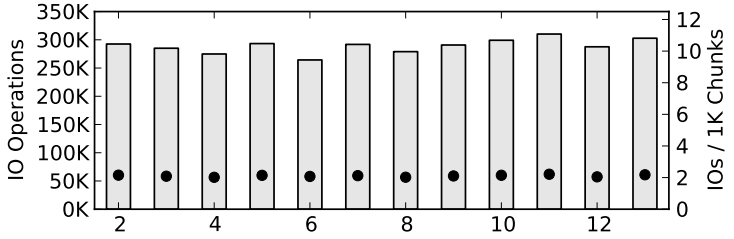
The high number of IO operations is not only the result of mispredictions, but also from chunk index lookups for new chunks where the (conceptual) Bloom filter returned a false positive result. Even in these two weeks the chunk lookup disk bottleneck is avoided.

Figure 29 shows the block identifiers of all fetch operations in weeks 8, 11, and 13 for the 4/0 configuration. The figures serve as examples for the block read pattern seen in all weeks. The complete set of figures for all weeks and all approaches can be found in the appendix.

The solid line in the middle covers most of the IO operations. These are the block fetch operations when the alignment is stable. The IO operations above the solid line are caused



(a) *UPB* data set, 4/0 BLC configuration.



(b) *JGU* data set, 8/0 BLC configuration.

Figure 28: IO operations per backup run for BLC approach. The dots denote the IO operations normalized to 1,000 processed chunks.

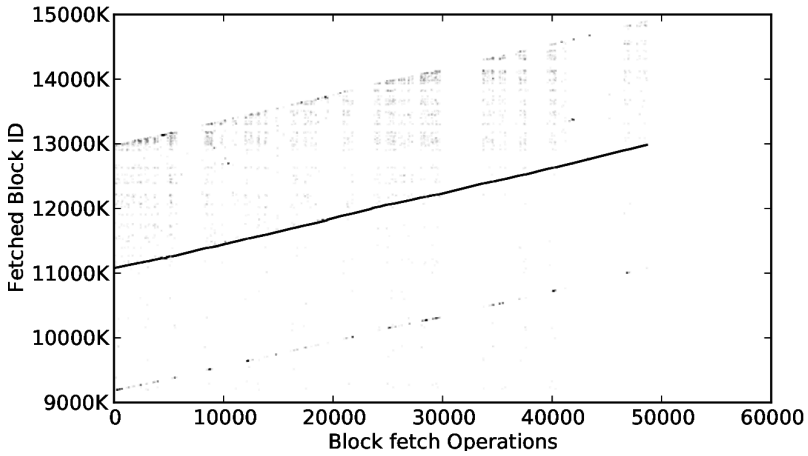
by intra-backup redundancies, i.e., chunks that are redundant within a backup generation. These blocks have been created in the current backup run.

The dots below the solid line denote fetch operations of blocks written in the second latest backup generation. This means that there are requests for some chunks that existed in the second latest backup generation, but do not existed in the latest backup generation. None of the backup runs contains chunks that existed in the third last backup generation, but not in the newest two. Thus, it is important to fetch block recipes from at least two generations. The number of these early generation fetches is not very high, but significant enough that the overall result is worse if the fetches are limited to the latest backup generation only.

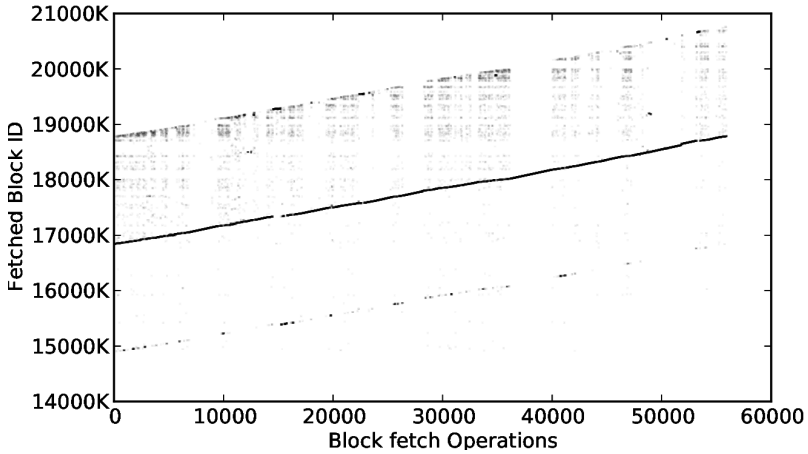
In the outlier week 11, 16% more blocks are fetched due to the difficulties in that week to find a good alignment. Even for that week, the prevalent IO pattern is the straight line of correctly aligned blocks. Then, most block requests are sequential on disk and, thus, are not forcing timely disk seek operations. Therefore, the random IO assumption has been conservative.

Figure 30 shows the development of the prediction quality in weeks 8, 11, and 13 for the 4/0 configuration. For week 13, the graph shows that the prediction quality always stays above 99%. It also shows that the prediction quality includes a temporal burstiness. When there are mispredictions it is likely that other mispredictions occur in the near future. In week 11, the prediction quality drops to only 80% in one phase.

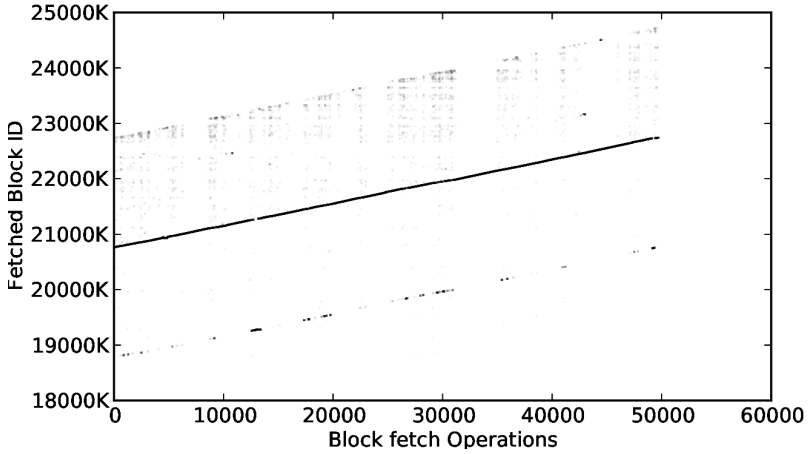
Figure 28b shows the IO numbers of the *JGU* data set. Normalized per 1,000 chunks, the BLC needs more IO operations for the *JGU* data set. While in most weeks only around one IO per 1,000 chunks is needed for the *UPB* data set, around two IO operations per 1,000 chunks are needed in the *JGU* data set in all weeks.



(a) Week 8.

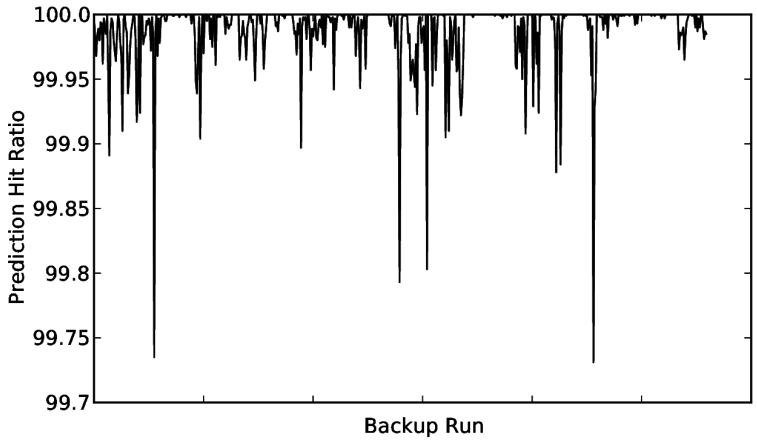


(b) Week 11.

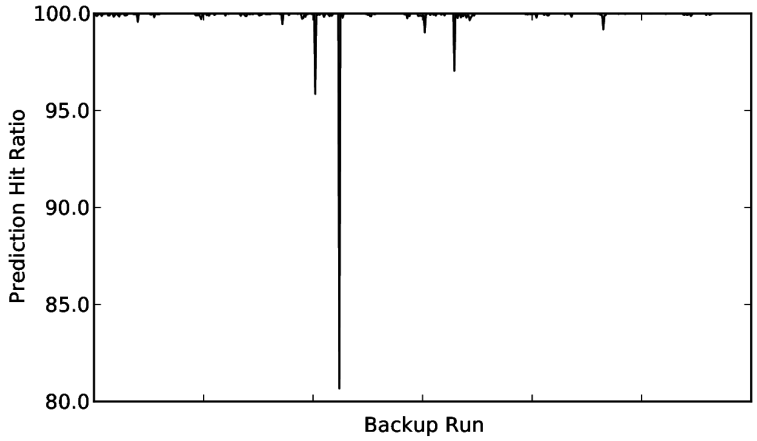


(c) Week 13.

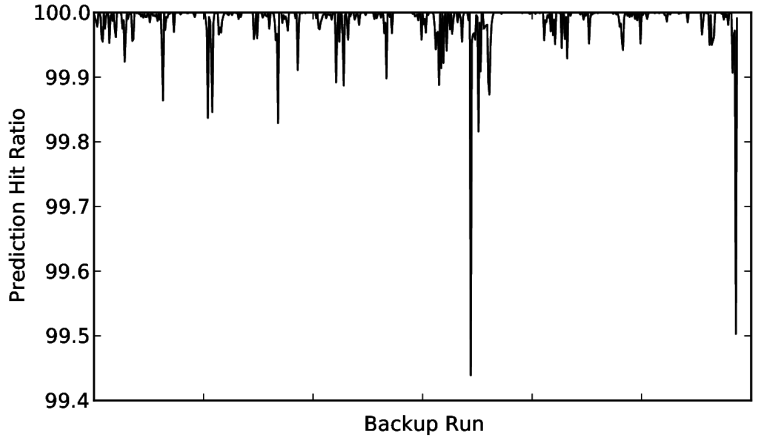
Figure 29: BLC block fetch operations in *UPB* dataset.



(a) Week 8.



(b) Week 11.



(c) Week 13.

Figure 30: BLC miss rates in *UPB* dataset.

The reason for the worse performance in the second data set is not the prediction quality alone. While the block chunk cache hit rate is reduced from 99.98% on average in the *UPB* data set to 99.94%, it is more relevant that more IO operations are issued for block fetch operations to achieve the block chunk cache hit rate. In the *UPB* data set, on average 0.86 block fetch IO operations are issued per 1,000 chunks. The number increases to 1.50 fetch-induced IO operations per 1,000 chunks in the *JGU* data set.

With the 4/0 configuration, 1.46 fetch-induced IO operations are issued per 1,000 chunks in the *JGU* data set. But the reduced about of fetches is combined with a slightly reduced block cache hit ratio (99.943% to 99.937%). This hit ratio reduction represents enough additional IO operations so that the total number of IO operations is 0.5% higher than with the 8/0 configuration.

So, the 8/0 as well as the 4/0 configuration need more than 66% more fetch IO operations and still the cache hit rate is less than in the *UPB* data set. One reason for this is that data sets that are more difficult to predict creates additional IO operations in two ways. A reduced cache hit rate creates the additional chunk index lookups and additional fetch operations. The number of fetch operations increases, e.g., because more blocks need to be fetched based on the values in the diff cache before there is block cache hit rate.

The comparison of the 8/0 and the 4/0 configuration in the *JGU* data set illustrates that while a larger diff value cache might lead to more unsuccessful fetch operations, the higher hit ratio outweighs that longer fetch chains and in the end also reduce the number of issues fetch operations.

Finally, the effects of the major configuration parameters are discussed, focusing on the *UPB* data set.

The size of the difference cache is the most important configuration parameter. As discussed above, it provides a trade-off. A larger diff cache tries more different blocks before giving up and declaring that a chunk is a miss. This can increase the number of IO operations spent on fetching blocks, but also provides more opportunities to find a good alignment, i.e., a block with the searched fingerprint. It is therefore better in handling noisy patterns in the data stream, which may increase the prediction hit ratio and reduce the number of fetched blocks. The configuration values 4/0 (resp. 8/0 for *JGU*), whose results are presented above, provide for these data sets the best compromise in this trade-off.

Figure 31 shows the average total number of IO operations, the IO operations caused by the BLC for fetching block recipes, and the number of BLC misses for the best configurations with a diff cache size of 1, 2, 4, 8, and 16. The best configurations with diff caches of size 2, 8, and 16 also perform well with less than 5% more IO operations than the configuration with a diff cache size of 4. A diff cache with only a single value actually causes 92% more IO operations. The intuitive assumption that a single alignment is sufficient is false given the noise found in real-world traces.

The best configuration with a diff cache size of 16 is interesting as it has a non-zero minimal difference counter, which has been presented as a refinement. The results show that this refinement is not providing better results given the data sets for configurations with a small difference cache. Only when the difference cache becomes larger, this refinement yields better results. The 16/0 configuration needs 7.7% more IO operations than the 16/1 configuration.

The question on the importance of having a difference cache containing more than a single diff value can also be approached from another direction. In the second week of

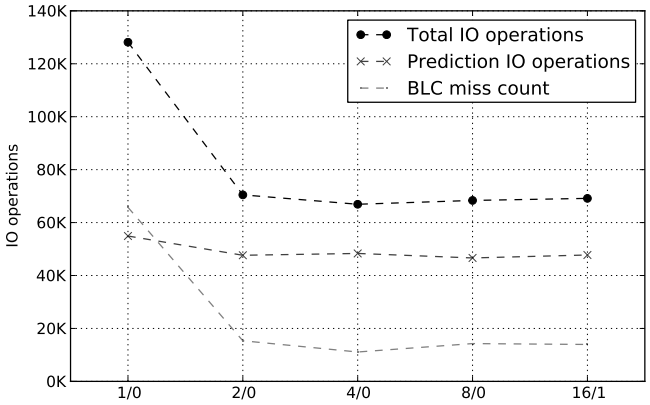


Figure 31: Comparison of different diff cache sizes for the BLC approach in the *UPB* data set.

the *UPB* data set using the 4/0 configuration, there are 1,889 different leading diff values, which have been able to successfully predict chunks.

The leading diff values are often able to predict the next chunk correctly so that there are "leading chunk runs" in which the leading chunk is not changing. One way to measure the length of a run is to count how many hitting fetch operations have been performed with the leading chunk before a different leading chunk takes over. On average, a leading diff value is used by 22.4 hitting fetch operations. The maximal run length is 3,777 hitting fetch operations.

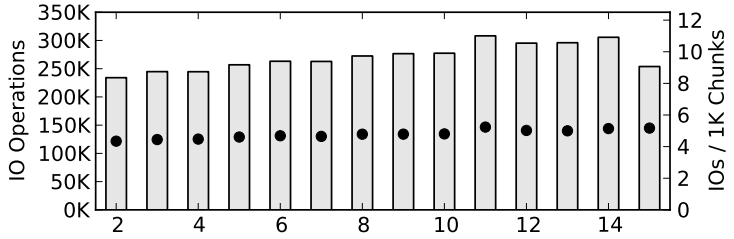
22% of all leading diff values are used in more than one run. On average, a leading diff value is used in 2.7 runs, while a reused leading diff value is referenced on average by 8.5 runs. The maximal number of different runs of a leading diff value is 118.

Critical for the difference cache is that often recent leading diff values are reused. In 59.2% of all changes in the leading diff value, the new diff value is still in the diff value cache. In 65.6% of all changes in the leading diff value, the absolute difference between the diff values is 1.

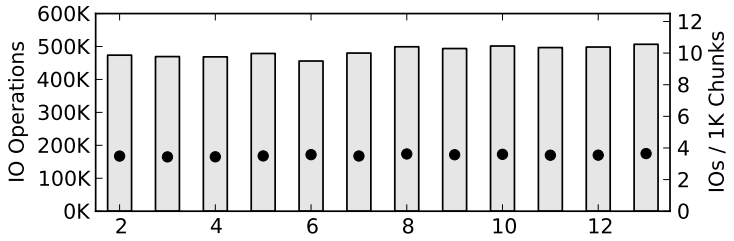
The size of the block chunk cache is not a sensitive configuration parameter. Cutting the block chunk cache in half from 2,048 blocks to 1,024 using the 4/0 configuration increases the amount of IO operations by 1.9% in the *UPB* data set. This is the second best BLC configuration explored during the simulation runs.

Even a cache of 256 blocks only increases the average number of IO operations by 5.4%. Similarly, increasing the block chunk cache to even 8 K blocks would only eliminate additional 3.5% of the IO operations. While a larger block chunk cache dominates smaller block chunk cache sizes, larger caches have a vanishing effect on the overall performance. The reason is that the BLC aims to find a good alignment with a previous backup run and then runs over all previous blocks in a mostly sequential way.

With a block chunk cache size of 2048 blocks and a chunk cache size of 1024, around 0.1% of all chunk fingerprints are held in memory (*UPB* data set). With the Bloom filter is configured with a RAM consumption of 1 byte per chunk [LEB+09, ZLP08], the BLC uses main memory with a size equivalent to 5.1% of all chunk fingerprints.



(a) *UPB* data set.



(b) *JGU* data set.

Figure 32: IO operation per backup run for the Container-Caching approach. The dots denote the IO operations normalized to 1,000 processed chunks.

The Block Locality Cache approach needs the least amount of IO operations compared to the other approaches in both data sets by a significant margin.

11.2.3 *Results: Container-Caching Approach*

The Container-Caching approach fetches the container metadata into a cache and uses the cached fingerprints to avoid chunk index lookups. The approach performs well in both data sets and it is able to avoid the disk bottleneck.

The best configuration uses a container cache of 1,024 elements with around 1,000 chunks per container. Depending on the week, between 2.3 and 2.8% of the unique chunks of the *UPB* data set can be stored in the cache. This configuration uses around 16 times more main memory for caching than the Block Locality Cache, excluding the Bloom filter. With the Bloom filter, main memory is used equivalent to 7.3% to 7.8% of the unique chunk fingerprints and around a third more than the Block Locality Cache.

Figure 32a shows the number of IO operations per backup run for the *UPB* data set. In most backup runs, the container-caching approach needs about four times more IO operations than the best BLC configuration. The corresponding results for the *JGU* data set are shown in Figure 32b.

One thesis stated in this part is that the approach is prone to aging effects. The locality of the first backup generation becomes a less good predictor over time. The normalized IO performance (dots in Figure 32a) shows a declining performance in the *UPB* data set. It increases from 4.5 IO operations per 1,000 chunks to 5.0 in 15 weeks. A linear regression shows that the best fitting increase rate is 0.06 per week. The number of loaded containers per 1,000 chunks increases from 2.1 to 2.6 (+19%). Here, the linear regression predicts a weekly increase of 0.03.

However, the behavior is not that clear in the *JGU* data set as the variance in both metrics is high. Still, both metrics have a small increasing trend using a linear regression model. The normalized total number of IO operations is fitted with an increase of 0.014 and the normalized number of loaded container with 0.005.

This trend can be explained by looking at the IO pattern issued to load the container fingerprint information. Figure 33 shows the container read pattern in weeks 8, 11, and 13. The figures of container read pattern for all weeks can be found in the appendix.

The solid lines are the expected container read operations based on the captured locality. All containers until the container around the 35,000 mark have been created in the first backup generation.

The set of lines above the solid line is important and explains the aging. These container reads come from chunks that have been written in later backup generation and are, as expected, reused in a similar order in the following backup generations. Each new backup generation adds a few new containers that are fetched from disk to cache in later backup generations. Therefore, each new backup generation adds a new line to the read pattern.

This effect has also been called "fragmentation" by Zhu et al. [ZLP08]. It reduces the effectiveness of the Container-Caching approach in the long run [ZLP08] . They mention that de-fragmentation is possible, but do not describe a possible de-fragmentation mechanism. De-fragmentation approaches targeted on improving the restore performance, e.g., by Kaczmarczyk et al. [KBKD12] may also avoid the aging effect.

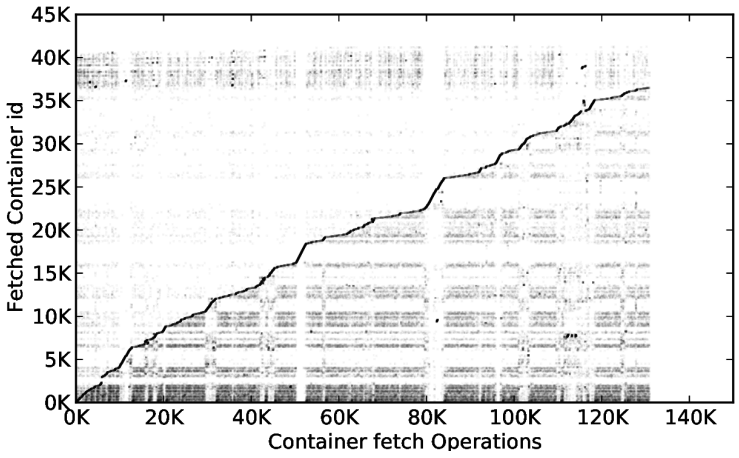
The read operations below the solid line load containers that contain chunk data, which is used more than once in a backup generation, i.e., these read operations are caused by intra-backup redundancy. While a de-fragmentation approach may be able to reduce the aging effects, the effects of intra-backup redundancy are problematic for the approach.

As long as containers are used as basis for the locality detection and the cache is not large enough to hold a large fraction of the intra-backup redundancy chunks, the intra-backup redundancy, even in the absence of aging-based fragmentation, renders a linear chunk ordering impossible. This "de-linearization" leads to significant re-load operations for containers.

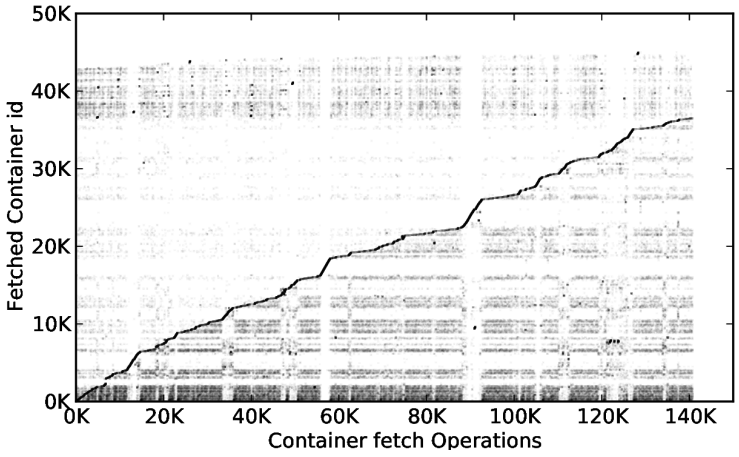
The de-linearization has been investigated for backup systems by some researchers as it also reduces the restore performance. The known approaches selectively store chunks redundantly if this reduces the de-linearization [TYFS12, LEB13]. These approaches may also reduce the number of IO operations needed for chunk prediction. However, if these approaches are used, the deduplication system is only approximate as not all redundant chunks are deduplicated.

It is easy to see that a larger container cache size dominates small caches in this approach. Reducing the container cache size to 512 containers of around 1,000 chunks in the *UPB* data set increases the number of IO operations by 28% to 347,992 and by even 60% to 434,490 if further reduced to 256 containers. In the *JGU* data set, the effect is similar but not that strong. 10% more IO operations are necessary with a container cache of 512 containers and 23% more operations are necessary if only 256 containers are cached.

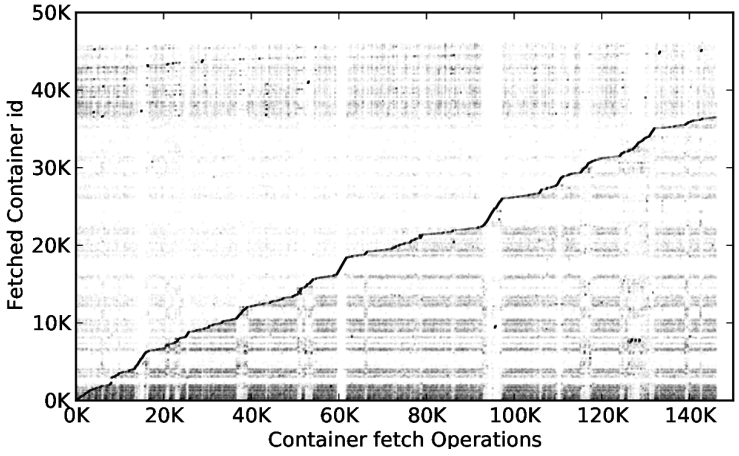
Conceptually, an increased cache size is reducing the re-loading of previously fetched containers due to inter-backup data deduplication. Therefore, a larger cache is, mainly, reducing the load operations below the solid lines in Figure 33. However, the simulation shows that a LRU cache that absorbs a large amount of these intra-backup container re-loads needs to be very large. Custom caching strategies tailored to this scenario can be investigated, but they are beyond the scope of this thesis.



(a) Week 8.



(b) Week 11.



(c) Week 13.

Figure 33: Container read operations in *UPB* data set.

Table 8: Best Exact Sparse Indexing configuration (both data sets).

Parameter	Value	Description
Sampling Rate	$1/2^5$	Every 32nd chunk is sampled as hook
Max. Segment Count	8	A segment is compared against up to 8 existing segments
Max. segm. per hook	8	Per hook chunk, up to 8 segments are stored
Segment Size	20 MB	A segment contains chunks of 20 MB data on average
Cache Size	64	64 segments are cached

The Container-Caching approach is able to overcome the disk bottleneck, too. However, it needs more IO operations than the BLC approach. Furthermore, even in these relatively short data sets, the approach shows signs of aging effects.

11.2.4 *Results: Exact Sparse Indexing*

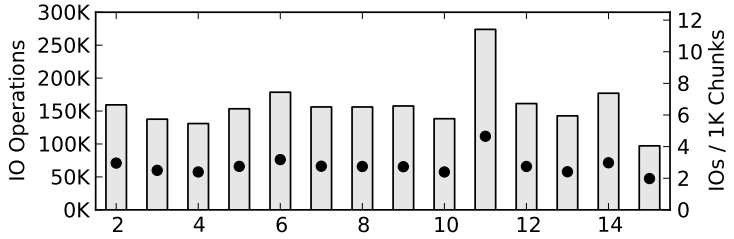
The exact variant of the Sparse Indexing approach performs well in both data sets with over 99% of the IO operations saved compared to the baseline. In the *UPB* data set, the approach needs 41% less IO than the Container-Caching approach. However, the best explored configuration for the Exact Sparse Indexing needs around 2.4 times more IO operations than the best BLC configuration for the *UPB* data set and 2.1 times the number of IO operations in the *JGU* data set.

The best configuration is identical in both data sets. They are shown in Table 8. The IO operations issued per backup run in the *UPB* data set are shown in Figure 34a. Figure 34b shows the same information for the *JGU* data set.

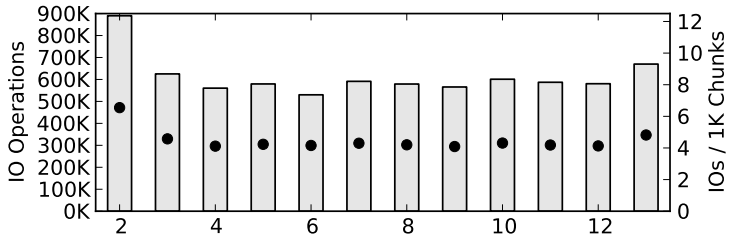
In all weeks, the approach needs more IO operations than the Block Locality Cache. The main reason for this is a higher misprediction rate. On average, Exact Sparse Indexing mispredicts 6.3 times more chunks than the BLC (all results here for *UPB* data set). In the exact-variant explored here, this leads to additional chunk index lookups. The approach needs 4.6 times more chunk index lookup IO operations. On the other hand, the prediction needs significantly fewer IO operations than needed to fetch the blocks. The BLC issues 36% more IO operations for block fetching than Exact Sparse Indexing needs to fetch segments from disk. In the end, the better prediction outweighs the cheaper approach of performing the prediction.

The original approximate version of Sparse Indexing would store 14.5% more chunks per week on average. This, however, would avoid all chunk index lookup operations. The average number of IO operations reduces to 40,562 per week, nearly 40% less IO than with the (exact) Block Locality Caching approach.

Figure 35 shows the segment read pattern for the weeks 8, 11, and 13 in the *UPB* data set using the best configuration. The figures for the segment read pattern for all weeks can be found in the appendix.



(a) *UPB* data set.



(b) *JGU* data set.

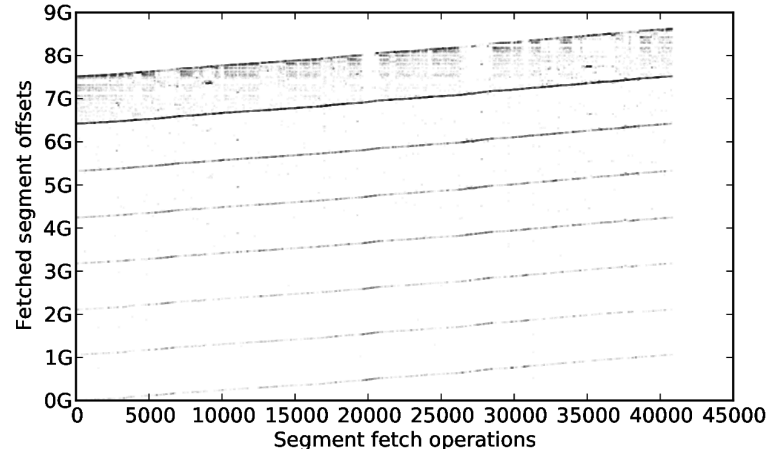
Figure 34: IO operations per week using the best explored Exact Sparse Indexing configuration. The dots denote the IO operations normalized to 1,000 processed chunks.

The pattern shows that the segment fetches are aligned on multiple previous backup runs. The reason is that each hook fingerprint entry stores up to a configurable number of segments (8 in the best explored configuration). Given that most hooks are only used once per backup, the segments point to a set of previous backup runs, which leads to these alignments. A smaller number of segments per hook would reduce the number of these alignments, but also reduce the prediction quality. The values of 2, 4, and 8 segments per hooks have been explored. Smaller values always lead to a higher number of total IO operations. For example, the best configuration with 4 segments per hook requires 5.6% more IO operations.

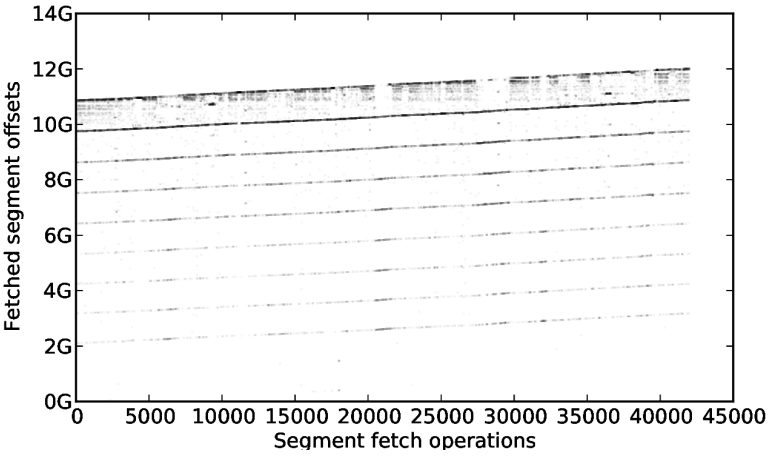
The best configuration is the most memory-consuming configuration in the configuration space proposed by Lillibrigde et al. [LEB+09]. Especially, a sampling rate of $1/2^5$ causes a sparse index to hold hook entries for approximately 3.2% of all chunks. The total main memory consumption increases to 8.2% of all chunk fingerprints when considering the Bloom filter. However, the Bloom filter is only necessary in the exact variant of Sparse Indexing.

Therefore, the main memory consumption is slightly larger than for the Container-Caching approach. However, smaller sampling rates severely reduce the efficiency of the prediction. The best configuration in the *UPB* data set with a sampling rate of $1/2^6$ needs 75% more IO operations (278,278 IOs on average). Even 3.8 times the number of IO operations are needed for the best configuration with a sampling rate of $1/2^7$ (608,025 IOs on average).

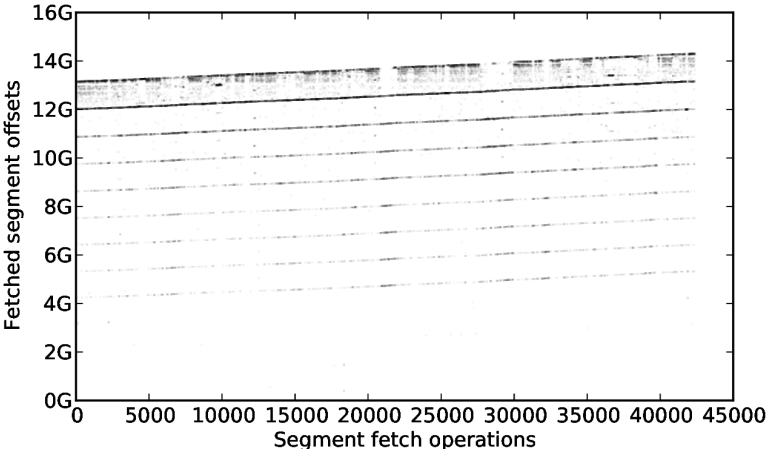
The reason for this effect lies in the increasing number of mispredictions. With a sampling rate of $1/2^6$, 2.1 times the number of chunks cannot be predicted using the approach, leading to additional chunk index lookup operations. With a sampling rate of $1/2^7$, even 5.2 times more chunks cannot be predicted. While the number of segment load operations is



(a) Week 8.

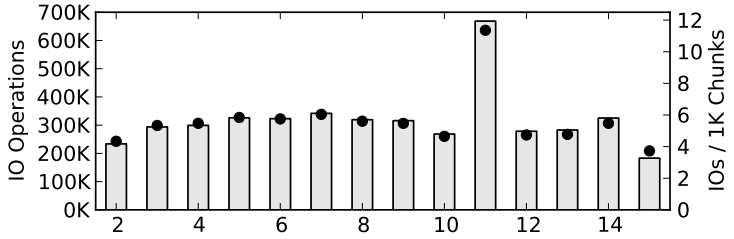


(b) Week 11.

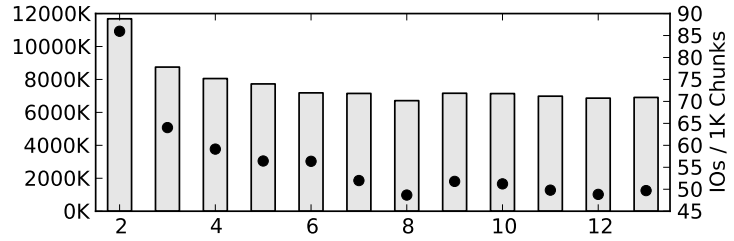


(c) Week 13.

Figure 35: Segment read operations in *UPB* data set.



(a) *UPB* data set.



(b) *JGU* data set.

Figure 36: IO operations per week using the best explored Exact Block-Based Extreme Binning configuration. The dots denote the IO operations normalized to 1,000 processed chunks. The scale of the normalized performance has been adapted in (b) so that the values are shown.

reduced by 10.3% (sampling rate $1/2^6$) resp. 22.4% (sampling rate $1/2^7$), the mispredictions out-weight the savings.

The Exact Sparse Indexing approach can avoid the disk bottleneck. This demonstrates that exact-variants of originally approximate deduplication approaches are possible.

11.2.5 *Results: Exact Block-Based Extreme Binning*

The Exact Block-based Extreme Binning approach shows the worst performance in both data sets. While the modification of Sparse Indexing as an exact approach for a block-based backup system is successful, the modification of Extreme Binning is not.

The best results for this approach have been achieved with the smallest explored block size of 256 KB, leading to a sampling rate of $1/2^5$ on average. The results for this configuration are shown in Figure 36a for the *UPB* data set and in Figure 36b for the JGU data set.

The approach features two levels of chunk prediction. In the full block duplicate level (it is a full file duplicate level in the original approach), all chunks in a block are marked as existing if the fingerprint of the full block is matching the full block fingerprint stored for the representative chunk fingerprint of the block. Of all successful predictions, 45.2% (*UPB*), resp. 44.7% (*JGU*) are based on the full block duplicate checks. This means if two blocks share the representative fingerprint, in more than 40% of the cases the blocks are identical compared to the first usage of the representative fingerprint. These predictions can be made without causing any IO operation.

Conceptually, this mechanism also seems to be prone to aging because the full block fingerprint for a representative fingerprint is – according to Bhagwat et al. [BELL09] –

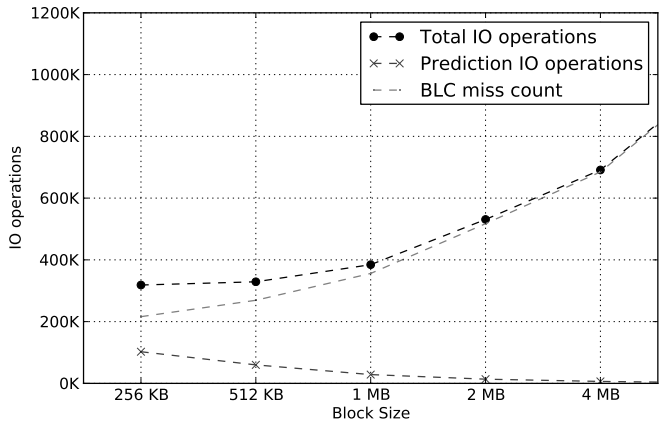


Figure 37: Comparison of different block sizes for the Exact Block-Based Extreme Binning approach in the *UPB* data set.

never updated. In the *UPB* data set, the ratio decreases from 45.3% to 45.1% just to increase again to 45.5% in the last week. In the *JGU* data set, the ratio decreases from 45.0% to 44.6%. Therefore, the aging thesis cannot be validated based on the available results.

The bin-level prediction is very expensive (in terms of IO) and not that successful. On average, 84.5% more chunks cannot be correctly predicted with the Extreme Binning variant than with Exact Sparse Indexing in the UPB data set. In the *JGU* data set, even 6.6 times more chunks are not predicted correctly. On the other hand, 3.6 (*UPB* and *JGU*) times more IO operations are performed to load bins instead of loading segments. This is the effect of calculating a representative fingerprint per dedupv1 block of 256 KB instead of much larger segments and of the simpler near-duplication detection approach.

Larger block sizes reduce the IO spent on the bin-level prediction, but also reduce the quality of the full block duplicate detection and the prediction quality in total. The effect of different block sizes on the total number of IO operations, the IO spend to fetch bins, and the IO operations due to chunk misses in the *UPB* data set is shown in Figure 37.

When considering an approximate block-based Extreme Binning variant, on average 14.5% (*UPB*), resp. 23.2% (*JGU*) more chunks are stored in each week, compared to an exact variant. While the IO requirements are significantly reduced, still on average 150,335 (*UPB*) resp. 591,023 (*JGU*) IO operations are issued.

The IO pattern of the simulated Exact Block-based Extreme Binning are highly random as visible in Figure 38 for week 8 of the *UPB* data set. The IO pattern for all weeks can be found in the appendix.

This is an artifact of the simulation that used hashing to place bin data on disk. However, this is not inherent to Extreme Binning in neither the original nor the exact variant. Other bin placements are possible, but since the performance results are always the worst of all compared approaches, this optimization has been skipped.

The results indicate that both block-based variants of Extreme Binning do not provide a good trade-off.

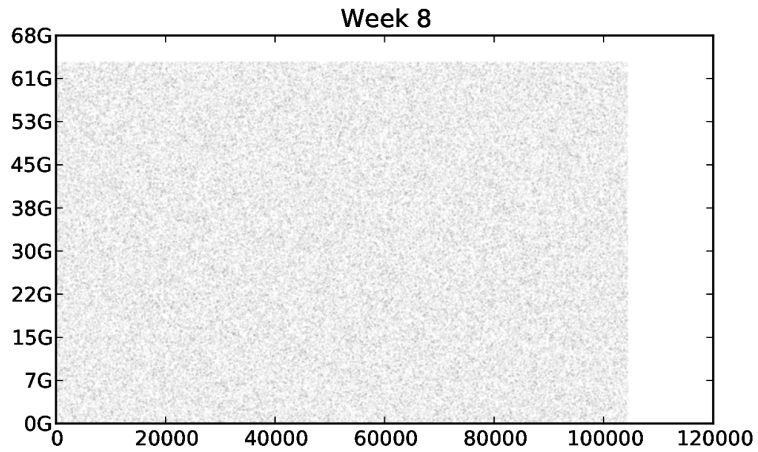


Figure 38: Bin read operations in *UPB* data set, Week 8.

11.3 EVALUATION: PROTOTYPE

In addition to the evaluation using trace-based simulations, the approach has also been implemented as a prototype and evaluated using it.

The Block Locality Cache has been integrated into the dedupv1 data deduplication system [MB10a]. A new field called "block hint" extends the chunk index entry. It is updated in the background as part of the garbage collection process.

In the garbage collection process, the usage counts of all chunks is updated based on recent block updates. Thus, it is efficient to also update the block hint field there. The data is not saved to disk immediately but delayed so that multiple updates to chunk index entries can be accumulated into a single IO operation. If the system crashes, the block hint field is restored in a log recovery step. As the usage counts of all fingerprints touched are updated, the additional maintenance of the block hint data does not cause any additional IO operations. It increases the data size of the chunk mapping data structure by 8 bytes per chunk.

The check of the cache data structure is in the critical path of the SCSI request. Up to 32 concurrent SCSI requests access the data structure in parallel. Therefore, fine-grained concurrency is necessary so that the BLC does not create a serialization point in the execution that would undo all performance improvements. Thus, a concurrent hash table is used to store all fingerprints that are currently cached. The hash table maps from the fingerprint to a set of all cached blocks that are currently providing the fingerprint. A spin lock protects the cached difference values. The LRU cache information about the blocks is stored 4-way associative by hashing the block.

The LevelDB key/value store [DG11], a LSM-Tree [OCGO96] implementation, is used to store block recipes. The index is configured with a internal block size of 32 KB and a LRU cache holding 4,096 data blocks.

The prototype implementation of the Block Locality Cache is evaluated using a single-node backup setup.

The deduplication node has an Intel Xeon CPU with 4 cores at 3.3 GHz and 16 GB RAM using Ubuntu 12.04 with a 3.2 Linux kernel with SCST performance patches. A software RAID-0 using 6 500 GB hard disks and an Intel 520 SSD with 120 GB RAM are directly attached to the node. The RAID-0 ensures maximal performance for the given hardware. A production system would certainly use a RAID-6 scheme using more disks. The RAID-0 and the SSD use the ext3 file system with the `noatime` option.

Another similar node is used as backup client. The client uses a RAID-0 of three Intel 520 Solid State Disks to store the backup data. This ensures that the backup client is able to read the backup data faster then the deduplication node can write it, so the backup client is not the throughput bottleneck. Both nodes are connected using a 10 GBit Ethernet network and use iSCSI for data transfer.

The backup data consists of two backup generations with 128 GB each. The data has been generated using the *UPB* data set using a "chunk-hash expansion approach". In this data generation approach, the redundancy and re-use pattern are simulated based on an existing trace data set. The data set is processed, chunk-by-chunk. For each base chunk, a block of data is generated. The size of the generated data block is identical to the size of the base chunk. The content is generated using a pseudo-random generator using a hash of the chunk fingerprint as seed. This ensures that two chunks with the same fingerprint generate a block of data of the same size and the same content. The data blocks are concatenated into a single file, simulating a large block-based backup file.

In this evaluation, the chunk index, block index, and container data are stored on the hard disk RAID. Only the operations log is stored on the SSD storage to achieve a low-latency commit. The system uses content-defined chunking with an expected chunk size of 8 KB, SHA-1 fingerprinting, and 4 MB containers.

Four dedupv1 filter chain configurations are evaluated:

BASELINE: Only the chunk index filter is executed so that most chunk lookups cause a disk access.

BLOOM FILTER (BF): A Bloom filter is checked before the chunk index filter is executed. The Bloom filter avoids most chunk index lookups for chunks that have not been stored before. It is configured with an expected false positive rate of 1%.

A Bloom filter is also used in the exact data deduplication systems presented by Zhu et al. [ZLP08] and at least some versions of the Venti system [QD02, RCP08, LH08].

BLC: The Block Locality Cache is used to avoid chunk index lookups to existing chunks. The block chunk cache is set to 256 with a difference cache size of 16. Only if the BLC is not avoid to predict a chunk lookup, the chunk index is queried.

BLC+BF: The Block Locality Cache and the Bloom Filter are combined. The Bloom filter first removes most requests for non-existing chunks, then the BLC is used to avoid most requests for existing chunks. Only if the Bloom filter and the BLC fail, the chunk is looked up in the chunk index.

Each configuration is executed 5 times. The results in Table 9 are presented with the confidence interval with a 95% confidence level.

The systems without the BLC have an average throughput of less than 30 MB/s in the second generation, which clearly shows the disk bottleneck. Even the extensive caching of already used chunks and the Bloom Filter is not able to avoid enough chunk index

Table 9: Throughput using HDD-based dedupv1 with different filter chain configurations (95% confidence interval).

	1. Backup Generation	2. Backup Generation
Baseline	15.3 MB/s (CI 15.0 - 15.6 MB/s)	17.0 MB/s (CI 16.2 - 17.8 MB/s)
BF	269.8 MB/s (CI 265.6 - 274.0 MB/s)	18.5 MB/s (CI 18.0 - 18.9 MB/s)
BLC	16.8 MB/s (CI 16.3 - 17.3 MB/s)	126.2 MB/s (CI 123.2 - 129.2 MB/s)
BLC+BF	267.0 MB/s (CI 266.1 - 267.9 MB/s)	366.2 MB/s (CI 361.4 - 371.0 MB/s)

lookups. The Bloom filter is able to overcome the disk bottleneck in the first generation because most of the chunks are new and most chunk index lookups can be avoided, but it has only a small effect in the second backup generation.

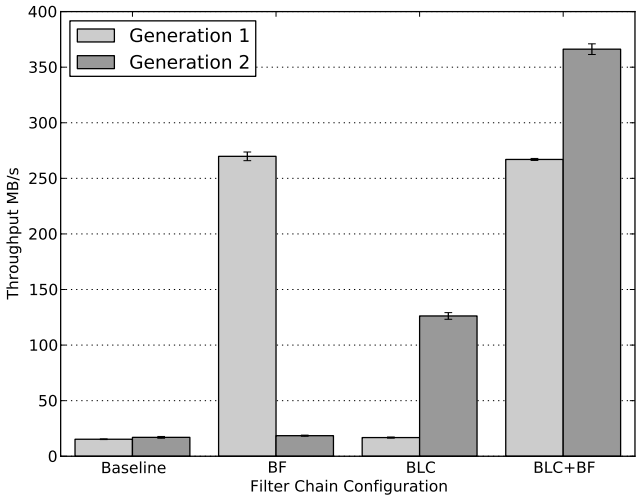


Figure 39: Throughput using HDD-based dedupv1 with different filter chain configurations (95% confidence interval).

With the Block Locality Cache, the system achieves 126.2 MB/s in the second data generation. Thus, the throughput is on average 6.6 times higher than in the second data generation using the baseline or BF-only configuration. The Block Locality Cache hit rate is 92.4% on all processed chunks and 99.2% on all redundant chunks on average. Per block fetch, on average 16.1 chunk index lookups could be avoided.

The system spends 91% of the write latency executing the filter chain. Within the filter chain, 15% of the time is spent in the BLC and even after avoiding the lookup of so many chunks, 85% of the time is spent for chunk index lookups. Of all chunk index lookups in the second generation, 89% are negative meaning that the chunk has not been stored before. On average, over 900 disk reads per second have been performed for chunk lookups. In the given setup using 6 hard disks, the chunk lookups consume most of the possible random IO operations per second.

That new chunks are sufficient to create a disk bottleneck even in the second generation explains why the combination of the BLC and the Bloom filter achieves the highest performance with over 260 MB/s in the first and 360 MB/s in the second generation. The disk IO operations for redundant chunks are avoided by the BLC and the disk IO operations for new chunks are avoided by the Bloom filter. Only mispredicted chunks and Bloom filter false positives actually cause chunk index lookups.

Another reason for the high performance if the BLC is combined with an approach that targets new chunks is that new chunks cause fetch operations that cannot be successful. Therefore, on average 24.4 chunk index lookups can be avoided per fetch operation in the second generation.

In the second generation, the Bloom filter handles 6.7% of all chunks requests. Additional 92.4% of all chunks are found by the BLC. The BLC hit rate is 99.1% because the BLC processes only existing chunks and a small fraction of false positive chunks that are lead through by the Bloom filter. Only 0.8% of all chunks reach the chunk index. On average 306 chunk index lookups are performed per second, which is a rate the disks can handle. The disk bottleneck is overcome.

The limitation in this evaluation setup is probably the usage of a single-ingest stream. Neither the CPU nor the hard disks are fully utilized. This indicates that even higher throughputs are possible if multiple backup streams are active concurrently.

The hardware setup used in this evaluation is roughly comparable with the setup used by Zhu et al. and Lillibridge et al. [ZLP08, LEB+09]. Zhu et al. used two dual-core 3 GHz CPUs with 8 RAM and a RAID-6 using 15 disk. They report a single-stream throughput of 110 MB/s in the first and 113 MB/s in later generations and a throughput of 139 MB/s and 217 MB/s using four parallel data streams. Lillibridge et al. report a throughput of 90 MB/s (single-stream) and 120 MB/s (four streams) using two dual-core 3 GHz CPUs with 8 GB RAM, and a 12-disk RAID-6. This shows that a HDD-based dedupv1 system is able to achieve a significantly higher throughput than existing approaches when the BLC is used in combination with a Bloom filter to filter out new chunks.

12

LIMITATIONS

This chapter discusses some limitations of the approaches presented in this part.

The approaches are designed to fit into the system architecture of the dedupv1 data deduplication system [MB10a]. They are not limited to this system environment, but are not applicable in all cases. Without specific knowledge of internal design decisions, it is impossible to claim that the approaches work for a specific alternative system.

An important assumption is that all compression approaches and the Block Locality Cache assume a fingerprinting-based data deduplication system. Furthermore, the main focus is a backup environment. All compression approaches, except the statistical prediction approach, work in other environments, too. E.g., Jin and Miller observed a significant fraction of zero-chunks in virtual machine disk images [JM09]. However, the file recipe overhead is only getting large for full backups with a high deduplication ratio. Recipe compression may not be important in non-backup systems. The Block Locality Cache uses workload locality properties only present in backup workloads.

Without an additional refinement, the Block Locality Cache and block and file recipe compression are mutually exclusive. The Block Locality Cache needs full chunk fingerprints in the recipes as these fingerprints are compared with the currently processed fingerprints. Without using full file fingerprints for the duplication detection the probabilities of hash collisions raise to an unacceptable level.

However, the BLC and recipe compression can be made compatible. The evaluation of the BLC showed that only the block recipes of the latest backup runs are fetched by the Block Locality Cache. Old block recipes can be compressed without any effect on the prediction quality of the BLC. Therefore, it is possible to store the block recipes with the code word and additionally with the full fingerprints initially. After a specified time, e.g., two weeks, the block recipes are rewritten and the full fingerprints are removed.

Also, all compression approaches, except the zero-chunk suppression, assume the usage of a full chunk index. If only a subset of the chunks is stored, it is – in general – not possible to use the page-based approach or to store a prediction fingerprint.

The Block Locality Cache is able to work with a sparse chunk index. However, without a full chunk index, the resulting deduplication system can only provide an approximate deduplication. This is an example for the independence of the prediction approach and the exact/approximate property similar to the approach presented by Guo et al. [GE11], which is an approximate container caching approach.

13

CONTRIBUTION

In this part, two contributions with respect to the management and usage of block and file recipes in data deduplication systems have been presented.

First, a combination of compression approaches for block and file recipes in deduplication systems have been proposed. Since file recipes can be responsible for a significant fraction of the physical disk capacity of deduplication systems with very high deduplication ratios, these results enable significant overall savings.

With limited storage and memory overhead, these methods allow shrinking the recipes in all data sets by up to 93%. All methods are practical, efficient, and scalable. They do not cause additional IO operations and their storage overhead is negligible when compared to the storage saved by applying the compression.

Besides, the block and file recipes have been used to devise a novel approach to overcome the chunk lookup disk bottleneck. The Block Locality Cache uses the locality properties of the existing block index data structure to predict blocks that are likely related to the currently written data. It is shown that this approach allows to capture the locality of the latest backup run and is, therefore, likely to be less prone to aging.

A trace-based simulation and a prototype evaluation show that this approach is able to overcome the disk bottleneck. Significantly less IO operations are necessary for the same data set than for the explored alternative approaches.

Part IV

DATA DEDUPLICATION IN HPC STORAGE SYSTEMS

14

INTRODUCTION

The previous research presented in this thesis is focused on contributing to the architecture and design of deduplication systems. The first part by presenting an approach how to scale deduplication systems and in the second part by presenting novel approaches improve the index data structures of deduplication systems.

The key usage scenario in these areas is deduplication as a tool for data backup. Besides backup, data deduplication has also been applied to virtual machine storage and WAN replication [ZLP08, SHWH12, ELW+07, CAVL09].

This part shows that the core techniques of data deduplication are also applicable to a new application domain and are able to find a significant amount of redundancy in storage systems for high performance computing (HPC) clusters.. It investigates the potential for data deduplication in HPC storage and answers the question of how much redundant data is stored in HPC storage systems, which could be found and removed by data deduplication techniques.

The research presented in this part has previously been published as "A Study on Data Deduplication in HPC Storage Systems" [MKB+12]. The work would not have been possible without the support of the staff and administrators of the involved data centers. The presentation in this thesis differs from the published version, as new data sets have been added. These data sets have not been available at the time of the original publication.

As HPC systems are used for challenging scientific computing problems such as climate research. Therefore, HPC systems and HPC storage have been a hot research topic for decades. Most research deals with either the performance aspect [JSPW+11, WUA+08] or the manageability aspect [MRHBS06, PGLP07]. Storage savings techniques have not been an important issue in the past.

However, this might and probably should change in the future. The increasingly higher storage requirements for future exascale computing demand storage systems out of hundreds of thousands of disk [Bra11]. Such a high number of disks creates new obstacles for the operation and management of such systems. On the other hand, the gap between the performance of computation and the performance of storage will likely increase further in the future. Thus, storage saving techniques that can trade computation for significant storage savings might be important in the future. While the usage of data compression has been explored for HPC storage systems, it is not clear that data duplication can find sufficient redundancies in HPC storage data to make its usage worthwhile. This lack is closed by this study on the potential for data deduplication in HPC storage systems.

The scientific computing performed in HPC centers generates massive amounts of data and produces new data that needs to be stored. In general, this has two reasons:

CHECKPOINTS: Checkpoints are a common mechanism of HPC applications to tolerate component failures. At certain intervals, the computation is stopped and the state of the application is written to the storage system. If a component fails, and the computation has to be restarted, it can resume using the state of the most recent checkpoint. According to Bent et al., the checkpoint IO is a major workload on HPC storage systems [BGG+09].

EXPERIMENTAL AND SIMULATION DATA: In addition to checkpoint data and other temporary job data, HPC storage systems also store the input and the output data of the jobs. Monti et al. state that the job output data set sizes have grown exponentially and that user jobs may even generate terabytes of output data per user job in some super computing centers [MBV08].

While probably every scientific group and every HPC data center has slightly different workflows, there are some general statements that are assumed to be true in most cases.

The data is often stored on parallel file systems for online access or on tape for archival access. While it would also be interesting to study the deduplication of archived HPC data, the focus here is on the online accessible data stored in parallel file systems. In such a file system, the data is usually striped across a large number of storage nodes. Examples for such file systems are Lustre [Bra02], IBM's GPFS [SH02], PVFS [CLI+00] (now Orange File System [RLC+08]), and Ceph [WBM+06].

Even, the online storage is managed using several hierarchies or even multiple file systems for each purpose. The scratch space of an HPC file system is used for temporary storage. The performance is of highest importance for the scratch space of a file system.

This is different from home and project directories. Performance is still essential, but it is also important that the data is stored reliable. While the data stored in a scratch space usually lasts only a short time, the data in home and project directories might be stored there for the duration of a research project.

A usual workflow of a HPC application is the following:

1. Users store applications and input data in their home or project directories. Input data might need to be copied into an application-specific "run" directory.

2. Applications produce large amounts of intermediate data and store it in the scratch space.

3. Post-processing extracts relevant data to answer the scientific question. This reduces data that is then stored again in the project directories.

4. Input and output data may be moved (or copied) to a tape archive for backup and long-term archival purposes.

The typical workflows of scientists and developers at the Los Alamos National Laboratory have been documented by Strong et al. [SJPW+11].

It is common for the storage systems in HPC data centers to provide an online capacity of several PB. Therefore, the storage is an important cost factor in the installation of such a data center.

If the data has significant redundancy that can be eliminated by using data deduplication, hundreds of TB of disk capacity could be saved.

The study presented here is the first large-scale study of data deduplication in HPC storage systems. It analyses the deduplication potential in multiple HPC data centers. The data centers are of various sizes and focus on different application domains.

In total over 3 PB of data have been analyzed. It is therefore the largest in-depth analysis of deduplication potential to date. In most data centers over 20% of the data can be removed by data deduplication. In contrast to the intuition, most of the redundancies is not caused by file copies, but it seems to be inherent to the applications and the data.

The expectation is that the findings presented in this part will motivate HPC storage researchers and developers to incorporate data deduplication techniques into the development of future generation HPC storage systems.

Related work is discussed in Chapter 15. The methodology of this study including its limitations are described in Chapter 16. In Chapter 17, basic statistics about the investigated file system data sets are presented. It gives a first look at the file systems and its basic statistics about the distribution of file sizes and for file extensions. The deduplication results are presented in Chapter 18. Chapter 19 discusses the results and gives first hints how, based on the results presented, data deduplication might be integrated into HPC storage systems. Finally, the contributions presented in this part are summarized in Chapter 20.

15

RELATED WORK

While this study is the first focusing on data deduplication potential in HPC storage, several similar studies exists in other contexts.

One of the first studies on data deduplication potential in file systems was presented by Policroniades and Pratt [PP04]. They presented the deduplication results for static chunking and content-defined chunking on various data sets including web data, personal files, research group files, scratch directories, and software distributions.

A study of 450 GB personal computer storage for whole file duplicates, static chunking, and content-defined chunking has been published by Mandagere et al. [MZSU08].

In 2009, Meister and Brinkmann presented a study using the *UPB* data set [MB09]. The study focused on the temporal redundancy that is necessary for backup systems and provided an in-depth analysis of some file formats to explain special pattern and effects.

Meyer and Bolosky present a study based on an engineering desktop workload at Microsoft [MB12]. They found that the removal of full file duplicates is highly effective for their workload.

Wallace et al. study backup workloads based on automatically collected basic statistics from over 10,000 EMC Data Domain installations and an in-depth analysis of other collected data sets with a total size of 700 TB [WDQ+12].

Jayaram et al. analyzed the similarity between virtual machine images in an Infrastructure-as-a-Service (IaaS) cloud environment [JPZ+11].

The data and the workloads generating the data is inherently different in HPC systems than in all the environments studied before.

Only in some situations such as if virtual machine images are used in HPC environments as proposed, e.g., by Keller, Meister and Brinkmann [KMB+11], some existing knowledge about deduplication potential might be transferred. In general, the results cannot be transferred. Thus, a new study focused on HPC has been necessary to gain knowledge about the deduplication potential in HPC storage systems.

Besides a discussion of the deduplication properties, statistics about the file metadata on the file systems, e.g. the file types and file sizes, are presented. Similar statistics have been presented, e.g. by Ramakrishnan in 1992 [RBK92] and by Dayal in 2008 [Day08]. Especially the latter is interesting because it provides information about the directory sizes, timestamp information, and sparse files that have not been investigate due to the different focus in this research. Adams et al. presented a related study about long-term scientific archival data and the workload pattern [ASM12]. Outside the domain of HPC storage systems, several studies have been published on the distribution of file system metadata, file size, and file formats [DB99, Vog99, ABDL07].

Other fields of related studies are the IO characteristics and IO pattern of HPC applications. These characteristics have been evaluated in several studies because the IO characteristics have a direct impact on the design of HPC system systems [NKP+96, CLI+00, CHA+11, Roto7, FZJZ08, LPG+11].

Most related to this study is Kiswany et al.'s work on a checkpoint storage system focused on desktop grids, which uses data deduplication [KRVG08]. Their analysis is focused on the applications BMS [Aga05] and BLAST [AGM+90] and is limited to a small data set.

Another approach to reduce the storage requirements is data compression. In contrast to deduplication, compression reduces redundancy within a very narrow, local scope by increasing entropy. Compression can be used in conjunction with deduplication, e.g., compressing the chunk data after the deduplication detection. Compression has been evaluated for HPC storage, especially for checkpoint data, in various publications [SG07, IAFB12].

16

METHODOLOGY

The FS-C tool suite, which has been described in Section 2.4, has been used to analyze the file system data of HPC storage systems. This study focuses on content-defined chunking with an expected chunk size of 8 KB. This is the default chunk size in enterprise-backup data deduplication systems and allows us to estimate the deduplication ratios and compare them with the corresponding values in that field.

To assess the impact of the chunking algorithm, some data sets have also been traced for static chunking. Larger chunk sizes are simulated by processing the trace file and form combined chunk fingerprints that resemble larger chunk sizes.

All chunks are fingerprinted using SHA-1. To reduce storage requirements, only the first 10 bytes of the 160 bit SHA-1 fingerprint are stored. The potential impact of the reduced fingerprint size is discussed in Section 16.2.

16.1 DATA SETS

The deduplication analysis has been performed in various data centers to increase the generality of the results. These centers have different sizes and focus on different application domains. If possible, the available data is split up into several data sets that focus on a phase in the HPC storage life cycle.

The data sets have been collected by the local support staff of the data centers using the FS-C tool suite under instruction of the author.

This section presents the different data sets and provides some background information about the data centers the data has been gathered from.

RENCI: RENCI is a collaborative institute that provides HPC computing infrastructure to several universities in North Carolina. The file systems at RENCI have a size of 140 TB in total. The trace file contributed by RENCI contains fingerprints of 11 TB of data from a single file system. While this is only a subset of the overall data, it is all data from one file system and therefore a valid trace file to measure the potential data deduplication savings in that instance.

DKRZ: The largest contributor of trace data in this study is DKRZ, the German Climate Computing Center. It is dedicated to earth science with the goal to provide high performance computing platforms, data management, and service for climate science.

In total, the system contains around 1,000 user home directories and 330 project directories. The total work file system contains around 3.25 PB of data with additional 0.5 PB on a scratch file system. The complete file system has been traced. The

complete data has been split up into different data sets to reflect potential internal differences by organization of field of research. The *DKRZ-B* data sets mainly consist of projects supported by the German Ministry for Education and Research (BMBF). The *DKRZ-M* data sets are scientific project data similar to *DKRZ-B* but funded by the Max Planck Society. The *DKRZ-C* data sets are projects supported by Climate Service Center (CSC) and the coastal research organization HZG. The *DKRZ-I* data sets denote a group of consortial projects. The *DKRZ-K* data set consists of various internal projects including test data, visualization, and guest account data. The *DKRZ-A* data set includes projects from the Alfred Wegener Institute for Polar and Marine Research (DKRZ-A). Finally, the *DKRZ-U* data sets are built from projects of the University of Hamburg.

RWTH: The *RWTH* data set has been generated from the HPC systems at the University of Aachen (RWTH). The major application domains are chemistry and mechanical engineering. The Lustre file system of the RWTH HPC group has been traced to completion. The data set has a size of 53.8 TB.

BSC: The *BSC* data sets come from the earth science department of the Barcelona Supercomputing Center (BSC). The scientific computations of this department are mainly weather and pollution forecasts.

The different file systems build some kind of hierarchy (from the data life cycle point of view). The *BSC-PRO* data set (used by around 20 users, 9.1 TB of analyzed data) and the *BSC-SCRA* data set (used by only 3 users, 3.0 TB analyzed data) are the file systems where the data currently being used is placed. The data sets *BSC-BD* and *BSC-MOD* (both used by around 30 users, 11.2 TB, resp. 20.1 TB) are file systems where data is kept until the researcher decides that it will not be needed anytime soon. Then it is moved to a hierarchical storage management (HSM) system. Thus, these file systems have a moderate to low access rate. There is no policy implemented to move data from one level to another; it depends on the user methodology and hints set by the system administrator.

16.2 BIASES, SOURCES OF ERROR, AND LIMITATIONS

This section discusses some sources of biases and errors as well as limitations of the analysis and the data sets.

The massive amount of data produced in HPC centers makes it impossible to generalize the evaluation results without taking possible biases into account.

While more than 3 PB of data has been traced and analyzed, this is not the complete storage available at these data sets. The trace runs of the file systems at the DKRZ, at RWTH, and of one of the file systems at RENCI ran until completion. However, it was not possible to trace the complete file system of the BSC data set, which makes the data set incomplete. Harnik et al. have shown that any naive sampling approach leads to bad estimations in the worst case [HMN+12].

The sample of data centers over-represents European data centers; only a single North American center and no center from Asia and Africa has been analyzed. Additionally, the limited number of data centers might introduce a bias and limit the generalization of the presented results.

Due to limited data access, it was not possible to check for what Dayan calls "negative and positive overhead" [Dayo8]. Positive overhead means that a file occupies more block storage than required by its file size. The reason could be that a small file may use a full block in the storage systems and that more blocks are allocated than currently used, e.g., by the Linux `fallocate()` function. Negative overhead means that a file uses less capacity in blocks on the file system than indicated by the file size. According to [Dayo8], this may be caused by an optimization that stores the contents of small files in their corresponding inodes or (probably more important) by so-called "sparse files" – files that contain holes in the middle of the file because they are just partly written.

Symbolic links are detected and excluded from the trace. However, hard file system links are not detected. Hard linked files would be counted as full file duplicates resulting in an overestimation of the redundancy. Nevertheless, as Section 18.4 shows, in all cases but one, the percentage of file replication is below 5% and it would be unlikely that a large percentage of this full-file replication comes from hard links.

Each file system walk has taken days or even weeks. Therefore, changes to the data set are only partially reflected. Deletions after the file has been chunked are not reflected. New files are only included if the directory scanner has not already processed the corresponding directory. It can also happen that a user moves already indexed files into a directory that will be indexed in the future, which results in an overestimation of duplicates. Again, this would be reflected in the full-file duplication ratio, which is in all but one case only a fraction of the overall deduplication ratio.

Due to the file system walk, it is impossible to relate deduplication and file modification patterns. Since every data set is only processed once, it is not possible to analyze the temporal redundancy of files.

The FS-C tool only collects the hashes of the file names. Therefore, it is not possible to analyze the properties of the data sets at a directory or user level. Since file metadata like owner, groups, and timestamps are not collected, it is not possible to relate deduplication and file age.

Another source of potential errors is caused by the reduction of the stored fingerprints to 10 bytes. This increases the likelihood of accidental hash collisions between fingerprints. Nevertheless, the birthday paradox equation gives us an upper bound of 0.7% to have one collision in a PB data set. However, the bias even if such a collision occurs is still sufficiently low.

17

FILE STATISTICS

This chapter presents statistical information of the file systems, which is also collected by FS-C. It gives a first look at the file systems and its basic statistics about the distribution of file sizes and for file extensions.

The last published study presenting similar information at a large scale and from multiple data centers has been from 2008 and even this study covers less than one PB in total [Dayo8]. While no deduplication ratios are stated in this chapter, the infrequency of such studies on these basic information makes this data interesting on its own right.

17.1 FILE SIZE STATISTICS

An overview of the data sets and their size distribution is given in Table 10. For each data set, it contains the number of files in the trace, the total capacity, the mean file size, and the median file size.

In addition, Table 11 contains the 10%, 25%, 50%, 75%, and 90% percentile file sizes in each data set. The 50% percentile, i.e., the median, for the file counts means that 50% of the files are smaller than the percentile. Table 12 shows the cumulative file sizes. For the capacity, the same median denotes the file size for which 50% of the capacity is occupied by smaller files and 50% by larger files.

Table 10: File Size Statistics from different HPC data sets.

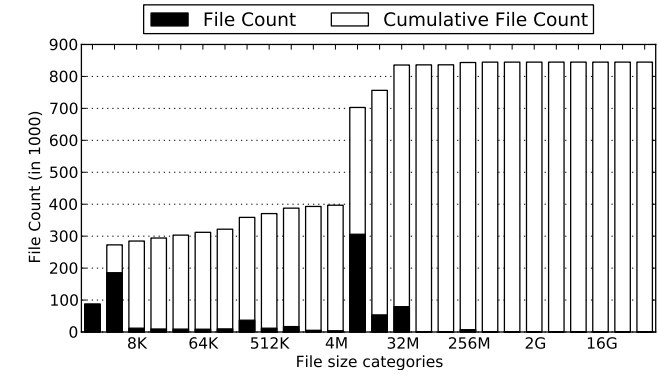
Data set	File Count	Capacity	Mean File Size	Median File Size
BSC-BD	865,063	11.2 TB	13.6 MB	11.1 MB
BSC-MOD	1,113,760	20.1 TB	18.9 MB	84.4 KB
BSC-PRO	3,987,744	9.1 TB	2.4 MB	1.0 KB
BSC-SCRA	76,957	3.0 TB	41.5 MB	9.7 KB
DKRZ-A	343,118	27.0 TB	82.4 MB	3.3 MB
DKRZ-B1	1,790,450	114.5 TB	67.1 MB	4.0 MB
DKRZ-B2	2,273,444	109.1 TB	50.3 MB	4.3 MB
DKRZ-B3	3,424,270	126.5 TB	38.7 MB	6.0 KB
DKRZ-B4	21,175,615	68.3 TB	3.4 MB	256.1 KB
DKRZ-B5	650,372	75.3 TB	121.4 MB	20.8 KB
DKRZ-B6	1,340,788	47.9 TB	37.5 MB	2.0 MB
DKRZ-B7	485,209	65.2 TB	140.9 MB	1.1 MB
DKRZ-B8	3,445,787	176.6 TB	53.7 MB	0.2 KB
DKRZ-C	6,712,804	122.5 TB	19.1 MB	534.7 KB
DKRZ-K	2,245,564	42.9 TB	20.0 MB	9.2 KB
DKRZ-M1	2,947,453	134.5 TB	47.9 MB	76.3 KB
DKRZ-M2	1,715,341	116.8 TB	71.4 MB	600.6 KB
DKRZ-M3	1,888,296	80.5 TB	44.7 MB	228.1 KB
DKRZ-M4	2,414,480	120.2 TB	55.2 MB	24.3 KB
DKRZ-M5	6,408,247	159.8 TB	26.1 MB	697.8 KB
DKRZ-I1	8,672,428	1040.0 TB	125.7 MB	5.3 MB
DKRZ-I2	973,132	511.0 TB	550.6 MB	50.0 KB
DKRZ-U1	26,876,598	124.7 TB	4.9 MB	26.5 KB
DKRZ-U2	12,632,061	152.8 TB	12.7 MB	14.1 KB
RENCI	1,024,772	11.1 TB	11.4 MB	124.3 KB
RWTH	1,804,411	53.8 TB	31.3 MB	1.1 MB

Table 11: File Size Percentiles from different HPC data sets.

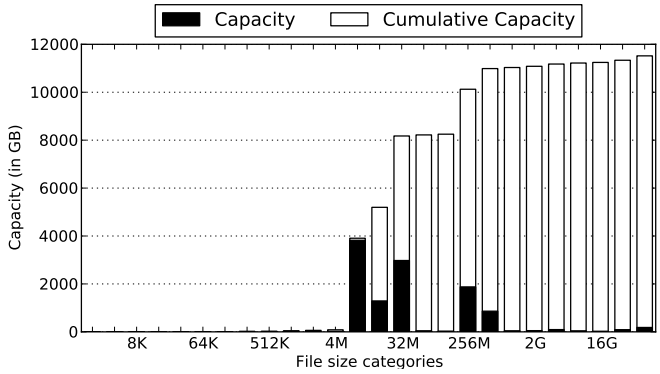
Data set	10%	25%	50%	75%	90%
BSC-BD	3.6 KB	7.6 KB	11.1 MB	13.3 MB	33.8 MB
BSC-MOD	8.1 KB	64.7 KB	84.4 KB	126.4 KB	5.6 MB
BSC-PRO	0.0 KB	0.4 KB	1.0 KB	49.0 KB	292.3 KB
BSC-SCRA	0.0 KB	0.3 KB	9.7 KB	1.1 MB	139.3 MB
DKRZ-A	2.6 KB	26.4 KB	3.3 MB	31.1 MB	76.0 MB
DKRZ-B1	2.8 KB	53.2 KB	4.0 MB	24.3 MB	95.4 MB
DKRZ-B2	4.5 KB	114.0 KB	4.3 MB	12.8 MB	54.8 MB
DKRZ-B3	0.5 KB	1.9 KB	6.0 KB	35.4 KB	1.3 MB
DKRZ-B4	3.0 KB	118.8 KB	256.1 KB	256.1 KB	256.1 KB
DKRZ-B5	1.2 KB	4.5 KB	20.8 KB	1.3 MB	74.4 MB
DKRZ-B6	3.1 KB	36.5 KB	2.0 MB	10.5 MB	29.7 MB
DKRZ-B7	7.2 KB	112.4 KB	1.1 MB	18.5 MB	165.1 MB
DKRZ-B8	0.0 KB	0.0 KB	0.2 KB	930.0 KB	25.3 MB
DKRZ-C	2.1 KB	19.8 KB	534,8 KB	3,2 KB	9.1 MB
DKRZ-K	0.5 KB	2.1 KB	9.2 KB	180.2 KB	982.8 KB
DKRZ-M1	1.0 KB	6.8 KB	76.3 KB	2.4 MB	38.1 MB
DKRZ-M2	2.8 KB	8.8 KB	600.6 KB	5.3 MB	68.3 MB
DKRZ-M3	2.4 KB	22.5 KB	228.1 KB	6.3 MB	31.6 MB
DKRZ-M4	0.1 KB	1.0 KB	24.3 KB	2.0 MB	76.3 MB
DKRZ-M5	1.3 KB	27.6 KB	697.8 KB	3.2 MB	43.5 MB
DKRZ-I1	0.1 MB	867.4 KB	5.3 MB	32.1 MB	221.7 MB
DKRZ-I2	1.8 KB	9.2 KB	50.0 KB	1.1 MB	122.1 MB
DKRZ-U1	0.2 KB	0.2 KB	26.4 KB	211,0 KB	554.3 KB
DKRZ-U2	0.2 KB	0.2 KB	14.1 KB	174.7 KB	1.3 MB
RENCI	4.3 KB	37.2 KB	124.3 KB	752.5 KB	3.7 MB
RWTH	1.0 MB	7.7 KB	1.1 MB	5.6 MB	8.0 MB

Table 12: Cumulative File Size Percentiles from different HPC data sets.

Data set	10%	25%	50%	75%	90%
BSC-BD	12.1 MB	13.3 MB	33.8 MB	257.4 MB	692.6 MB
BSC-MOD	55.1 MB	732.8 MB	2.6 GB	7.0 GB	27.3 GB
BSC-PRO	11.1 MB	206.0 MB	3.2 GB	29.6 GB	69.5 GB
BSC-SCRA	133.8 MB	156.8 MB	356.5 MB	4.1 GB	27.3 GB
DKRZ-A	42.5 MB	444.8 MB	1.0 GB	6.4 GB	22.6 GB
DKRZ-B1	53.5 MB	221.2 MB	1.5 GB	4.4 GB	24.2 GB
DKRZ-B2	40.9 MB	144.2 MB	3.4 GB	32.5 GB	85.7 GB
DKRZ-B3	376.9 MB	1.2 GB	4.6 GB	22.9 GB	53.8 GB
DKRZ-B4	80.9 MB	755.1 MB	4.6 GB	23.3 GB	117.1 GB
DKRZ-B5	330.3 MB	1.2 GB	5.2 GB	17.4 GB	23.6 GB
DKRZ-B6	31.0 MB	265.7 MB	1.9 GB	13.4 GB	39.9 GB
DKRZ-B7	279.6 MB	749.1 MB	2.2 GB	6.4 GB	17.6 GB
DKRZ-B8	174.7 MB	721.8 MB	2.0 GB	4.0 GB	14.5 GB
DKRZ-C	306,4 MB	8.2 GB	277.9 GB	2.4 TB	7.5 TB
DKRZ-K	465.2 MB	4.2 GB	45.9 GB	94.8 GB	426.0 GB
DKRZ-M1	66.8 MB	351.9 M	2.4 GB	6.7 GB	44.3 GB
DKRZ-M2	125.5 MB	554.0 M	1.9 GB	4.1 GB	30.4 GB
DKRZ-M3	81.9 MB	2.6 GB	28.8 GB	930.3 GB	5.0 TB
DKRZ-M4	15.5 MB	167.3 MB	5.4 GB	236.0 GB	7.3 TB
DKRZ-M5	233.2 MB	8.3 GB	570.0 GB	2.8 TB	16.1 TB
DKRZ-I1	19.0 GB	446.1 GB	5.4 TB	29.7 TB	135.5 TB
DKRZ-I2	49.0 MB	835.4 MB	7.0 GB	74.5 GB	4.1,TB
DKRZ-U1	456.2 MB	1.2 GB	22.6 GB	526.3 GB	1.8 TB
DKRZ-U2	195.4 MB	546.4 MB	4.6 GB	175.2 GB	792.1 GB
RENCI	37.8 MB	103.6 MB	254.3 MB	3.7 GB	37.7 GB
RWTH	43.4 MB	771.0 MB	18.0 GB	96.0 GB	512.0 GB



(a) File Size Count.



(b) File Size Capacity.

Figure 40: File statistics for the *BSC-BD* dataset.

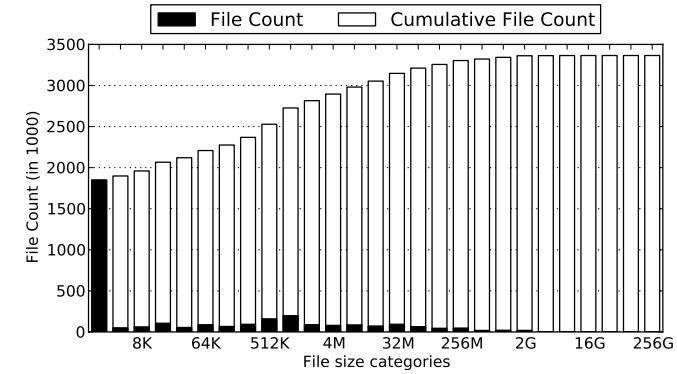
A pattern present in all data sets is that the file sizes are highly skewed. Most files are very small, but the minority of very large files occupies most of the storage capacity: 90% of these files occupy less than 10% of the storage space, in some cases even less than 1%.

For example, a fourth of the files in the *BSC-BD* data sets is less than 4 KB. Even with this large fraction of the overall file count, the capacity of these files is not relevant. Another third of the files has a size between 4 and 8 MB, where also a third of the total capacities lies. Figure 40 shows this information for the *BSC-BD* data set graphically. Figure 40a shows the file count distribution for the different file size categories. Figure 40b illustrates the distribution of the storage capacity to the file sizes. The corresponding figures for all data sets are available in the appendix of this thesis.

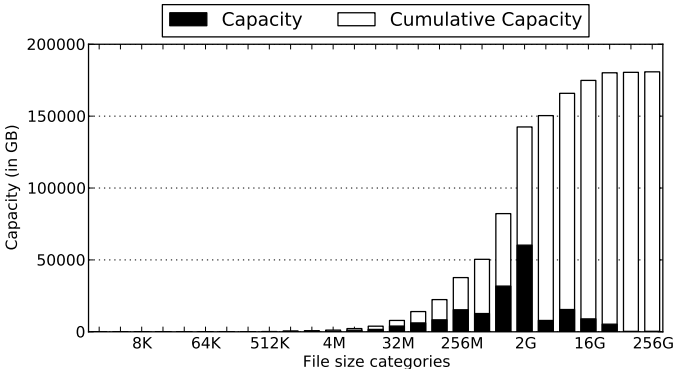
In these figures and in the following text, the files are grouped into bins, called file size categories. These are exponentially increasing ranges of file sizes, e.g., the file size category of 16 KB contains all files between 16 KB and 32 KB.

The *DKRZ-B8* data lies on the extreme end of this skew with more than 50% of the files being less than 200 bytes and 20% of files being empty (see Figure 41).

Files smaller than 4 KB are the most common file size category in all data sets except *BSC-BD*, *BSC-MOD*, *DKRZ-B1*, *DKRZ-B2*, *DKRZ-B4*, *DKRZ-B6*, *DKRZ-B7*, *DKRZ-I1*, and

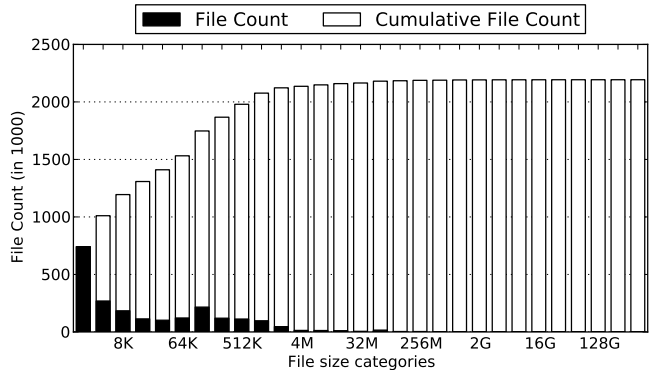


(a) File Size Count.

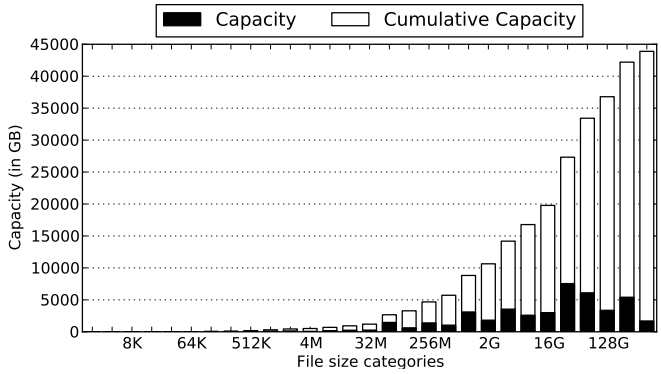


(b) File Size Capacity.

Figure 41: File statistics for the *DKRZ-B8* dataset.



(a) File Size Count.



(b) File Size Capacity.

Figure 42: File statistics for the *DKRZ-K* dataset.

RENCI. However, even within this list of exceptions in more than half of these data sets, the smallest file size category is a close second.

In most data sets, the median file size is less than 256 KB. In all data sets, the median is less than 6 MB.

On the other end, files larger than hundreds of GB or TB in size are also common. In the *DKRZ-K* (Figure 42) data set, for example, 10% of the total capacity is occupied by files larger than 426 GB.

17.2 FILE EXTENSION STATISTICS

The usage distribution of file extensions in the data sets is interesting from the viewpoint of data deduplication.

File extensions are an indication for the file formats of the files. However, file extensions in HPC systems are not as useful to categorize the data than in storage systems where files are more often created for human-consumption.

Figure 43 shows the distribution of file extensions according to the capacity of the files in relation to the total data set size. All file types that occupy less than 3% of the total capacity are aggregated into `Oth.`. The file type `N/A` indicates files that have no file extension.

The extensions `.nc` and `.ncf`, describing the NetCDF file format [RDE+11], are aggregated into one file type because the file format is the same. A large number of files have a suffix that only consists of digits such as `.000` or `.029`. These extensions are aggregated into a file type called numeric or `000`. Numeric extensions may represent different properties, like the ID of the task writing a checkpointing file or the iteration of a simulation run.

In most data sets, the `.nc/.ncf` file extension is a very popular file format. This format is the most used one in all *BSC* data sets. Only files where no file type was detected use more space. In all *DKRZ* data sets except *DKRZ-B2* and *DKRZ-M2*, NetCDF files use most of the capacity. In the *DKRZ-B3* data set, even more than 80% of the capacity and more than 70% of the files are NetCDF files. The NetCDF file format is within the top 16 of the file types in all data sets except *RWTH*.

The archive file format tar is the most prominent format in *DRKZ-B2*, and *DKRZ-C*. It is also prominent in the *DKRZ-A*, *DKRZ-B4*, *DKRZ-B6*, *DKRZ-M2*, and *DRKZ-I1*. In the HPC context, the `.tar` format is often used to prepare the files of a project to be migrated (or replicated) to a long-term archival system. The high amount of capacity this file format occupies on some data sets indicates that the data may have been copied to an archive, but the tar files have not been deleted.

The `.grb` extension indicates files that are in the GRIB file format [GRI94], which is a prominent data format for meteorology. This file format is used in some data sets.

In direct comparison with the data from *BSC* and *DKRZ*, the types are different at the *RWTH* and *RENCI*. At the *RWTH* data set, the HDF5 file format (`.h5`) [HDF11] occupies about half of the available space. *RENCI* data set is the only data set where compressed data plays a major role on the file system; about 25% of the data is stored in `.gz` files.

Figure 43 indicates that each project has its prevalent file type for scientific data (`.nc`, `.grb`, `.cdf`, `.h5`). Often only one of these has a widespread usage within a dataset. In all data sets, except *RENCI*, one of these file formats is in widespread usage.

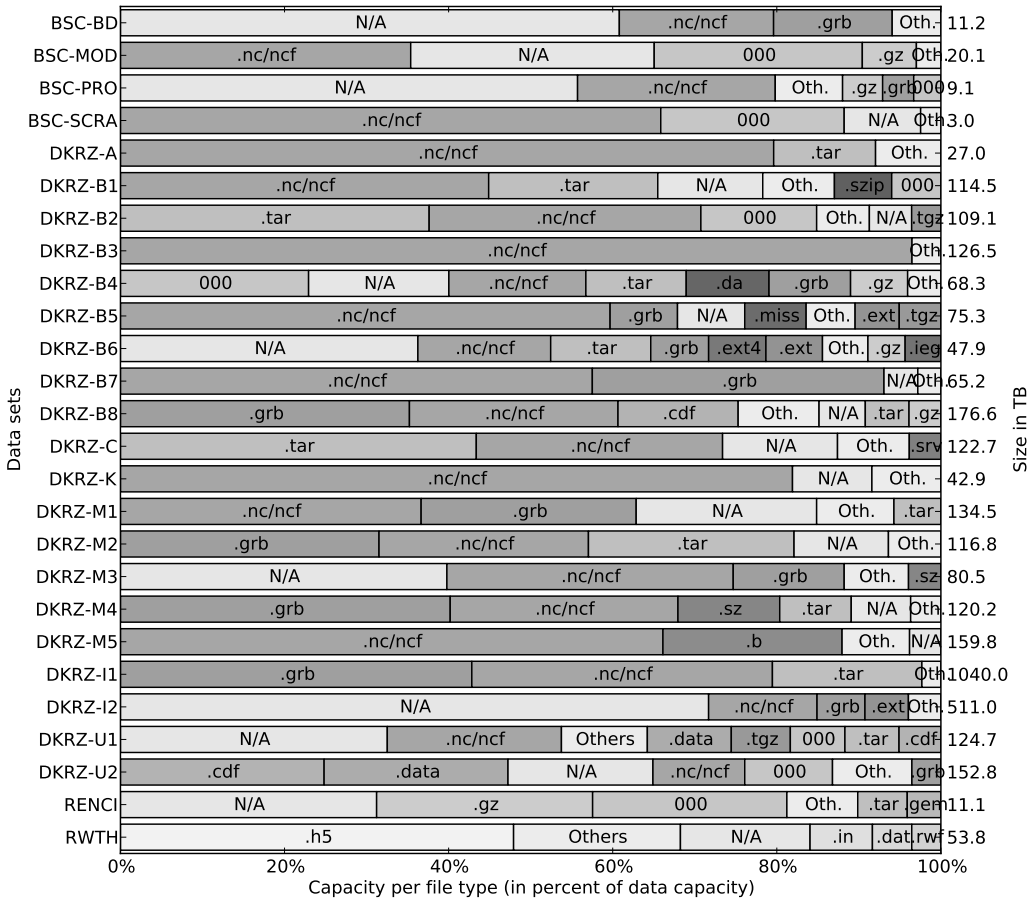


Figure 43: File type distribution in all data sets.

In fix data sets, this unknown type constitutes the biggest fraction of all file types. For example, the *RENCI* data set consists of 21.8% of unknown types. As nothing is known about the format of these files, this limits the insight that can be gained by relating deduplication ratio to the file extensions.

Similar to the file without a file extension are the files with a numeric extension. These files may come from different file formats, and these files cannot be inspected in more detail. However, files with only a numeric suffix are very common as shown in Figure 43. It is even the "file extension" that occupies most space in the *DKRZ-B4* data set.

18

DEDUPLICATION RESULTS

This chapter presents the main results of this part. It answers the question on the ratio of the storage capacity that could be reduced if data deduplication techniques would be applied to HPC storage systems.

After presenting an overview of the results, later sections will dig deeper into specific observations and the findings that have been deduced from the observations.

Figure 44 shows the deduplication ratios for all HPC data sets using content-defined chunking with an expected chunk size of 8 KB. Figure 45 shows a histogram of the deduplication ratios.

The deduplication ratios have been determined using the Hadoop-based method or Harnik's estimation method. Both methods have been introduced in Section 2.4. All deduplication ratios have a bounded error of at most 1% with a probability of 99.9%.

With a few exceptions, the ratios are located within a surprisingly small range. Most data sets have a deduplication ratio between 15% and 30%.

An outlier with less deduplication potential is *BSC-BD* with only 7%. On the other hand, *DKRZ-B3* has a deduplication potential of 74.4%. A consultation with the responsible scientists revealed that the project of the *DKRZ-B3* data set has been recently finished when the file system walk has been performed. Users have been in the process of rearranging data and deleting intermediate files.

The relatively stable results between the data sets increases the confidence that this are not only sampling artifacts, but that the results are transferable to other HPC storage sites. These results for itself are promising so that deduplication should be considered for future HPC storage system architectures.

However, the trace data allows more observations, which will be described in the remaining section of this chapter. First, the relation of the file sizes and the deduplication ratios found (Section 18.1) will be discussed. Then, the focus is on the file types and file extensions to correlate them to the deduplication ratios (Section 18.2).

After that, the sharing of chunks across the different data sets within the same data center will be analyzed in Section 18.3. This allows insights into the sources of the deduplication potential. Another lookup on the causes of the deduplication potential is provided in following two sections. Section 18.4 analyzes the amount of deduplicated data is caused up copies of the identical file. While most presented results rely on content-defined chunking with an expected chunk size of 8 KB, Section 18.5 peaks into the behavior using other chunking strategies. In Section 18.6, it will be analyzed how much space is occupied by chunks that contain only zeros. In other settings, this is known to be a common source of duplication. In Section 18.7, the usage distribution of chunk references will be analyzed.

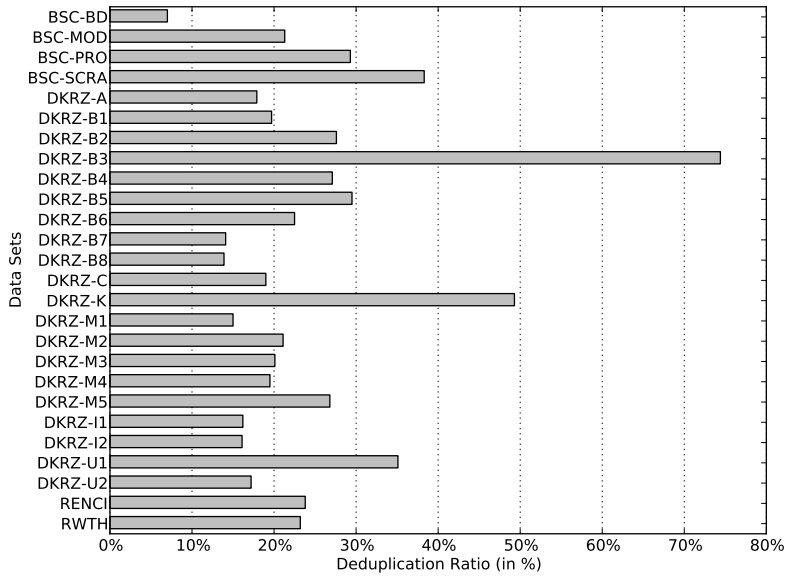


Figure 44: Deduplication ratio using content-defined chunking with an expected chunk size of 8 KB on different HPC data sets.

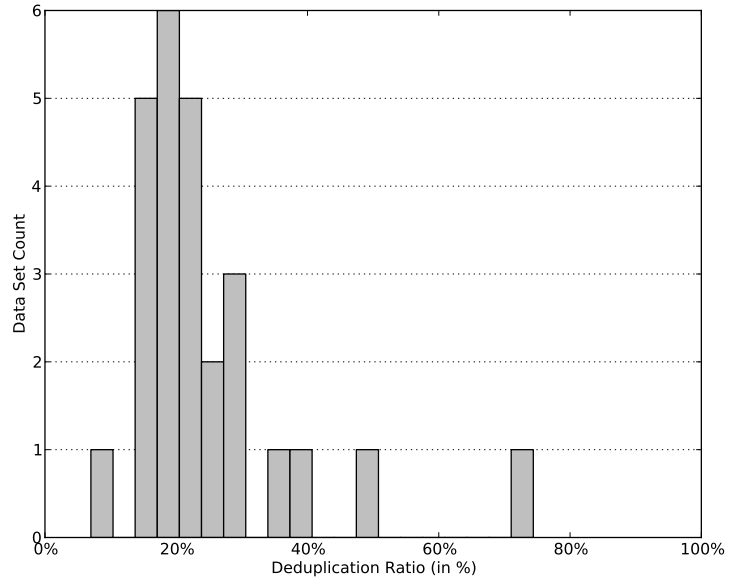


Figure 45: Histogram of the deduplication ratios using content-defined chunking with an expected chunk size of 8 KB on different HPC data sets (20 buckets).

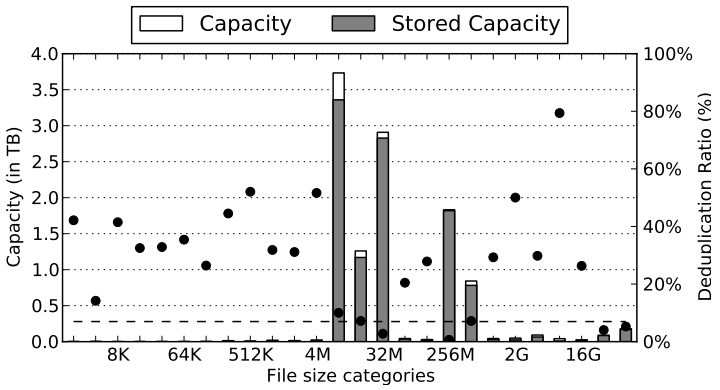


Figure 46: Deduplication results grouped by file size categories, *BSC-BD* data set.

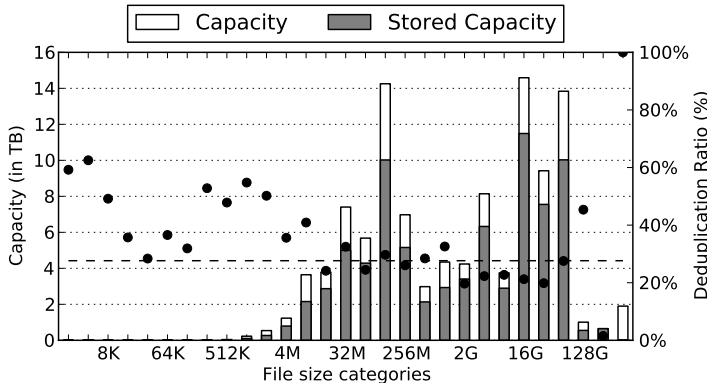


Figure 47: Deduplication results grouped by file size categories, *DKRZ-B2* data set.

Similar to other workloads, chunk usage is highly skewed and a small fraction of the shared chunks is responsible for a signification part of the data deduplication.

All following sections will focus on a subset of the data sets. It was not possible to perform this detailed analysis for all data sets because the necessary hardware for a Hadoop-based analysis has not been available anymore.

18.1 DEDUPLICATION BY FILE SIZES

This section explores the relation between the deduplication ratio and the file size. As mentioned before, HPC file systems usually store a high ratio of small files, but the most capacity is occupied to the medium and large files in a file system.

While small files may place a burden on the metadata performance of HPC storage systems, they are not that significant from a deduplication point of view.

Figure 46 show the deduplication ratios categorized by file size category for the *BSC-BD* data set. The figure presents multiple pieces of information. Besides the logical capacity

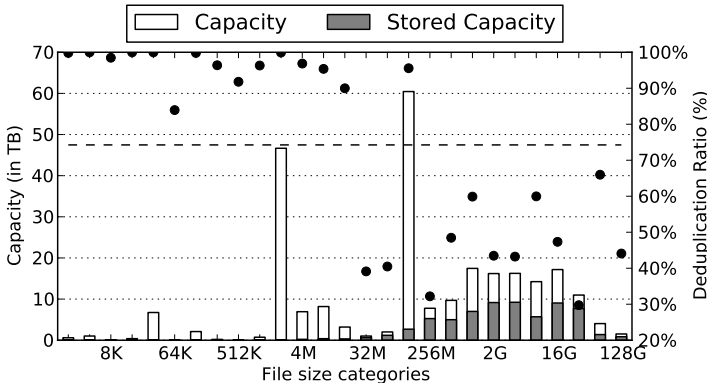


Figure 48: Deduplication results grouped by file size categories, *DKRZ-B3* data set.

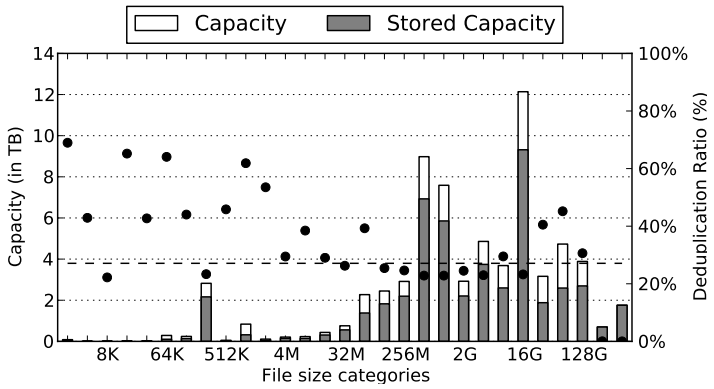


Figure 49: Deduplication results grouped by file size categories, *DKRZ-B4* data set.

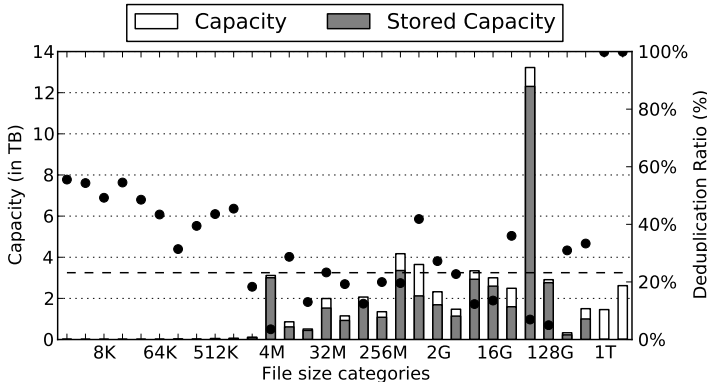


Figure 50: Deduplication results grouped by file size categories, *RWTH* data set.

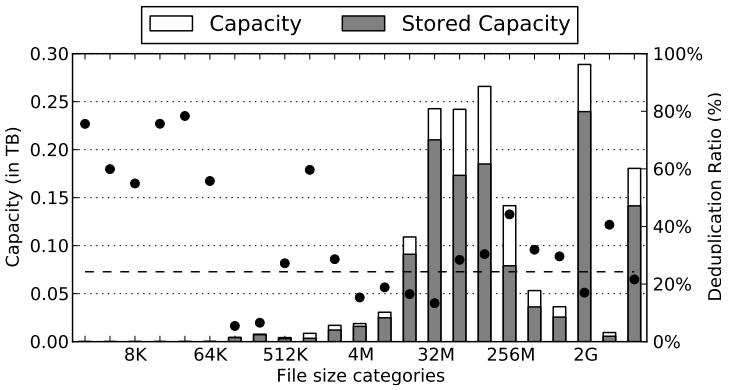


Figure 51: Deduplication results grouped by file size categories, *RENCI* data set.

for files with a given size and the capacity that is actually needed to store the deduplicated data, additional information is included in the figure. The dot indicates the deduplication ratio within a file size category. The dotted line denotes the aggregated overall deduplication ratio over all stored data. The corresponding figures for all data sets can be found in the appendix of this thesis. The figures of a subset of featured datasets are also presented in this section.

The deduplication ratio is varying in all data sets. The minimal deduplication ratio in one size category is always less than half the ratio of the file size category with the highest value. In all data sets, there is at least one significant category where the deduplication ratio is higher than 20%.

In some data sets, the deduplication ratio is relatively uniform in all file size categories with significant capacity. The *RENCI* data set is a prime example for this (Figure 51). The *DKRZ-B4* data set is very similar from this point, if the two largest file size categories are ignored (Figure 49).

In other data sets, the categories that carry most of the capacity drive the deduplication ratio either up or down. The *DKRZ-B2* data set is an example where the 128 MB category is by far the most important category as shown in Figure 47. Its deduplication ratio from over 60% drives the ratio up. Only one other significant file size category has a ratio higher than the overall deduplication ratio.

In the *DKRZ-B3* data set where the 128 MB category with a deduplication ratio of over 90% drives the overall deduplication ratio to nearly 75%. The data set exhibits a high deduplication in general, but no other significant category has a deduplication ratio higher than 60%. The reverse is true in the *BSC-BD* data set, which is the data set with the lowest deduplication ratio in the study. The major categories between 8 MB and 32 MB store the majority of the data and their deduplication ratio is driving the overall ratio.

Other pattern that emerges in the majority of data sets is that the deduplication ratio in the largest file size category is low to zero. This is the case in *BSC-BD*, *BSC-MOD*, *BSC-SCRA*, *DKRZ-B1*, *DKRZ-B4*, and *DKRZ-B6*. No pattern without exception: The ratio is near 100% in the *RWTH* data set, which indicate these files contain the data in that files is stored in other files multiple times. It is likely that these are archival files.

In other data sets, it can be observed that large files contain a lot of redundancy, too. For example, in the *DKRZ-A* and the *RWTH* data sets, the ratios for the largest two categories are over 90% (see Figure 50 for *RWTH* dataset).

On all data sets, small files (less than 1 MB) can be duplicated by a ratio of over 20%. In the *DKRZ-B3* data set, small files can even be reduced by more than 80% (Figure 48). Such high ratios for small files indicate full file copies.

18.2 DEDUPLICATION BY FILE EXTENSIONS

This section looks at the results from the perspective of file extensions (file formats). Especially important in this section are not the most common file extensions, but file extensions that occupy the most capacity.

Figure 52 shows the deduplication results for the 10 most space consuming file types in the DKRZ-B6 data set. Again, the corresponding figures for all data sets can be found in the appendix.

The overall deduplication ratio is 21.3% in *DKRZ-B6*, and most of the important file types actually have a comparable ratio. For example, 21.2% of the chunks in the nc files, which store data in the NetCDF format, can be deduplicated. From the 20 most space consuming file formats in this data set, 13 file formats have a deduplication ratio of 15% and higher. It is usually not the case that only one specific file type has a high deduplication ratio and no or only little deduplication ratio is found in the other files.

The most important exception in this data set is the `.gz` file format, which is a stream compression format using the gzip algorithm, a LZ77 variant [ZL77]. GZip-compressed files only have a deduplication ratio of 5.7%.

GRIB files (`.grb`) have only a slightly better deduplication ratio. Depending on user settings, GRIB files could be compressed, which could explain the low deduplication ratio in this data set. As described in detail for other workloads [MB09], compressed files hurt the deduplication ratio. Small changes in nearly identically compressed data results in severe re-codings so that most data is not recognized as duplicate.

Table 13 shows two types of information for the most important file extensions: `.tar`, `.grb`, `.nc/.ncf`, and the `N/A` file extension of files where no file type could be detected.

It shows the deduplication ratios, which is the fraction of redundant data of the total capacity used by this file extension. It also shows the total redundancy fraction, which is the fraction of the redundant data of this file extension on the total capacity of the system. The total redundancy fraction is therefore a function of the deduplication ratio of the file extension and of the fraction of capacity used by this extension.

The results show that (with the exception of the *BSC-DB* data set) NetCDF files and files without a file extension have a deduplication ratio higher than 20%. The deduplication ratio for these files seems to be consistent in different data sets. On the other hand, the deduplication ratio for other file types differs vastly in the different data sets. For example, grb and tar files range from almost no deduplication (`.grb` 1.1%, `.tar` 0.0% in *BSC-BD*, Figure 53) to over 87% in *BSC-SCRA* (Figure 54).

Tape archives require data transfers in larger granularity and data is usually aggregated using the tar format. A reason for the high deduplication ratio of (uncompressed) tar archives might be that the users also keep unpacked archives on the file system, leading to duplicated data. A high deduplication ratio in a tar archive file indicates that archives

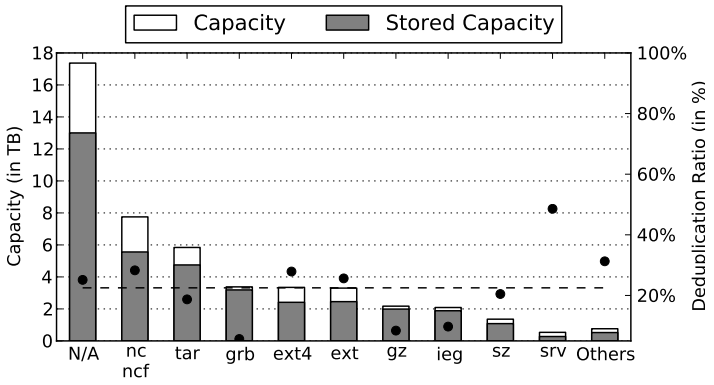


Figure 52: Deduplication results grouped by file types, *DKRZ-B6* data set.

Table 13: Deduplication ratios and the fraction of the redundant data of the total data set capacity of four usually space-consuming file types in different data sets with content-defined chunking and an expected chunk size of 8 KB.

Data set	Deduplication Ratio				Total Redundancy Fraction			
	`.tar`	`.grb`	`.nc`	`N/A`	`.tar`	`.grb`	`.nc`	`N/A`
BSC-BD	1.1%	0.0%	5.6%	7.6%	0.0%	0.0%	1.1%	4.6%
BSC-MOD	53.6%	50.0%	21.2%	13.5%	0.5%	0.0%	6.1%	4.0%
BSC-PRO	87.9%	11.3%	44.9%	24.7%	0.7%	0.4%	10.7%	13.7%
BSC-SCRA	80.8%	87.8%	51.2%	22.6%	0.0%	0.0%	23.3%	2.1%
DKRZ-A	18.2%	1.7%	16.4%	28.7%	2.3%	0.0%	13.1%	0.7%
DKRZ-B1	25.2%	23.0%	19.7%	23.4%	4.9%	0.2%	8.9%	3.0%
DKRZ-B2	25.6%	27.2%	38.2%	21.6%	9.7%	0.2%	12.7%	1.1%
DKRZ-B3	42.3%	21.1%	75.0%	28.6%	0.2%	0.0%	73.5%	0.2%
DKRZ-B4	14.1%	24.1%	37.3%	25.3%	1.6%	2.4%	6.2%	4.4%
DKRZ-B5	49.5%	7.4%	30.0%	41.6%	1.3%	0.6%	17.9%	3.4%
DKRZ-B6	18.7%	5.6%	28.3%	25.1%	2.3%	0.4%	4.6%	9.1%
DKRZ-B7	65.8%	7.9%	15.5%	25.9%	1.2%	2.8%	8.9%	1.1%
DKRZ-B8	21.3%	5.6%	20.7%	26.8%	1.1%	2.0%	5.3%	1.5%
DKRZ-K	57.2%	9.2%	47.7%	77.4%	1.0%	0.0%	39.1%	7.5%
DKRZ-M1	21.4%	3.8%	17.8%	21.2%	1.2%	1.0%	6.5%	4.7%
DKRZ-M2	25.0%	6.5%	26.5%	45.0%	6.3%	2.0%	6.8%	5.2%
RENCI	31.6%	67.8%	33.9%	30.7%	2.5%	0.0%	1.7%	16.4%
RWTH	45.4%	n.a.	59.2%	11.2%	0.8%	0.0%	0.0%	1.8%

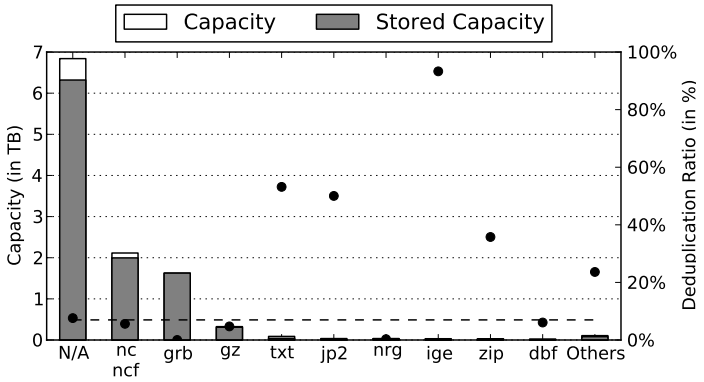


Figure 53: Deduplication results grouped by file types, *BSC-BD* data set.

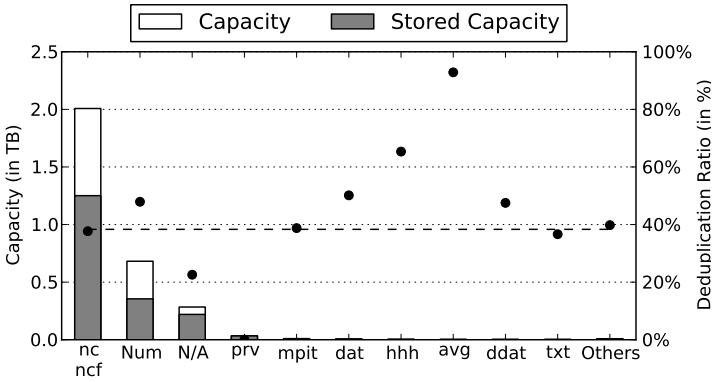


Figure 54: Deduplication results grouped by file types, *BSC-SCRA* data set.

have been created and the original data has not been removed. This is one of the cases that indicate a suboptimal usage of the storage system caused of its end-users. In most data sets, either tar archives do not usage a significant fraction of the overall storage capacity or the deduplication ratio is not very high so that the influence of this suboptimal usage is not high. In most data sets, the redundancy of tar files does not make up more than 1 or 2% of the total data (see Table 13). However, in the *DKRZ-B2* data set, a high deduplication ratio and a high importance come together and around 9.7% of the total storage capacity is used by redundancy from tar files.

18.3 CROSS DATA SET SHARING

Up to now, each data set was analyzed independently from the other, even when they originated on the same physical storage system. Now, the additional potential from deduplication due to chunk sharing between data sets is assessed. "Cross data set shared chunks" are chunks that occur in both data sets.

A shared chunk would mean users from independent projects store identical regions in some files. This could happen, for example, when a scientific application is used in several projects and similar input data is used, e.g., boundary conditions for simulation. If the deduplication domain is the individual data set, such a chunk will be stored twice. The redundancy removed within data sets is a lower bound of the possible global deduplication.

The cross data set sharing has been analyzed for a subset of all data sets. There the deduplication ratio is compared between the sum of the data reduction if each data set is its own deduplication domain and the setting where the union of the data sets is the deduplication domain. If the domains of deduplication are the four *BSC* data sets themselves, the aggregated deduplication ratio is 20.5%. By combining all data sets into one large deduplication domain, the deduplication ratio increases to 22.4% (+9.2%).

The *DKRZ-U1* and *DKRZ-U2* data sets show a similar picture. The shared chunks in the data sets *DKRZ-U1* and *DKRZ-U2* increases the ratio from 25.3% to 26.2% (+4.0%).

Cross data set sharing between the data sets *DKRZ-B1* and *DKRZ-B2* increases the deduplication ratio by 14.8% from 23.6% to 27.1%. The individual deduplication rates are 19.7% and 27.6%; the aggregated rate is 23.6%.

While a 15% increase as seen in the DKRZ-B example is significant, most of the redundancy originates within the same data set. This is different from other workloads. For example in virtual machine workloads where most of the redundancy comes from sharing across virtual machine images [JM09].

18.4 FULL FILE DUPLICATES

It is a simple variant of data deduplication to eliminate redundancy when the contents of two files are identical. This is also called single instancing [BCGD00]. This section evaluates how much of the redundancies removed by data deduplication can also be removed by single instancing.

Based on the chunk fingerprints, a fingerprint of the complete file data is derived by concatenating all chunk fingerprints and then calculating a new fingerprint from the chunk fingerprint list. This file fingerprint is not identical to a fingerprint of the file data itself,

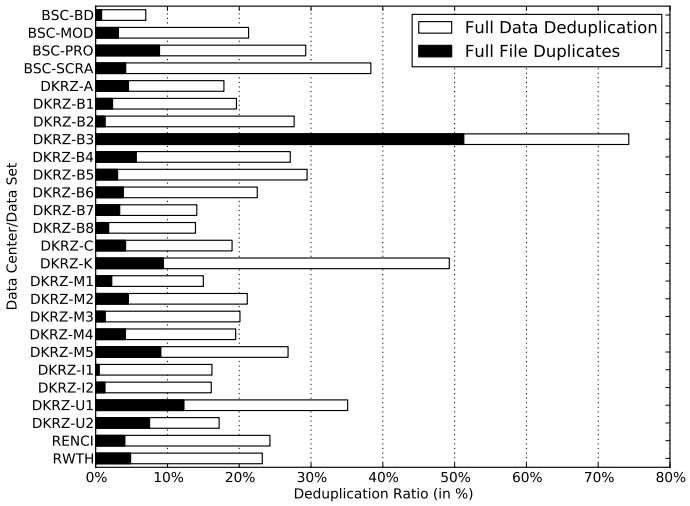


Figure 55: Full file duplicates and the full deduplication ratio for different data sets.

but equivalent in that if two fingerprints derived from the chunk fingerprints are identical, the file content is also identical with a very high probability.

Figure 55 compares the deduplication ratio using the content-defined chunking with the savings based on the elimination of full file duplicates. For all but a single data set, the deduplication ratio using full file duplicates is 80% smaller when compared to full deduplication.

For example, for the *RWTH* data set 5% of data is duplicated, but with data deduplication an additional 17% of intra-file redundancy could be removed, resulting in a total reduction of 22%.

In the *DKRZ-B3* data set, 51% redundancy could be found by file duplication elimination only (and 74% with full data deduplication). This unusual behavior can be explained with file copies created by the scientists in preparation for archiving all the project data.

The results show that file duplicate elimination alone is not an effective technique in HPC storage because saving ratios drop significantly.

In another analysis, full file duplicates are searched between the all *BSC* data sets and between all *DKRZ-B* data sets. The analysis shows that cross data set file duplicates exist but not as many as expected. By combining all *BSC* data sets, an additional 1.8% deduplication rate is gained against 3.8% when the individual data sets are used. This is an increase by 50% but still small compared to the sub-file redundancy found with full data deduplication.

The overlap between data sets can be as little as 0.02% if *BSC-PRO* and *BSC-SCRA* are combined. The largest overlap seems to be between *BSC-MOD* and *BSC-PRO* with 1.0%. The results are similar in the *DKRZ* data set: The full file duplicate ratio increases from 2.3% to 4.7%.

18.5 OTHER CHUNKING STRATEGIES AND CHUNK SIZES

The chunk size and the chunking strategy used up to now is often applied in backup systems because it presents a suitable trade-off between deduplication rate and index overhead. Nevertheless, most of the redundant data comes from large continuous runs and a larger chunk size could find similar deduplication with a much lower overhead. Therefore, additional chunking strategies and chunk sizes have been used to assess the behavior of the deduplication ratio depending on the chunk size for some of the smaller data sets.

It was infeasible to evaluate all combinations of chunk sizes and chunking strategies on all data sets. However, given that the deduplication rate is relatively stable between data sets, a small sample may provide sufficient insight for the direct comparison between the different approaches.

The data set is a single 1.5 TB DKRZ project directory tree containing 10,000 files and it has been traced with 4 different chunking strategy/chunk size pairs: content-defined chunking and static chunking with 32 KB and 64 KB.

The deduplication ratio is 42.2% and 40.8% for content-defined chunking with an expected chunk size of 32 KB and 64 KB, respectively. As expected, the deduplication rate declines, but even with an average of 64 KB chunks a significant fraction of redundancy is detected. For fixed size chunks, the deduplication rate is significantly smaller than with content-defined chunking. Only 35.1%, respectively 33.8% of the redundant data blocks are detected with 32 KB and 64 KB fixed size chunks. Unpacked tar files are an example for which the rate declines with static chunking because file metadata is inserted into the data stream, which leads to different fingerprints for the fixed blocks.

The loss of detected redundancy is most severe in large files: For the largest file in the data set (80.0 GB), content-defined chunking with an expected chunk size of 32 KB finds a redundancy of 19.6%, while no redundancy is found with fixed size chunks. On the other hand, the disadvantage of fixed size chunking is smaller for the NetCDF files in the data set; the duplication is reduced from 31.2% to 23.6% if 32 KB chunks are used.

18.6 ZERO-CHUNK REMOVAL

The importance of the zero-chunk has been stressed multiple times in this thesis. To check if the zero-chunk is also common in HPC storage, some *DKRZ* data sets have been searched for it. The intuition is that, e.g., input files may contain matrix data. If these matrices are sparse and a lot of entries contain zeros, this also leads to zero-chunks.

Figure 56 shows the deduplication rate and the zero-chunk deduplication rate for all DKRZ data sets that have been investigated for the zero-chunk.

In the *DKRZ-A* data set, the chunk containing only zeros use 3.1% of the space. This also means that simply eliminating block containing only zeros can reduce the storage usage by 3.1%. In the *DKRZ-B3* data set, the zero-chunk savings make up 24.3%. For the other data sets, the values range between 0.8% (*DKRZ-B7*) and 17.7% (*DKRZ-M5*). In the *DKRZ-M5* data set, zero-chunks make up more than 66% of the total redundancy.

It has been observed by Dayal that sparse files are used in HPC storage systems [Day08]. His data indicates that at most 0.5% of the data are stored in unallocated blocks of sparse files.

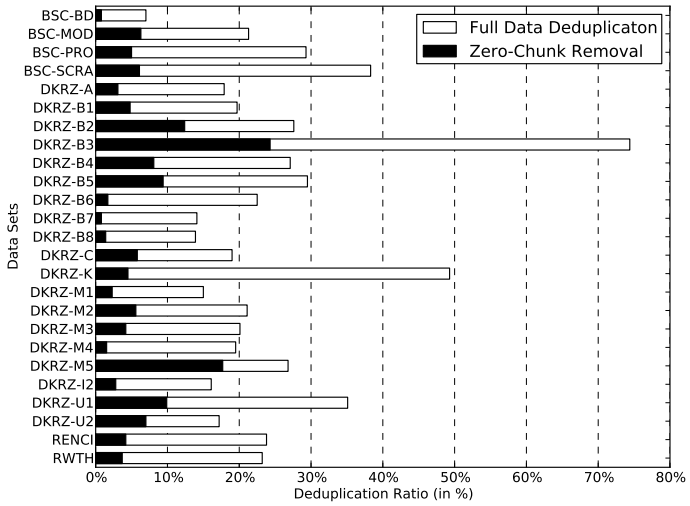


Figure 56: Zero chunk duplicates and full deduplication ratio for different DKRZ data sets.

The FS-C trace tool is not able to track sparse files. Therefore, it is not possible to finally answer the question if a zero-chunk really occupies capacity or if they are already stored in sparse files. If files are sparse, some fraction or even all of these zero-chunks are not stored on disk but handled as sparse blocks by the file system. However, not all zero data is handled as sparse blocks by file systems. Most file systems only detect this if the user explicitly seeks beyond the end of the file. Write operations where the data contains zeros are usually not detected.

Based on the collected trace data, it is impossible to determine to which amount zero-chunks are already stored in sparse files.

18.7 CHUNK USAGE STATISTICS

The previous section discussed the contribution of the zero-chunk to the deduplication. It was shown that the zero-chunk contributed more to the redundancy than other chunks and, thus, has the highest data reduction potential. Besides the zero-chunk, there might be chunks that occur more often than other chunks so that chunk usage distribution is skewed. This has been investigated by the author for a non-HPC-dataset in a previous study [MB09].

To investigate the chunk usage skewness in these data sets, the chunk usage statistics of the *DKRZ-B1* data set are assessed in depth. The results show the skewness: Of all chunks, 90% were only referenced once. This means that the chunks are unique and do not contribute to deduplication. The most referenced chunk is the zero-chunk, which has been discussed before. The second most referenced chunk has 5.4 million references. The mean number of references is 1.2 and the median is a single reference.

Figure 57 shows the cumulative distribution function (CDF) for the most referenced chunks that are referenced at least twice. As can be seen, the most referenced 5% of all

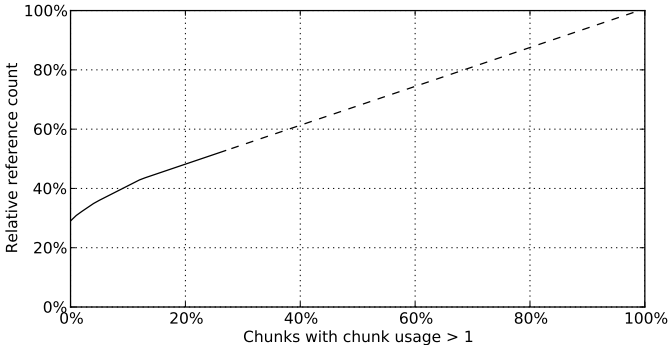


Figure 57: Cumulative distribution function (CDF) of the most referenced chunks in the DKRZ-B1 data set that are referenced at least twice. The dotted region consists of chunks that are referenced exactly twice.

chunks account for 35% and the first 24% account for 50% of all references. Of all chunks that are referenced multiple times, about 72% were referenced exactly twice (dotted region).

19

DISCUSSION

In the last chapter, it was shown that HPC file systems contain a notable amount of redundancy. This chapter discusses the results, possible explanations, and other observations.

There are two viewpoints on why there exists redundancy in the HPC file systems: The first one is that redundancies within an HPC storage system, and especially within a project, indicate bad practice (and workflows).

Exact file copies cause at least some unnecessary redundancies. Another kind of "obvious bad user behavior" is keeping unpacked tar files on the storage. This happens, for example, when copying tar files between the tape archival system and the online storage and not removing the tar file afterwards. The deduplication analysis can be used in HPC centers to detect suboptimal workflows to support their users.

However, the results show that such suboptimal behaviors exist, but they do not account for the majority of the observed redundancies. For example, the redundancy of tar archive usually occupies less than 2% of the total storage capacity. However, in one data set redundancy of tar archives alone occupies nearly 10% of the total storage capacity.

Another viewpoint is that the redundancy naturally arises from the data and the workflow of HPC application. In this case, educating the users of HPC storage systems cannot easily eliminate the redundancy, but it might be worthwhile to in-cooperate data deduplication into HPC storage systems.

Given the capacity needed for HPC storage systems and the high performance requirements, deduplication can only be carefully applied in certain parts of a HPC storage infrastructure. Classic deduplication architectures used in backup systems are not applicable to HPC storage because of throughput and scalability requirements as well as the specific IO patterns of HPC storage.

The purpose of this research is not to answer how to apply deduplication in HPC storage, but to show that it would be worthwhile to research and develop such systems. A few hints to consider based on the study results are:

POST-PROCESSING: Most backup deduplication systems perform the deduplication detection and removal inline. In backup systems one of the reasons why inline deduplication is often preferred is that the redundancy is removed at the time the data is written. Otherwise the back-end storage capacity needs to be highly over-provisioned to accommodate the data of a complete backup before the redundancy is removed. It is not critical in HPC storage systems to avoid such over-provisioning as at each time each HPC application only writes new data that occupies a small fraction of the overall storage capacity.

Deduplication as an additional post-processing step reduces associated overhead and minimizes impact on write performance.

However, in the future even inline deduplication might become applicable to reduce the IO requirements. This might be the case if the computing capacity continues growing while the performance penalty of storing data increases.

LARGE CHUNKS: Using large chunks can reduce the overhead of deduplication, but there is a trade-off as larger chunk sizes detect less redundancy. The optimal setting for a deduplicating HPC system is probably a chunk size much larger than the 8 KB, which has been used in this study.

SMALL FILES: Files smaller than 1 MB account for an insignificant amount of the used disk space (though their number is very large). It might be worthwhile to avoid deduplicating these files using such resource-intensive approaches as content-defined chunking. For example, small files could be excluded from the redundancy detection or they could only be deduplicated at a full file level.

PROJECT-LOCAL DEDUPLICATION: The analysis shows that the majority of the deduplication comes from a local scope. Cross project and cross data set sharing provides part of the overall deduplication, but it does not cause the majority of the redundancies found. This contrasts, e.g., with the experiences from deduplicating virtual machine images, where most of the duplicates are not found in a single virtual machine image but by sharing between many different virtual machines (see, e.g., Jin et al. [JM09]). This could be a helpful hint for future research and development, as smaller project-local indexes might be enough to find the majority of redundancies.

VOLUME SPECIFIC: Since an HPC center offers several independent volumes with different characteristics it might only be possible to integrate deduplication into specific volumes. Scratch folders are likely to be subject to constant changes and data is only kept for a short time until it has been analyzed with post-processing tools. However, data in project directories (and home-directories) is stored for longer periods and benefits from deduplication by increasing available storage space.

ZERO-CHUNK REMOVAL: Sparse files can be seen as an arcane form of deduplication. However, today large runs of zeros are not removed and transformed into sparse blocks. The findings indicate that this might be worthwhile for a file system to avoid writing empty blocks. Checking if a block only contains zeros is essentially free with respect to CPU and storage.

The biggest obstacle for the usage of data deduplication is its integration into parallel file systems. Parallel file systems are optimized for certain IO pattern like striped IO, which is often used by HPC application. To achieve the necessary throughput, the deduplication system has to ensure that these IO pattern are handled with sufficient performance.

Furthermore, in comparison to backup systems, HPC storage systems are required to perform different workloads and IO pattern well. Backup systems are often optimized for sequential writes operations only. It is necessary to find ways to support all necessary IO pattern for HPC storage systems. For inline deduplication, such a support is necessary for write and read operations. In a post-processing deduplication system, write operations are

done without any interference by the data deduplication, but read operations still require care.

On the other hand, the history of data deduplication showed that its usage is constantly extending to new fields. After the usage for backup, it has spread even to primary storage using different approaches to overcome the disk bottleneck [WMR12]. While none of these approaches can be a directly applied to combine data deduplication and HPC storage, the research of this problem appears to be worthwhile.

20

CONTRIBUTION

The analysis presented in this part shows for the first time that HPC storage systems have a significant potential for data deduplication.

In most data sets, between 20 and 30% of the data is redundant and could be removed by data deduplication techniques. However, this potential to reduce the amount of storage capacity that is required is not facilitated by today's HPC file systems.

The data shows that data deduplication techniques might help in the future to reduce the enormous storage requirements of HPC storage. However, today's deduplication approaches and deduplication systems are not suitable for the need of HPC storage.

These findings should motivate storage researchers and storage engineers to investigate if and how data reduction techniques as data deduplication can be integrated into HPC storage systems.

Part V

CONCLUSION

21

CONCLUSION AND FUTURE WORK

In this thesis, I have shown multiple novel approaches to increase the scalability and throughput of data deduplication systems as well as to further reduce the storage capacity needed by compressing block and file recipes. Furthermore, I have shown in a large-scale study that data deduplication would allow reducing the storage requirements of HPC storage.

In this chapter, I will give a conclusion of the thesis and highlight possible future work and open questions.

21.1 CONTRIBUTIONS

In Part II, the design for an exact inline-deduplication cluster was presented. The cluster deduplication system detects the same redundant chunks as a single-node deduplication system, while still using small chunk sizes. Key elements of the design are

1. the separation of the node on which a chunk index entry for a chunk can be found and the node on which the chunk data is stored in a container.

2. a combination of the communication protocol and the storage design to avoid sending large amounts of data over the network.

In most situations, only small messages containing only the fingerprints of the data are exchanged over the internal network.

A prototype based on the dedupv1 system was used to evaluate the performance of the design. The evaluation shows that it is possible to combine exact deduplication, small chunk sizes, and scalability within one environment while using only commodity GBit Ethernet interconnections.

The goal of the research on an exact data deduplication cluster was to show that such a design is possible with reasonable efficiency and scaling properties and to evaluate the limitations of such an approach.

A combination of the prototype-based evaluation, a modeling of the communication pattern, and simulations has been used to predict the scalability for larger cluster sizes and to explore the limits of the scalability. The results show that the design has a limited scalability if the cluster is small. However, for larger chunk sizes and larger cluster sizes, the scalability improves.

Part III introduced new techniques based on the block and file recipe data structure of data deduplication systems.

On the one hand, I proposed using deduplication-aware compression methods to reduce the size of block and file recipes. With limited storage and memory overhead, these methods allow shrinking the file recipes by up to 93%. All methods are efficient and scalable. They do not cause additional IO operations and their storage overhead is negligible compared to the storage saved by applying the compression schemes.

Furthermore, I proposed a novel approach to overcome the chunk lookup disk bottleneck, which is based on block and file recipes. The Block Locality Cache uses the locality properties of the existing block index data structure to predict blocks that are likely related to the currently written data. It is shown that this approach allows to capture the locality of the latest backup run and is, therefore, likely to be less prone to aging.

Part IV shows that HPC storage systems have a significant potential for data deduplication. In most data sets, between 20 and 30% of the data is redundant and could be removed by data deduplication techniques. However, this potential to reduce the amount of storage capacity that is required is not facilitated by today's HPC file systems.

The data shows that data deduplication techniques might help in the future to reduce the enormous storage requirements of HPC storage. However, today's deduplication approaches and deduplication systems are not suitable for the need of HPC storage. It is expected that these findings may motivate storage researchers and storage engineers to investigate if and how data reduction techniques as data deduplication can be integrated into HPC storage systems.

In summary, I presented multiple significant contributions to the field of data deduplication. Using these contributions, data deduplication systems are able to scale to a cluster of machines using exact data deduplication and small chunk sizes. The disk bottleneck is avoided using a novel recipe-based caching scheme and the overall storage usage of a data deduplication system is further reduced by compressing block and file recipes.

Furthermore, I peaked into a new application field where data deduplication techniques might be applied successfully. However, probably the contributions on the design of deduplication systems are not applicable in this new field.

21.2 FUTURE WORK

The research presented in this thesis also leads to new open questions and allows future work.

The evaluation results of the di-dedupv1 system indicate that the system will further scale until the network infrastructure, e.g., the network switch, becomes a bottleneck. Future work might research if the derived scaling properties hold in clusters with more than 100 nodes and how the di-dedupv1 architecture could be scaled if a hierarchy of networks is involved.

It is still an open problem how the recipe compression approaches, presented in Part III, can be conceptually combined with approaches without a full chunk index, e.g. Sparse Indexing [LEB+09].

Another open problem are long-term aging effects. The current data indicates that the long-term effects can be handled using the described code word deactivation approach, but a deeper investigation of aging effects would be worthwhile.

Future work may include research on how the Block Locality Cache might be extended to work in cloud-backup settings. One issue in cloud backup is that most existing ap-

proaches are not able to cope with a massive number of data streams where each stream itself has locality properties but there is no overall ordering. For example, the de-linearizion of container caching-based approaches is likely to reduce its effectiveness further. Also, the Block Locality Cache is, as proposed before, not able to cope with a massive number of data streams. However, an extension of the Block Locality Cache where only blocks recipes are fetched from previous backups of the same data stream (or from highly similar data stream) might be able to overcome the chunk lookup disk bottleneck in a cloud-backup setting.

Another question that is not answered in this thesis is if and how the approaches for block and file recipes compression and the Block Locality Cache can be integrated into the cluster deduplication systems, di-dedupv1. Until now, these approaches have been focused on the single-node variant dedupv1.

Furthermore, the evaluation of the Block Locality Cache indicates that in some data sets locality-based approaches tend to be better than similarity-based approaches, but not in others. It would be interesting to research if these differences are also found in other data sets and which properties cause this behavior.

Part IV has established that data deduplication can be useful in HPC storage environments. The question how data deduplication might be integrated concretely and which trade-offs between storage performance and data deduplication efficiency are suitable is still an open question.

Part VI

APPENDIX

A

ADDITIONAL FIGURES: BLOCK LOCALITY CACHING EVALUATION

This appendix chapter contains additional figures that complete the figures on the read pattern of the novel Block Locality Caching approach and other approaches the BLC is compared too.

In Section A.1, shows the block read pattern of the Block Locality Caching approach in the *UBP* data set. Section A.2 shows the container read pattern of the Container-Caching approach in the *UBP* data set. Section A.3 shows the segment read pattern of Exact Sparse-Indexing in the *UPB* data set. Finally, Section A.4 shows the bin read pattern of Exact Block-based Extreme Binning in the *UPB* data set.

The sections also contain the figures which have been in Chapter 11. The explanation of the shown information can be found that chapter.

A.1 BLC READ PATTERN

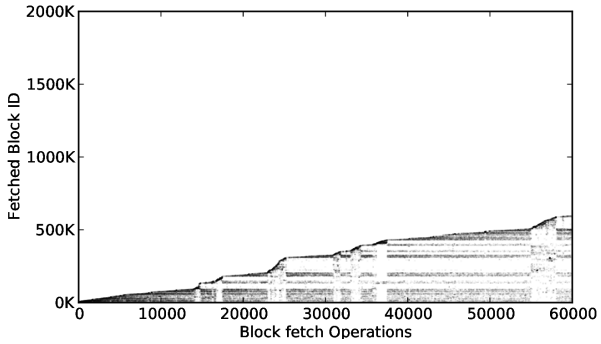


Figure 58: BLC block fetch operations in week 1 (*UPB* dataset).

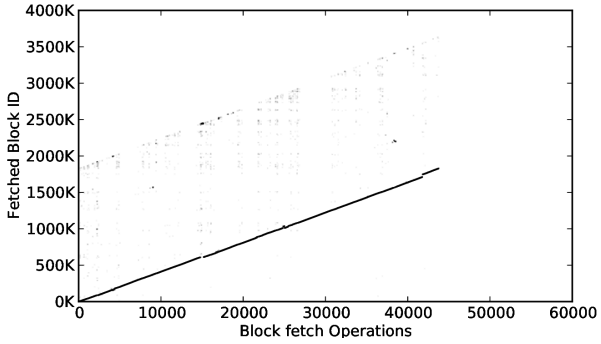


Figure 59: BLC block fetch operations in week 2 (*UPB* dataset).

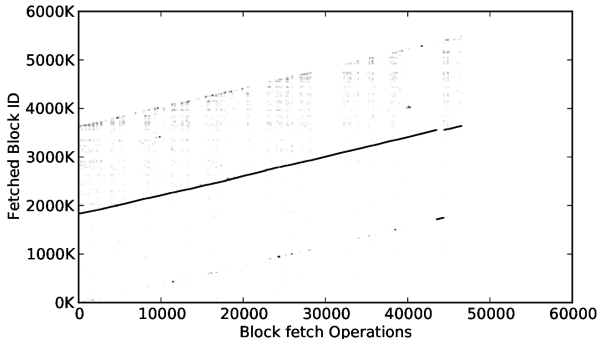


Figure 60: BLC block fetch operations in week 3 (*UPB* dataset).

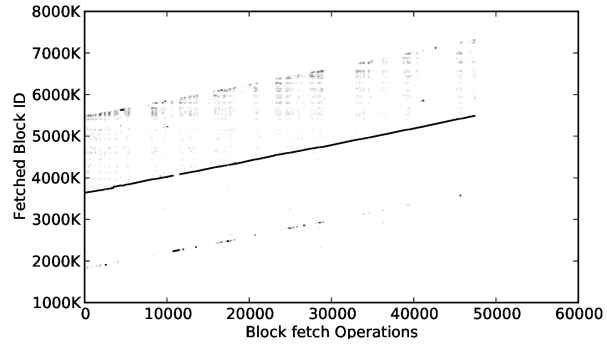


Figure 61: BLC block fetch operations in week 4 (*UPB* dataset).

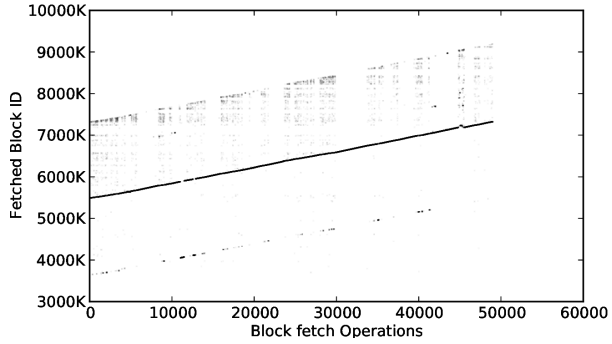


Figure 62: BLC block fetch operations in week 5 (*UPB* dataset).

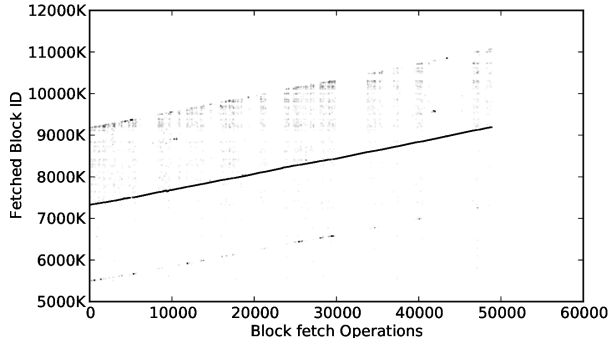


Figure 63: BLC block fetch operations in week 6 (*UPB* dataset).

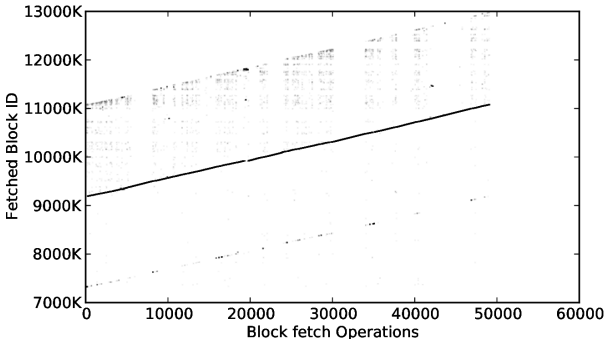


Figure 64: BLC block fetch operations in week 7 (*UPB* dataset).

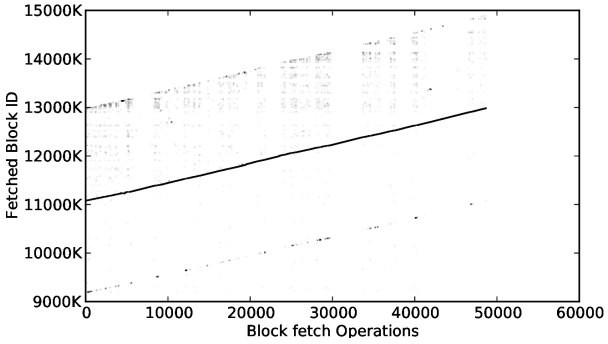


Figure 65: BLC block fetch operations in week 8 (*UPB* dataset).

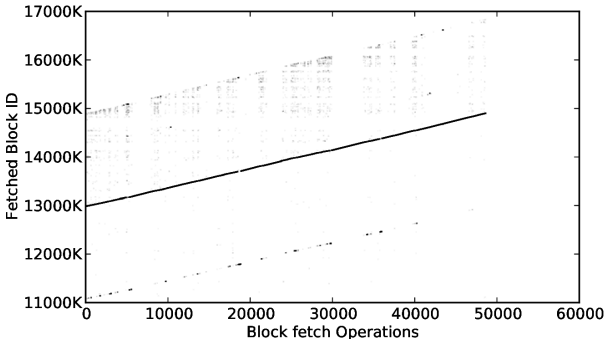


Figure 66: BLC block fetch operations in week 9 (*UPB* dataset).

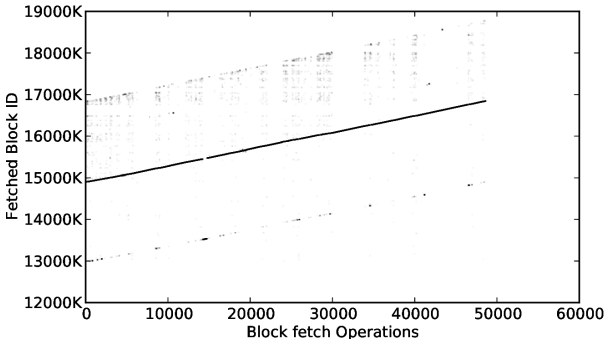


Figure 67: BLC block fetch operations in week 10 (*UPB* dataset).

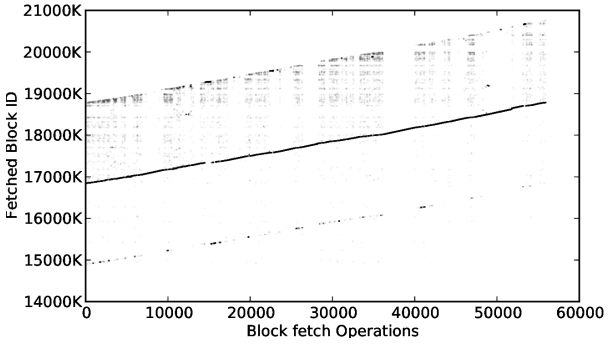


Figure 68: BLC block fetch operations in week 11 (*UPB* dataset).

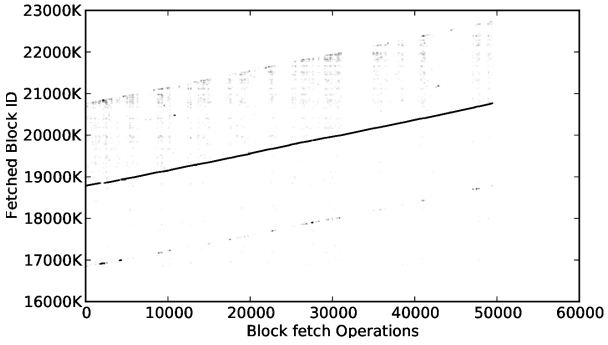


Figure 69: BLC block fetch operations in week 12 (*UPB* dataset).

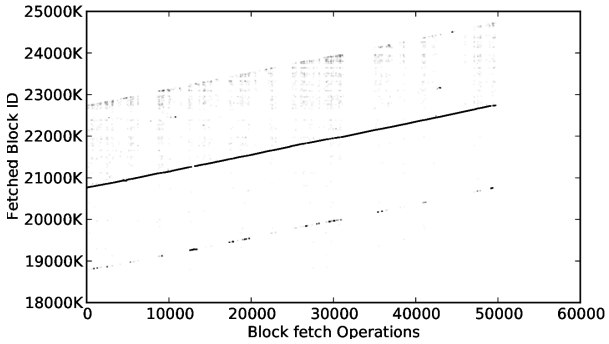


Figure 70: BLC block fetch operations in week 13 (*UPB* dataset).

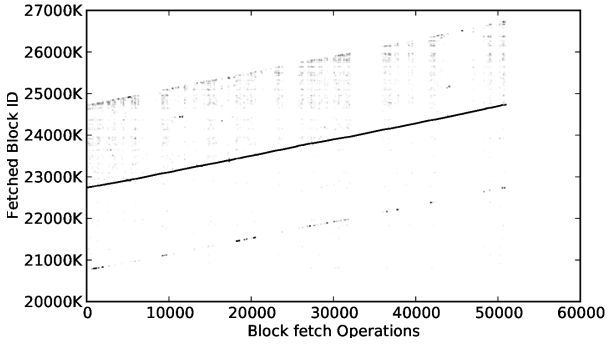


Figure 71: BLC block fetch operations in week 14 (*UPB* dataset).

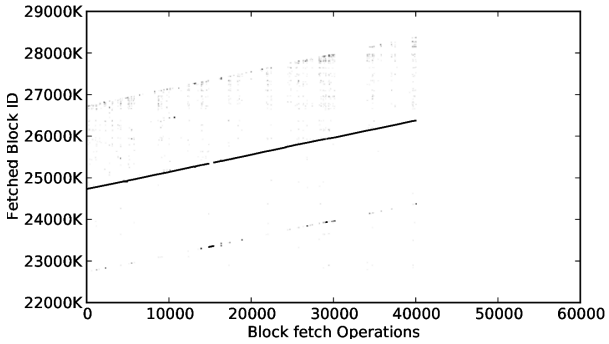


Figure 72: BLC block fetch operations in week 15 (*UPB* dataset).

A.2 CONTAINER READ PATTERN

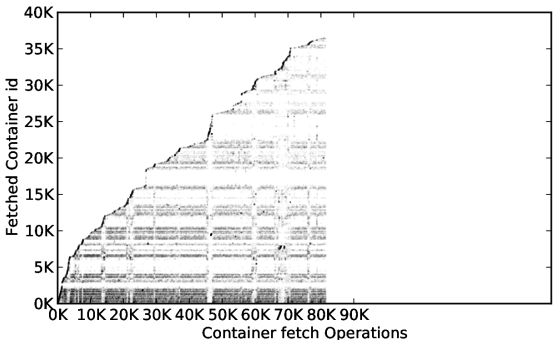


Figure 73: Container read operations in week 1 (*UPB* dataset).

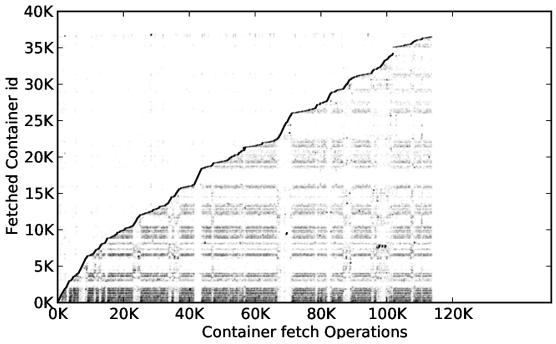


Figure 74: Container read operations in week 2 (*UPB* dataset).

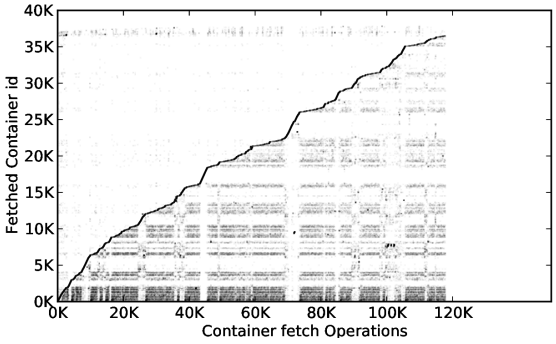


Figure 75: Container read operations in week 3 (*UPB* dataset).

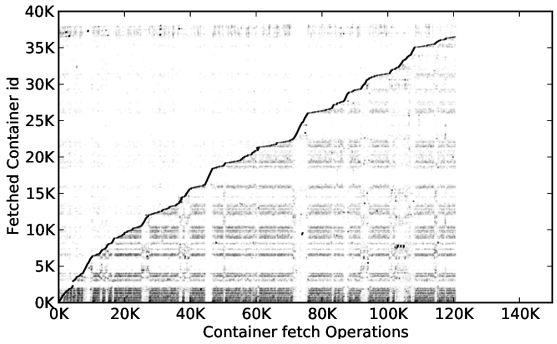


Figure 76: Container read operations in week 4 (*UPB* dataset).

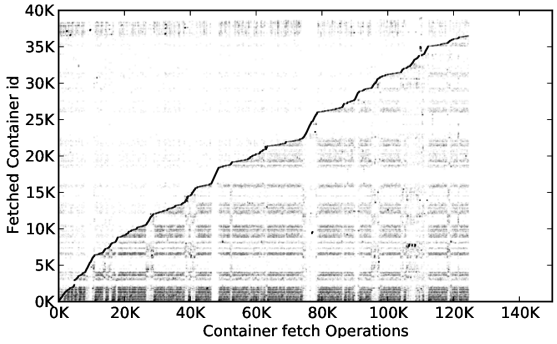


Figure 77: Container read operations in week 5 (*UPB* dataset).

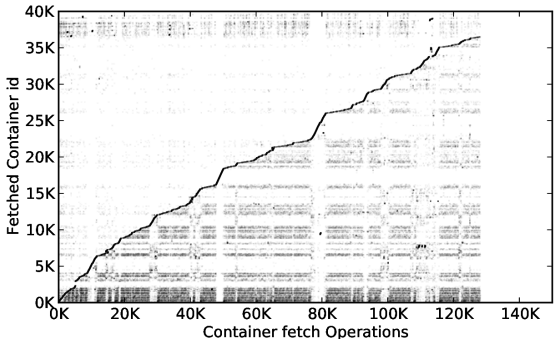


Figure 78: Container read operations in week 6 (*UPB* dataset).

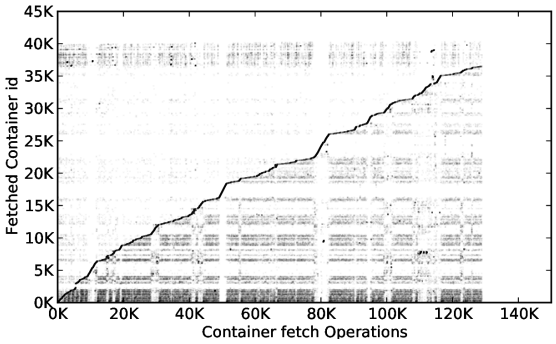


Figure 79: Container read operations in week 7 (*UPB* dataset).

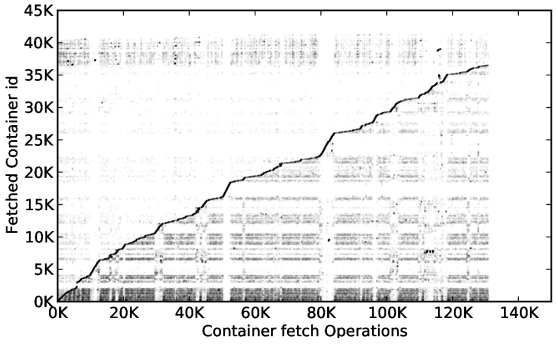


Figure 80: Container read operations in week 8 (*UPB* dataset).

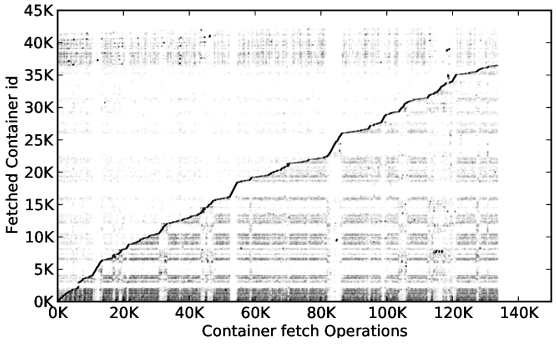


Figure 81: Container read operations in week 9 (*UPB* dataset).

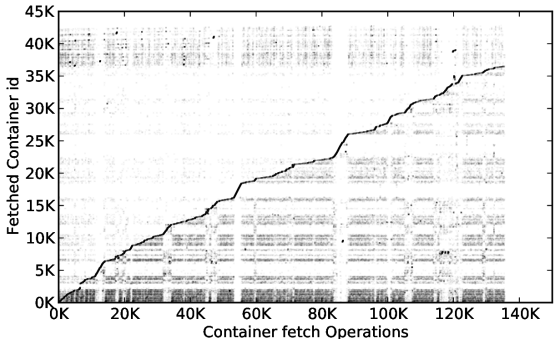


Figure 82: Container read operations in week 10 (*UPB* dataset).

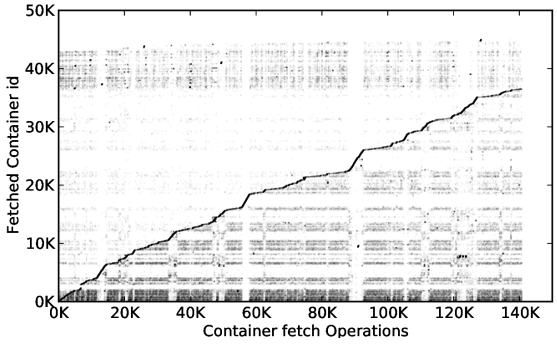


Figure 83: Container read operations in week 11 (*UPB* dataset).

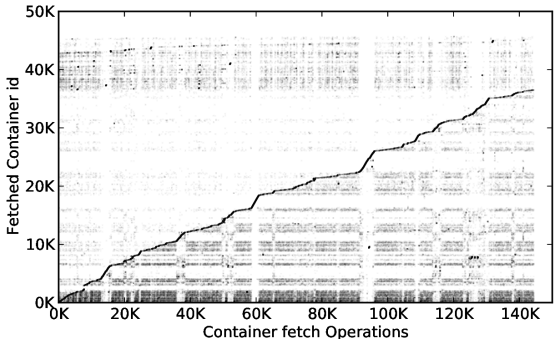


Figure 84: Container read operations in week 12 (*UPB* dataset).

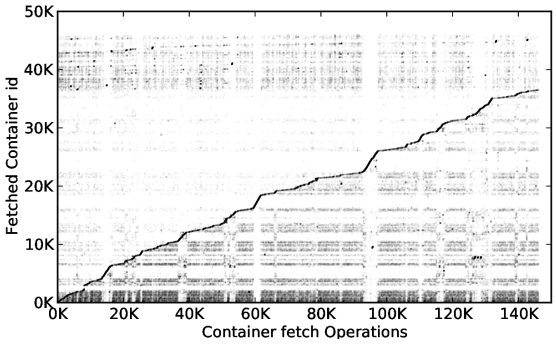


Figure 85: Container read operations in week 13 (*UPB* dataset).

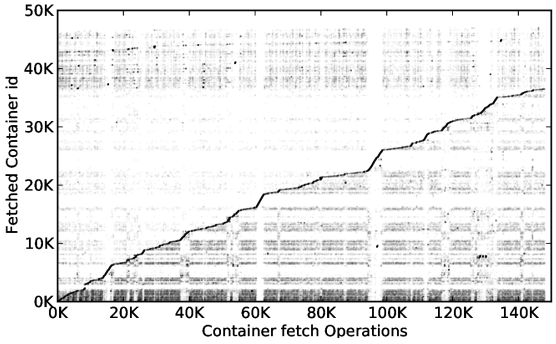


Figure 86: Container read operations in week 14 (*UPB* dataset).

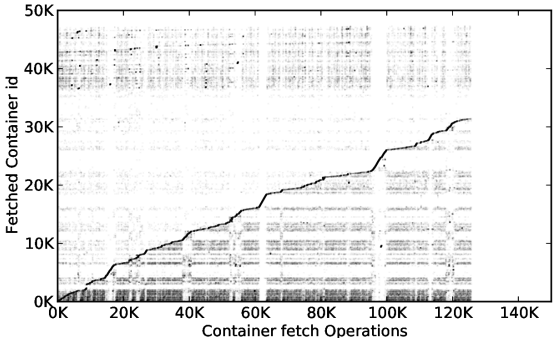


Figure 87: Container read operations in week 15 (*UPB* dataset).

A.3 SEGMENT FETCH PATTERN

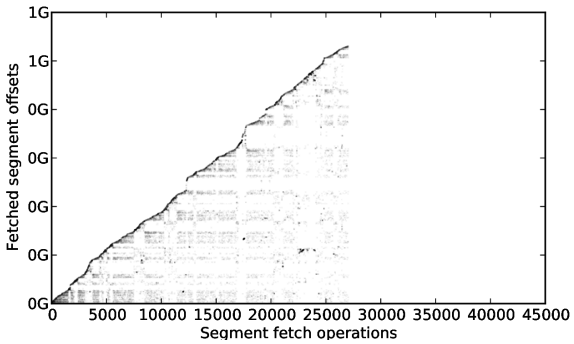


Figure 88: Segment fetch operations in week 1 (*UPB* dataset).

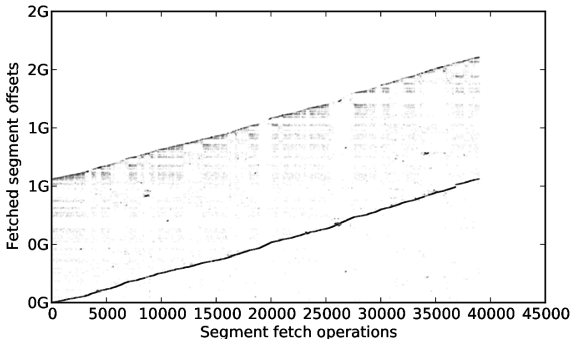


Figure 89: Segment fetch operations in week 2 (*UPB* dataset).

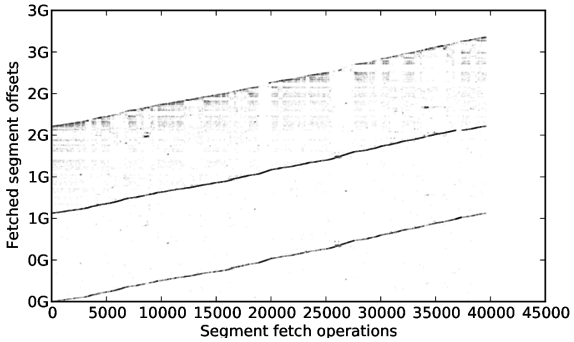


Figure 90: Segment fetch operations in week 3 (*UPB* dataset).

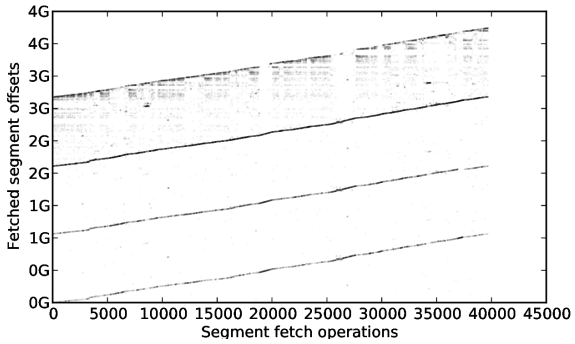


Figure 91: Segment fetch operations in week 4 (*UPB* dataset).

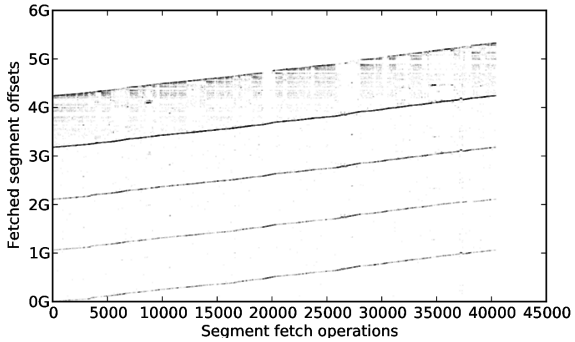


Figure 92: Segment fetch operations in week 5 (*UPB* dataset).

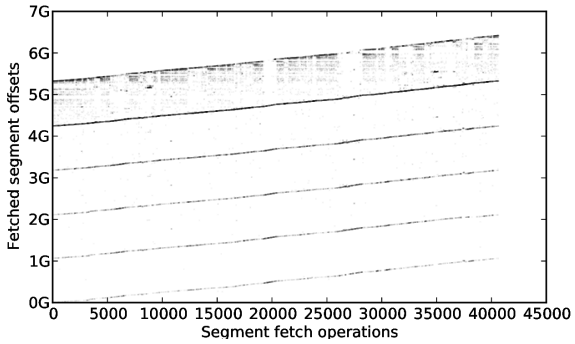


Figure 93: Segment fetch operations in week 6 (*UPB* dataset).

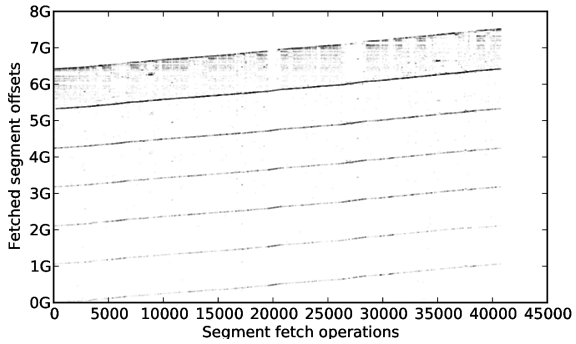


Figure 94: Segment fetch operations in week 7 (*UPB* dataset).

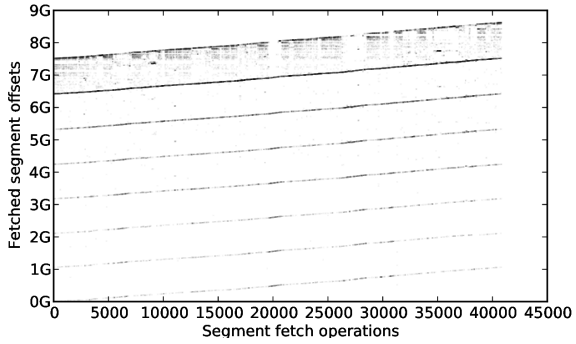


Figure 95: Segment fetch operations in week 8 (*UPB* dataset).

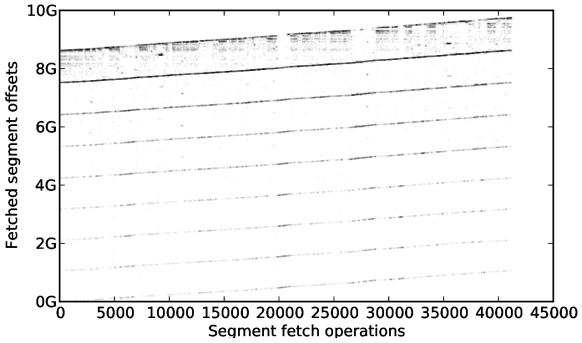


Figure 96: Segment fetch operations in week 9 (*UPB* dataset).

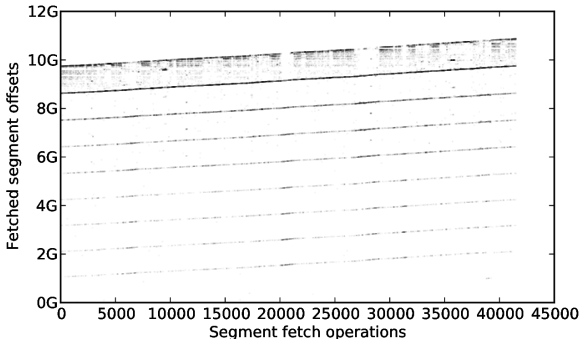


Figure 97: Segment fetch operations in week 10 (*UPB* dataset).

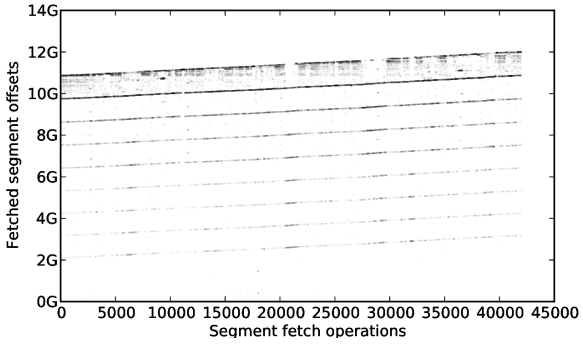


Figure 98: Segment fetch operations in week 11 (*UPB* dataset).

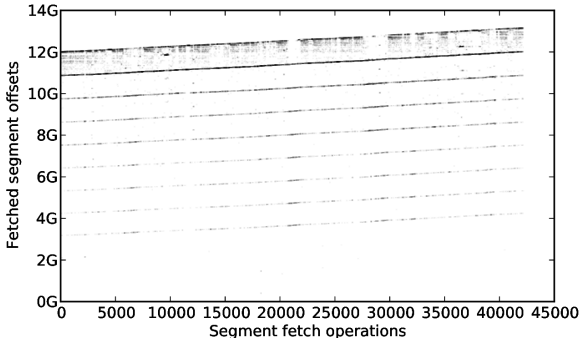


Figure 99: Segment fetch operations in week 12 (*UPB* dataset).

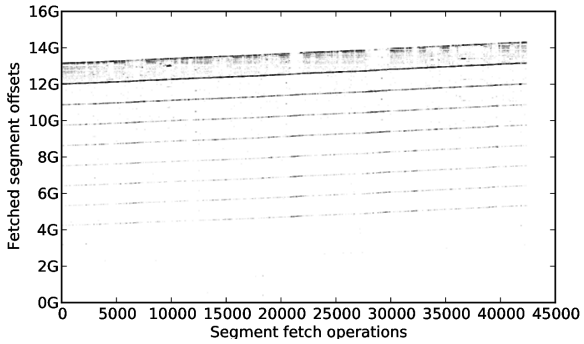


Figure 100: Segment fetch operations in week 13 (*UPB* dataset).

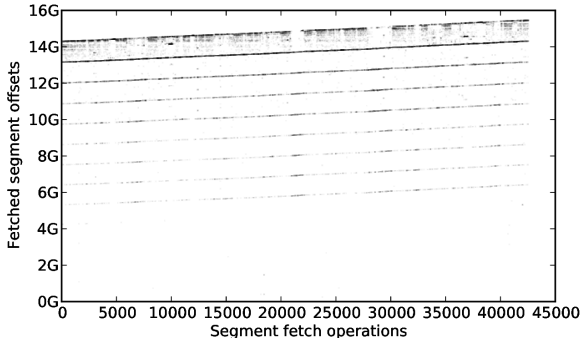


Figure 101: Segment fetch operations in week 14 (*UPB* dataset).

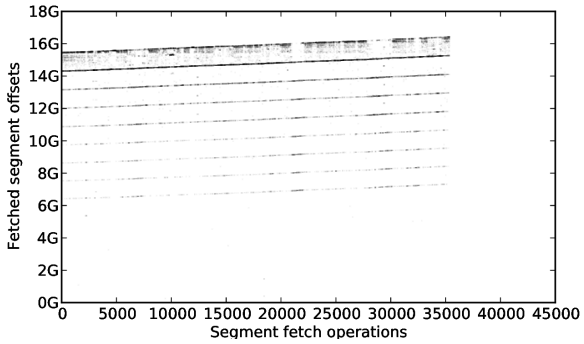


Figure 102: Segment fetch operations in week 15 (*UPB* dataset).

A.4 BIN FETCH PATTERN

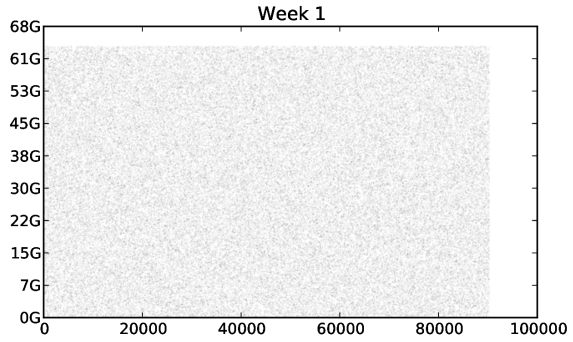


Figure 103: Bin fetch operations in week 1 (*UPB* dataset).

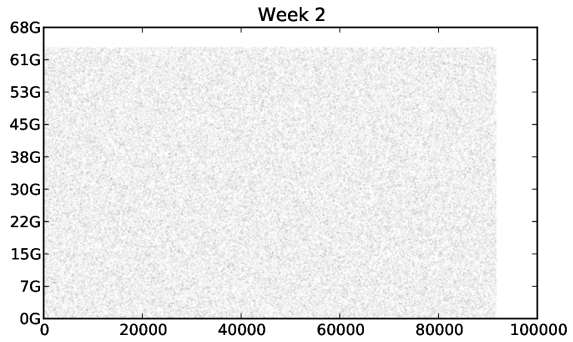


Figure 104: Bin fetch operations in week 2 (*UPB* dataset).

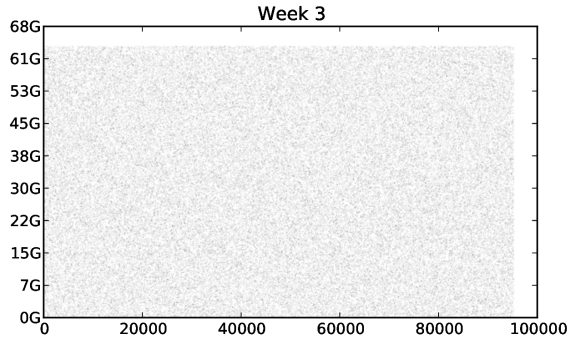


Figure 105: Bin fetch operations in week 3 (*UPB* dataset).

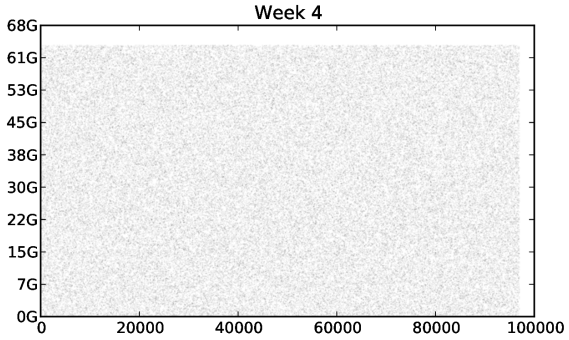


Figure 106: Bin fetch operations in week 4 (*UPB* dataset).

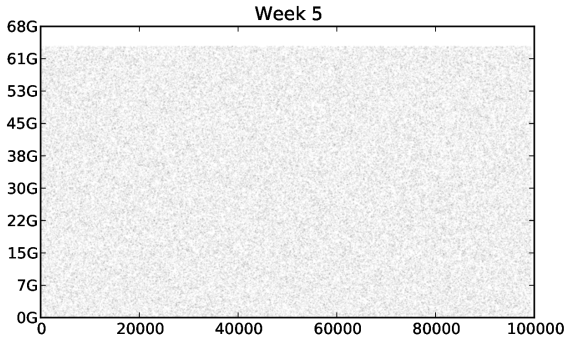


Figure 107: Bin fetch operations in week 5 (*UPB* dataset).

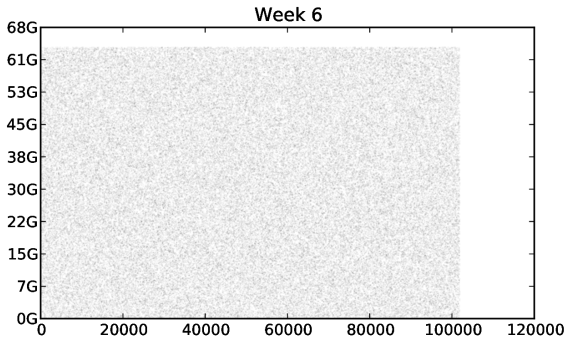


Figure 108: Bin fetch operations in week 6 (*UPB* dataset).

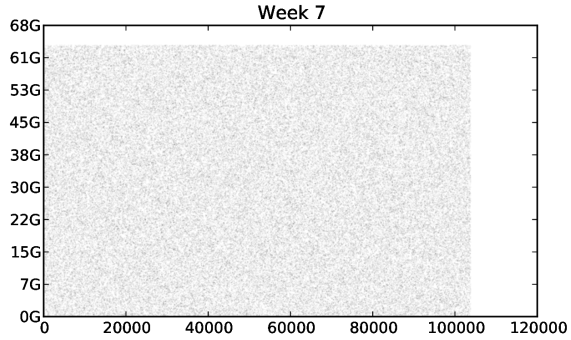


Figure 109: Bin fetch operations in week 7 (*UPB* dataset).

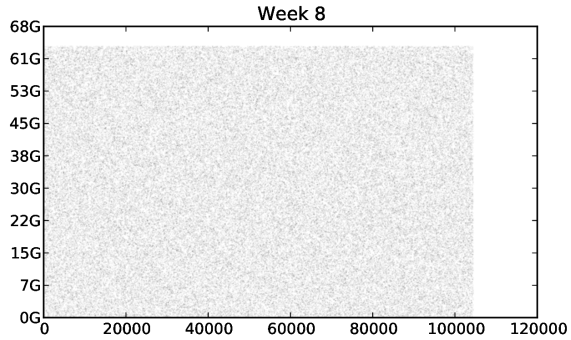


Figure 110: Bin fetch operations in week 8 (*UPB* dataset).

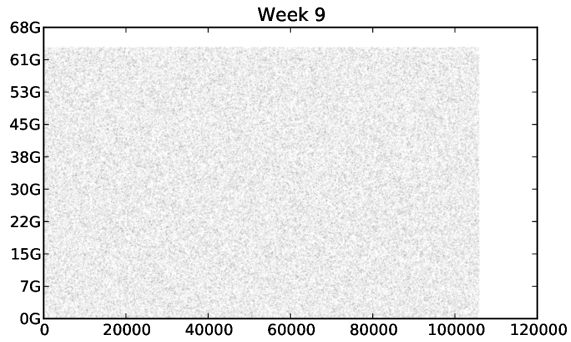


Figure 111: Bin fetch operations in week 9 (*UPB* dataset).

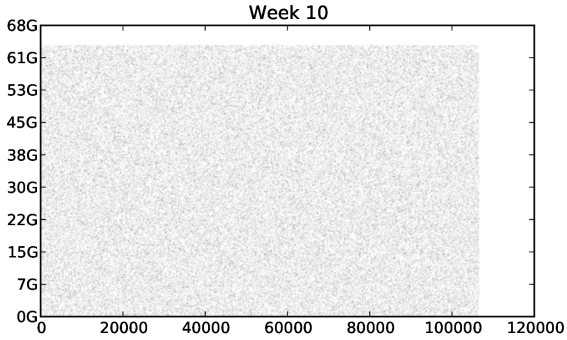


Figure 112: Bin fetch operations in week 10 (*UPB* dataset).

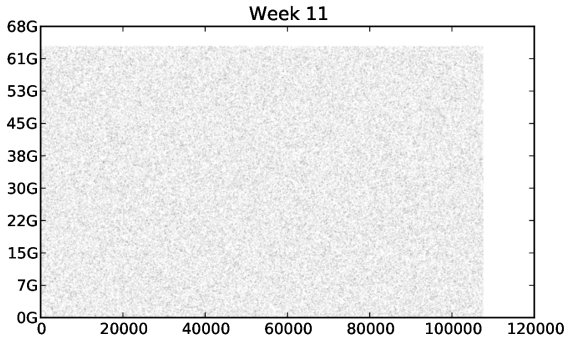


Figure 113: Bin fetch operations in week 11 (*UPB* dataset).

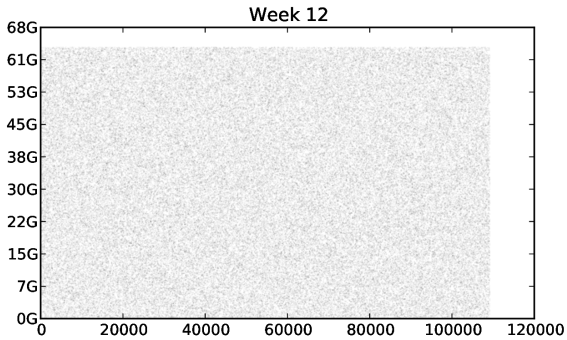


Figure 114: Bin fetch operations in week 12 (*UPB* dataset).

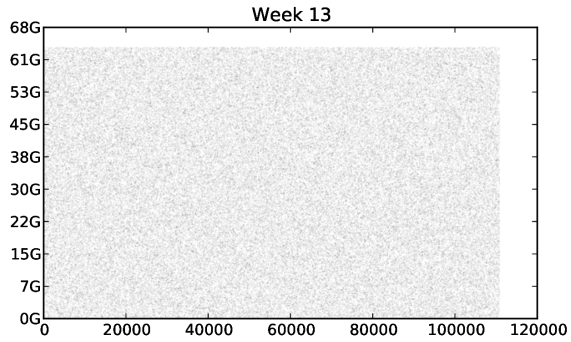


Figure 115: Bin fetch operations in week 13 (*UPB* dataset).

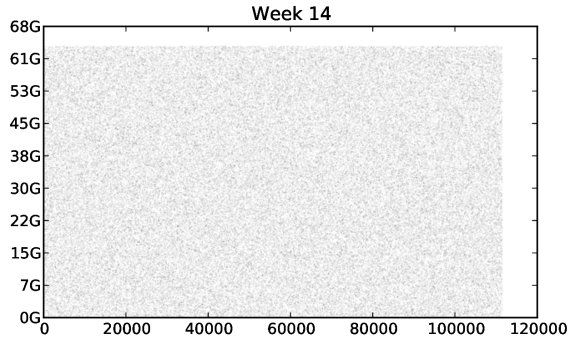


Figure 116: Bin fetch operations in week 14 (*UPB* dataset).

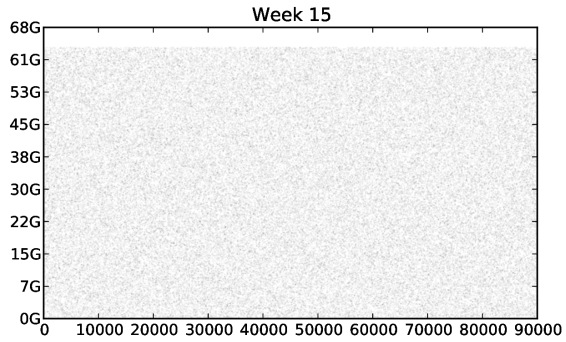


Figure 117: Bin fetch operations in week 15 (*UPB* dataset).

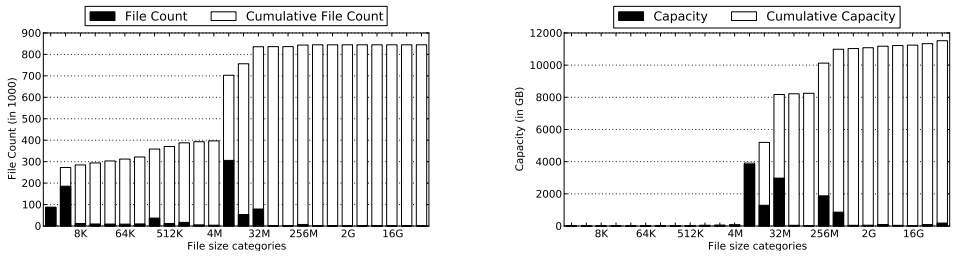
B

ADDITIONAL FIGURES: DATA DEDUPLICATION IN HPC STORAGE SYSTEMS

This appendix chapter contains additional figures that complete the part on the usage of data deduplication in HPC storage systems. In Section B.1, the filesystem metadata statistics are presented. In Section B.2, the data deduplication results are presented. Both sections also contain the figures which have been in Chapter 17 and Chapter 18 for completeness.

The explanation of the shown information can be found in Chapter 17 and Chapter 18.

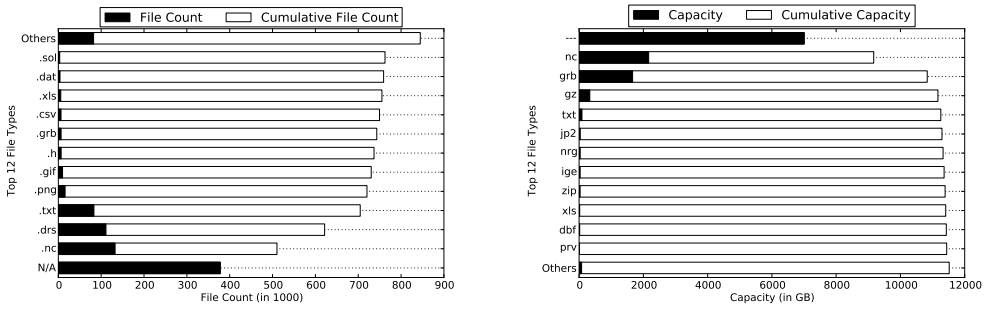
B.1 FILESYSTEM METADATA STATISTICS



(a) File Size Count

(b) File Size Capacity

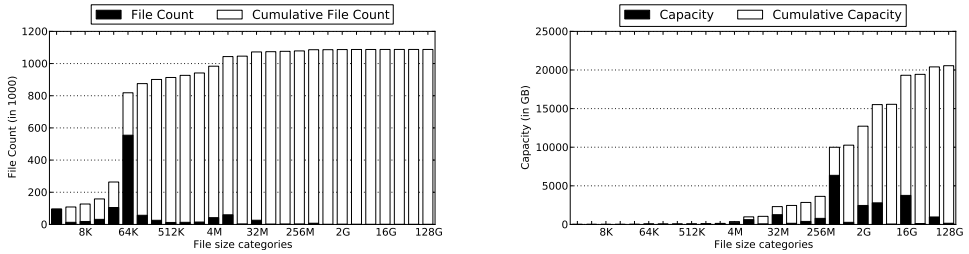
Figure 118: File statistics for the BSC-BD dataset



(a) File Type Count

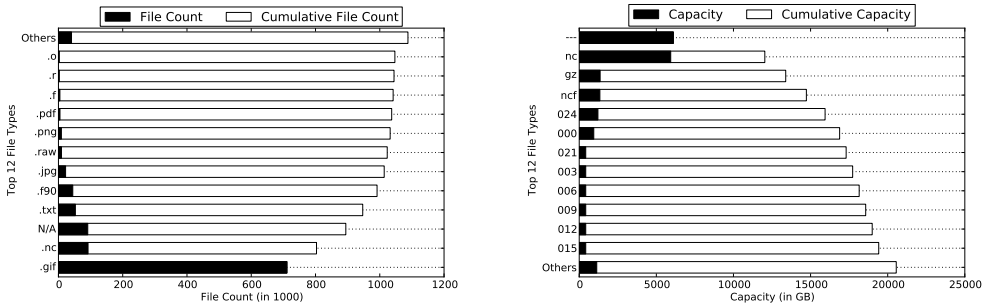
(b) File Type Capacity

Figure 119: File type statistics for the BSC-BD dataset



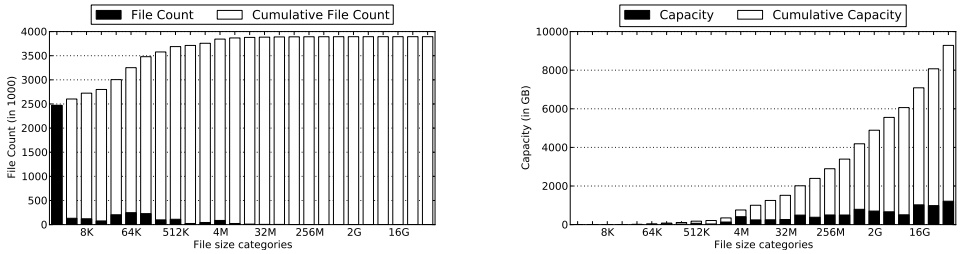
(a) File Size Count (b) File Size Capacity

Figure 120: File statistics for the BSC-MOD dataset



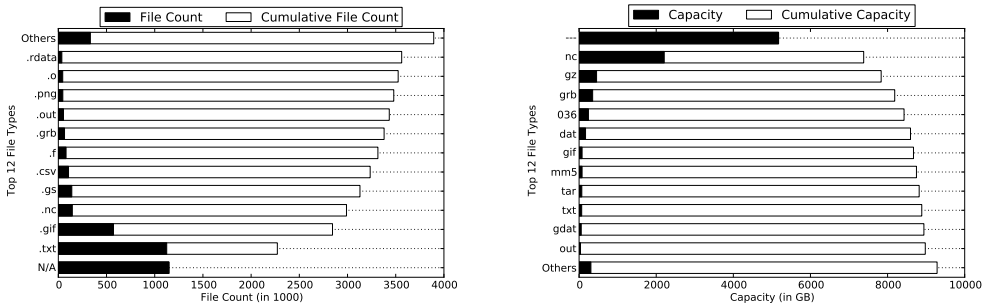
(a) File Type Count (b) File Type Capacity

Figure 121: File type statistics for the BSC-MOD dataset



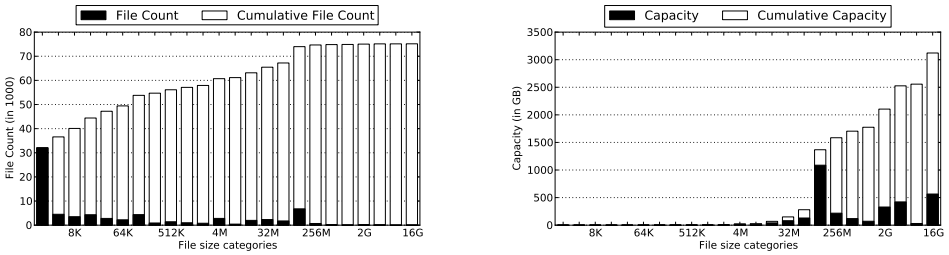
(a) File Size Count (b) File Size Capacity

Figure 122: File statistics for the BSC-PRO dataset



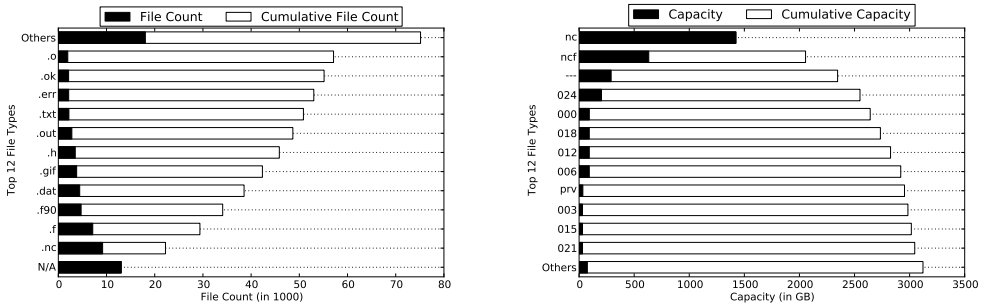
(a) File Type Count (b) File Type Capacity

Figure 123: File type statistics for the BSC-PRO dataset



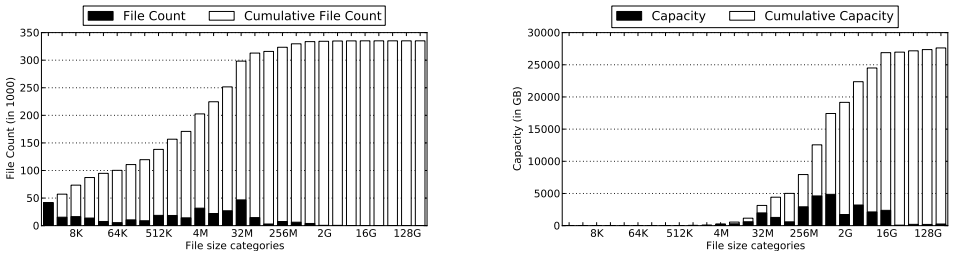
(a) File Size Count (b) File Size Capacity

Figure 124: File statistics for the BSC-SCRA dataset



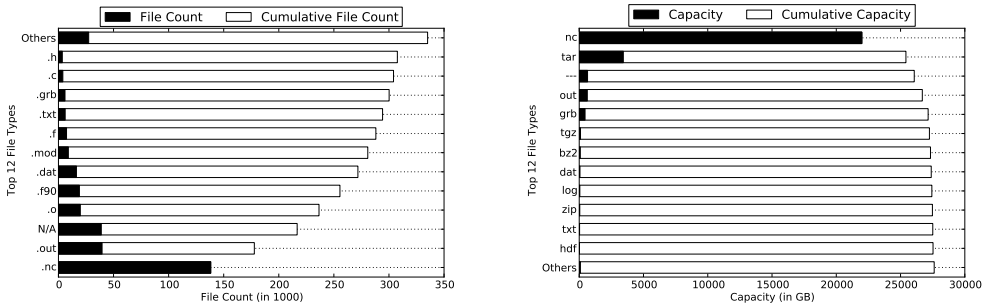
(a) File Type Count (b) File Type Capacity

Figure 125: File type statistics for the BSC-SCRA dataset



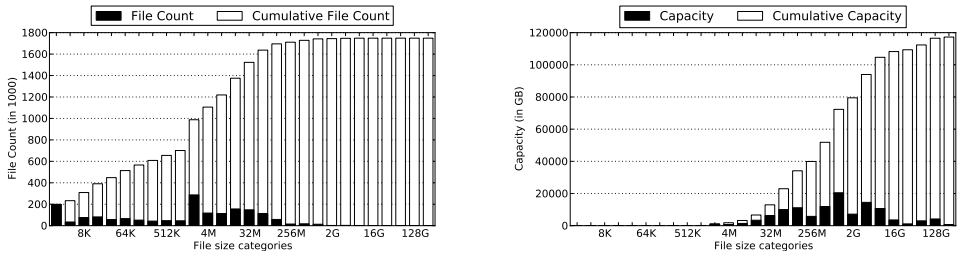
(a) File Size Count (b) File File Size Capacity

Figure 126: File statistics for the DKRZ-A dataset



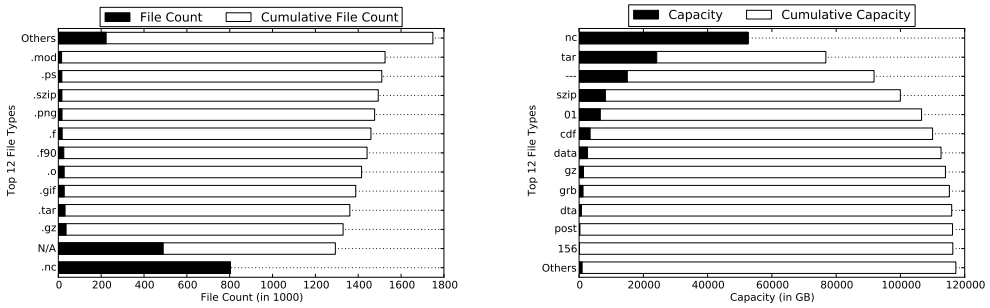
(a) File Type Count (b) File Type Capacity

Figure 127: File type statistics for the DKRZ-A dataset



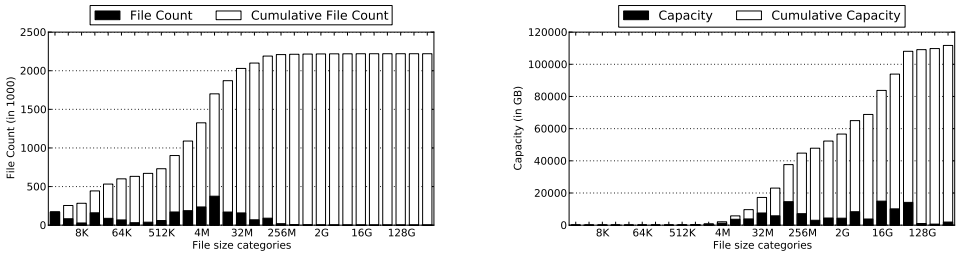
(a) File Size Count (b) File File Size Capacity

Figure 128: File statistics for the DKRZ-B1 dataset



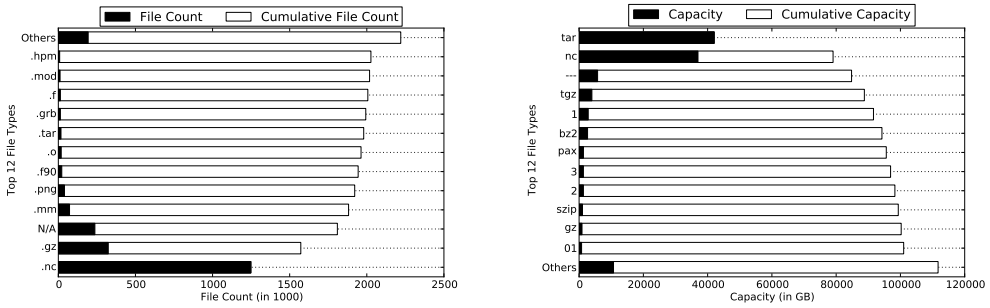
(a) File Type Count (b) File Type Capacity

Figure 129: File type statistics for the DKRZ-B1 dataset



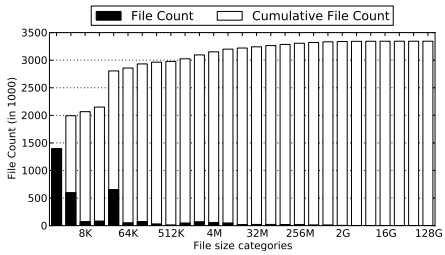
(a) File Size Count (b) File Size Capacity

Figure 130: File statistics for the DKRZ-B2 dataset

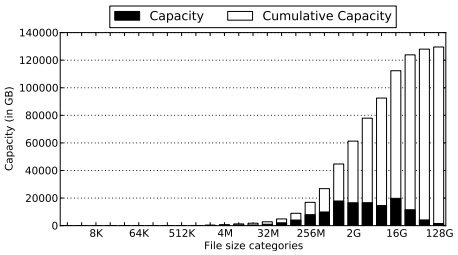


(a) File Type Count (b) File Type Capacity

Figure 131: File type statistics for the DKRZ-B2 dataset

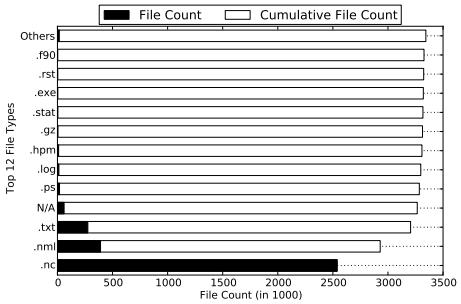


(a) File Size Count

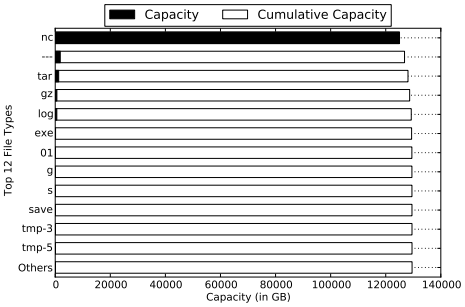


(b) File Size Capacity

Figure 132: File statistics for the DKRZ-B3 dataset

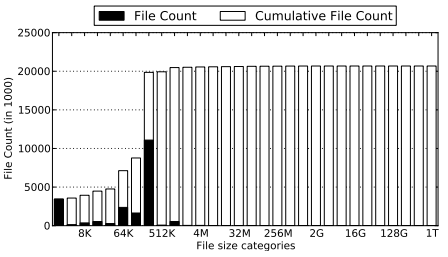


(a) File Type Count

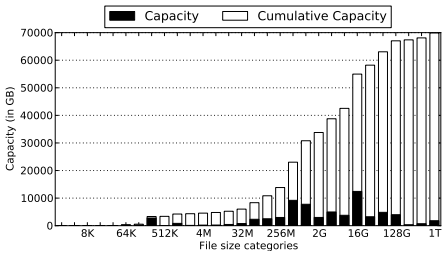


(b) File Type Capacity

Figure 133: File type statistics for the DKRZ-B3 dataset

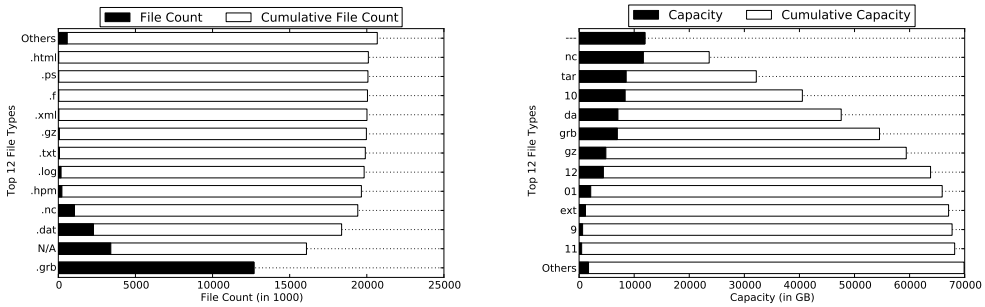


(a) File Size Count



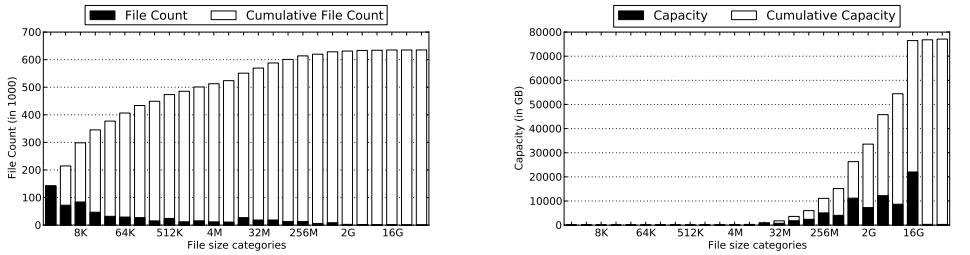
(b) File Size Capacity

Figure 134: File statistics for the DKRZ-B4 dataset



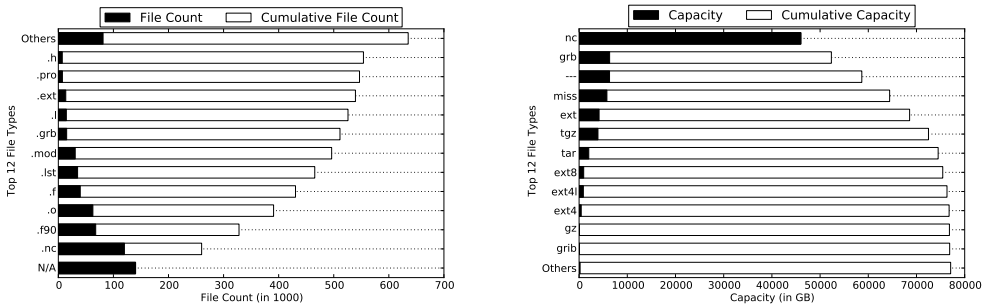
(a) File Type Count (b) File Type Capacity

Figure 135: File type statistics for the DKRZ-B4 dataset



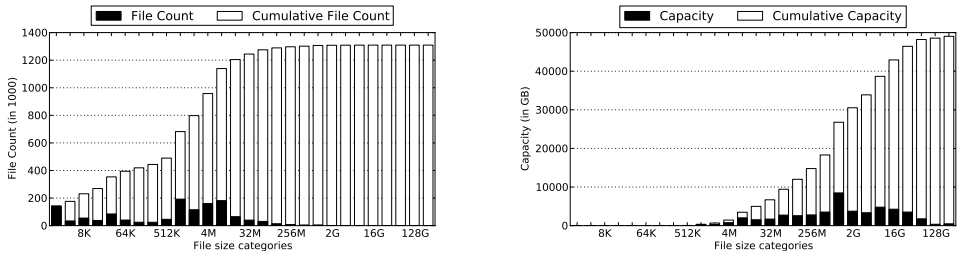
(a) File Size Count (b) File Size Capacity

Figure 136: File statistics for the DKRZ-B5 dataset



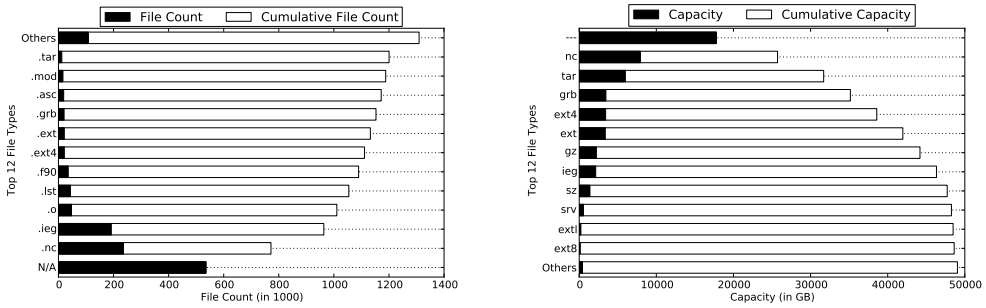
(a) File Type Count (b) File Type Capacity

Figure 137: File type statistics for the DKRZ-B5 dataset



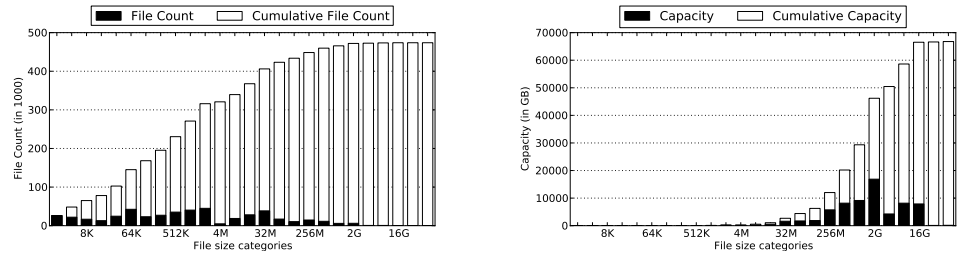
(a) File Size Count (b) File Size Capacity

Figure 138: File statistics for the DKRZ-B6 dataset



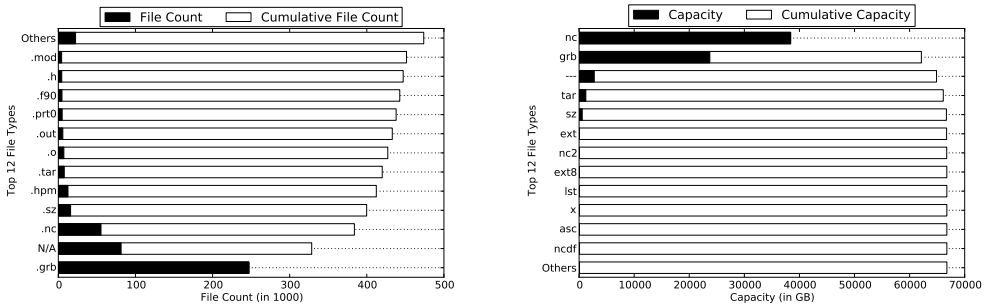
(a) File Type Count (b) File Type Capacity

Figure 139: File type statistics for the DKRZ-B6 dataset



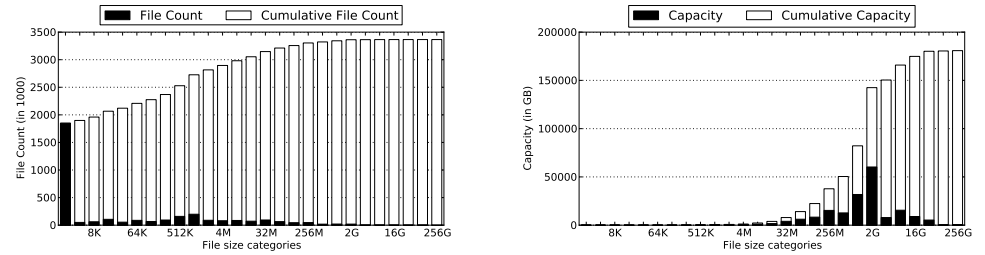
(a) File Size Count (b) File Size Capacity

Figure 140: File statistics for the DKRZ-B7 dataset



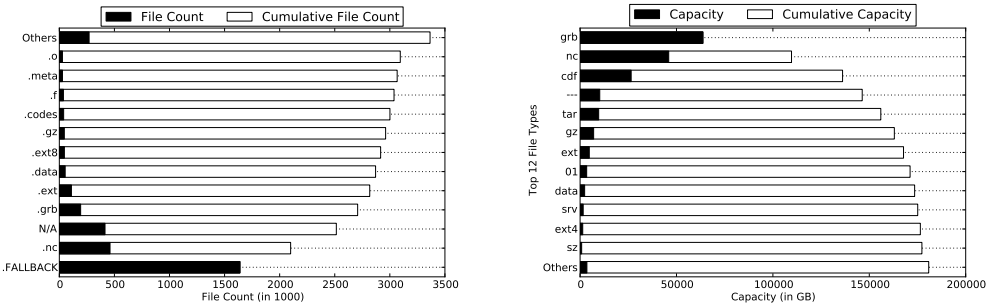
(a) File Type Count (b) File Type Capacity

Figure 141: File type statistics for the DKRZ-B7 dataset



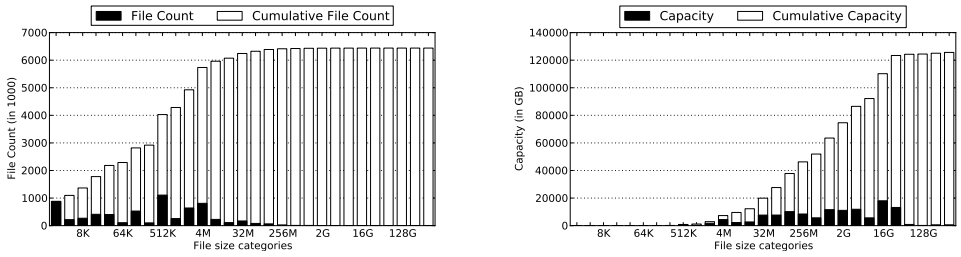
(a) File Size Count (b) File Size Capacity

Figure 142: File statistics for the DKRZ-B8 dataset



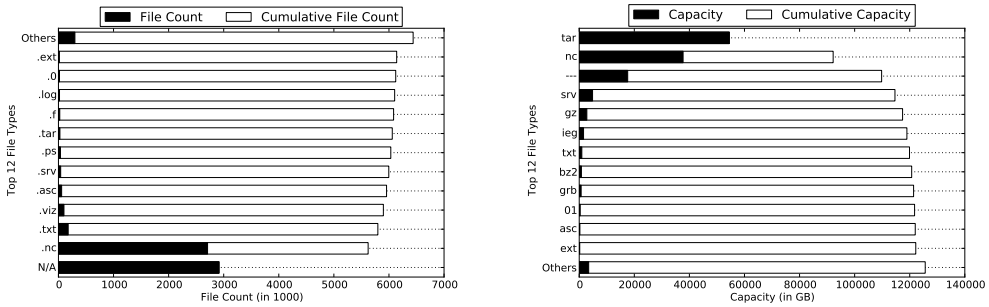
(a) File Type Count (b) File Type Capacity

Figure 143: File type statistics for the DKRZ-B8 dataset



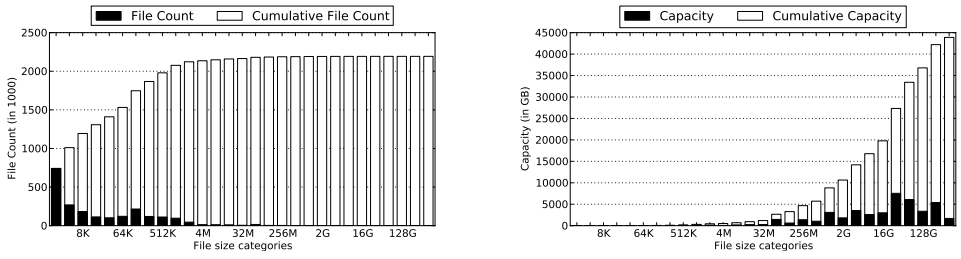
(a) File Size Count (b) File Size Capacity

Figure 144: File statistics for the DKRZ-C dataset



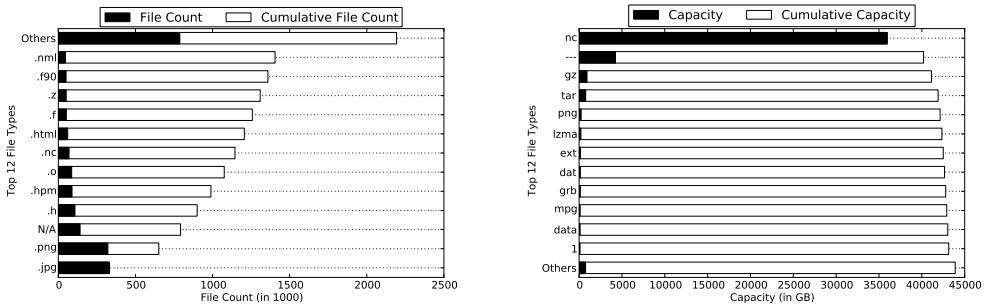
(a) File Type Count (b) File File Type Capacity

Figure 145: File type statistics for the DKRZ-C dataset



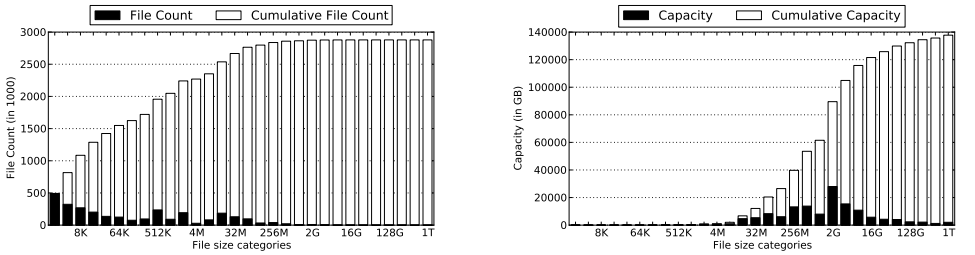
(a) File Size Count (b) File Size Capacity

Figure 146: File statistics for the DKRZ-K dataset



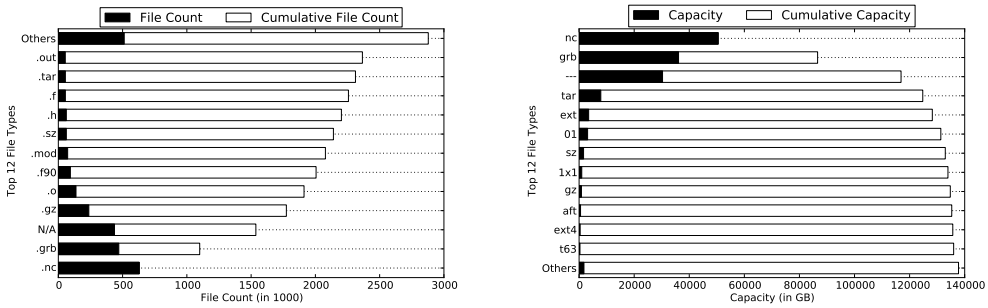
(a) File Type Count (b) File Type Capacity

Figure 147: File type statistics for the DKRZ-K dataset



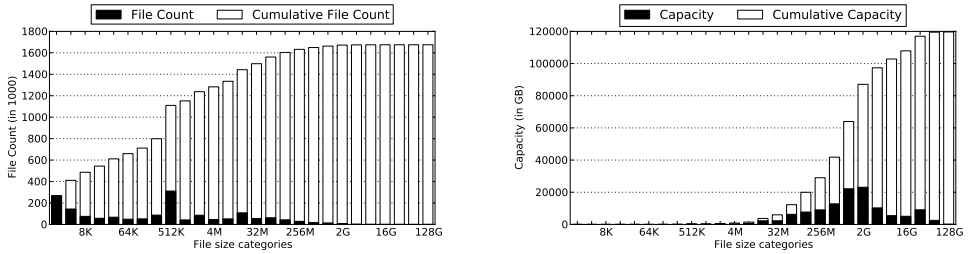
(a) File Size Count (b) File Size Capacity

Figure 148: File statistics for the DKRZ-M1 dataset



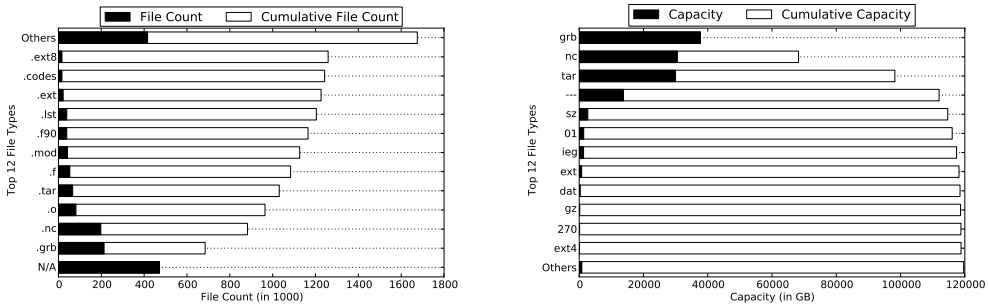
(a) File Type Count (b) File Type Capacity

Figure 149: File type statistics for the DKRZ-M1 dataset



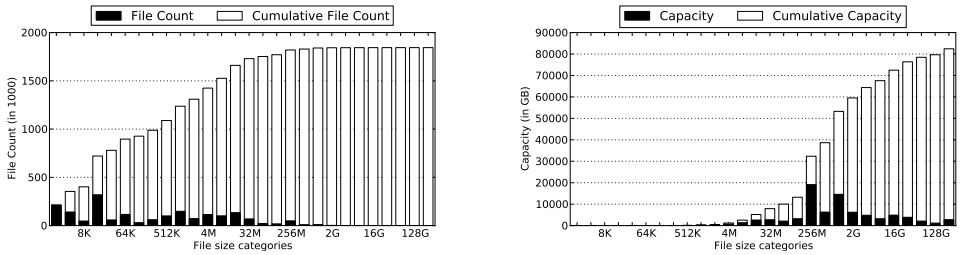
(a) File Size Count (b) File Size Capacity

Figure 150: File statistics for the DKRZ-M2 dataset



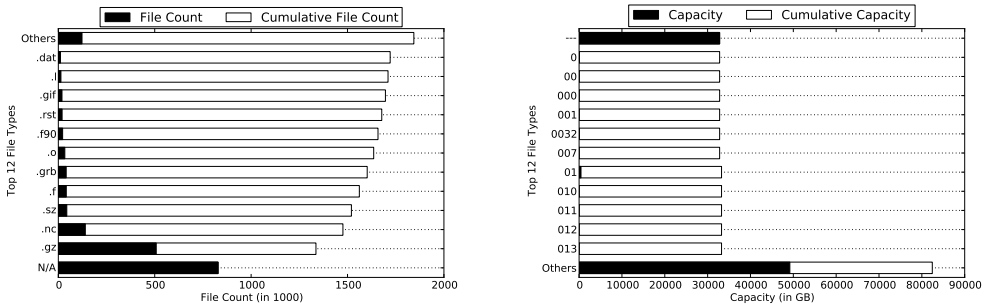
(a) File Type Count (b) File Type Capacity

Figure 151: File type statistics for the DKRZ-M2 dataset



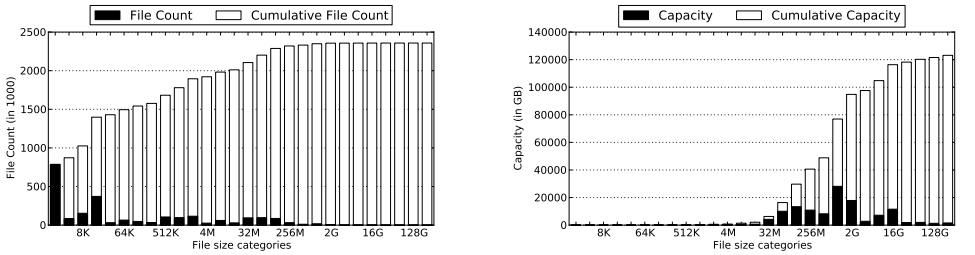
(a) File Size Count (b) File Size Capacity

Figure 152: File statistics for the DKRZ-M3 dataset



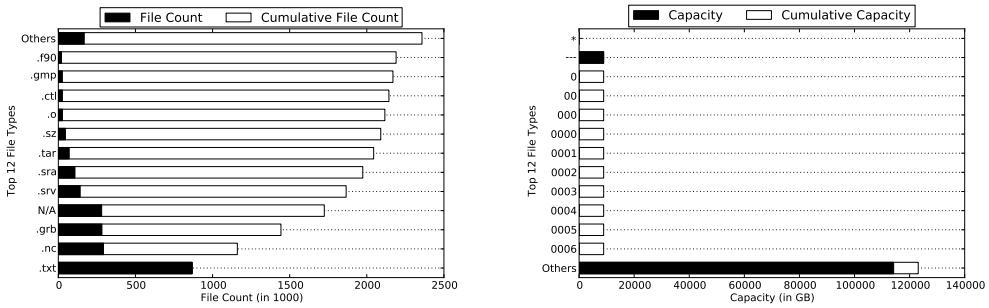
(a) File Type Count (b) File Type Capacity

Figure 153: File type statistics for the DKRZ-M3 dataset



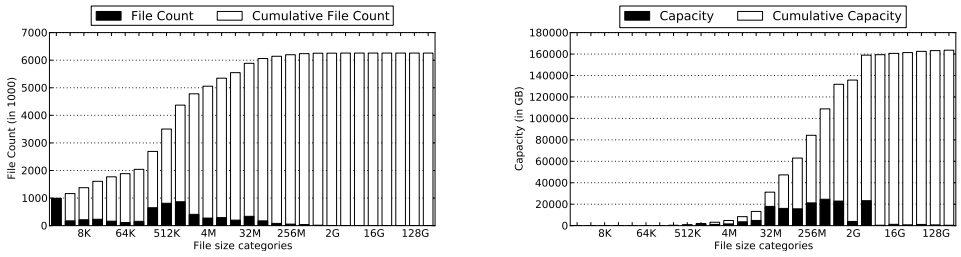
(a) File Size Count (b) File Size Capacity

Figure 154: File statistics for the DKRZ-M4 dataset



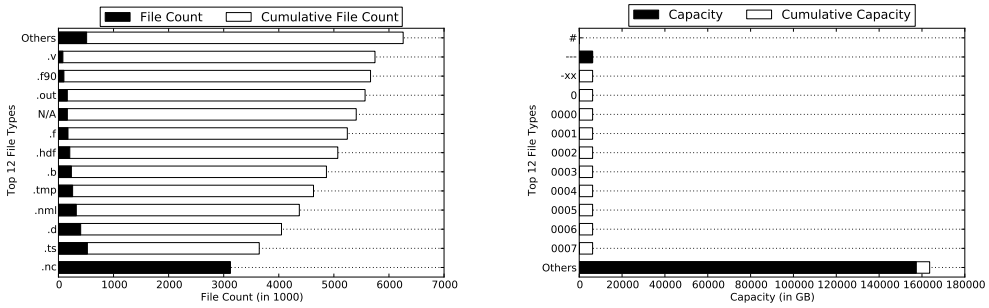
(a) File Type Count (b) File Type Capacity

Figure 155: File type statistics for the DKRZ-M4 dataset



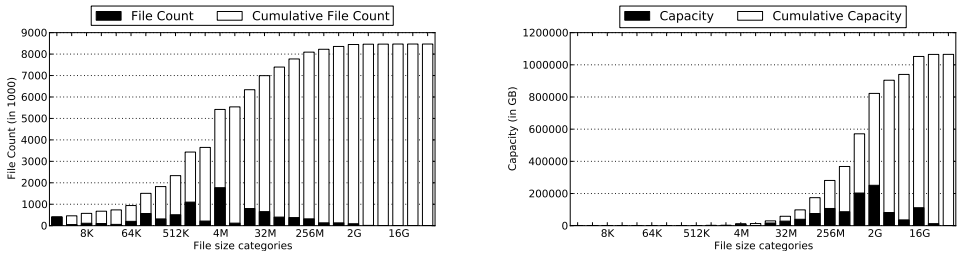
(a) File Size Count (b) File Size Capacity

Figure 156: File statistics for the DKRZ-M5 dataset



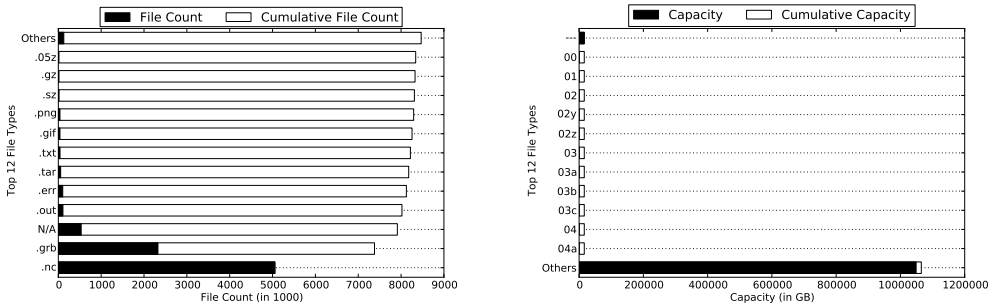
(a) File Type Count (b) File Type Capacity

Figure 157: File type statistics for the DKRZ-M5 dataset



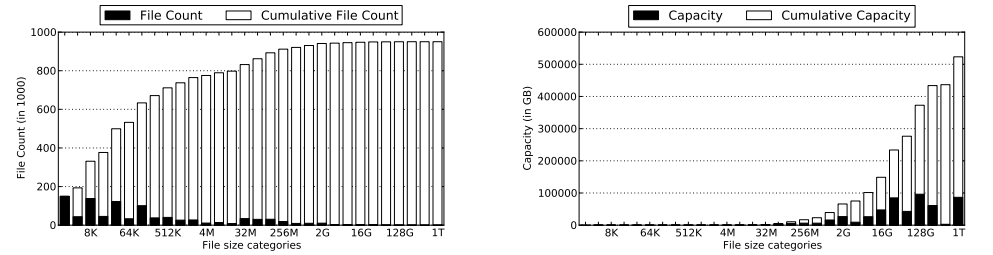
(a) File Size Count (b) File Size Capacity

Figure 158: File statistics for the DKRZ-I1 dataset



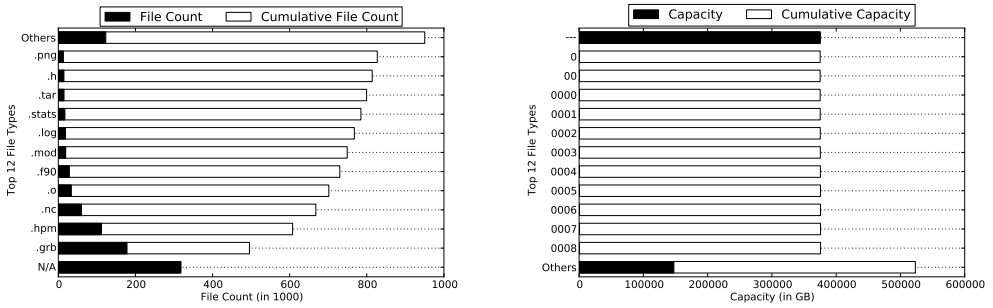
(a) File Type Count (b) File Type Capacity

Figure 159: File type statistics for the DKRZ-I1 dataset



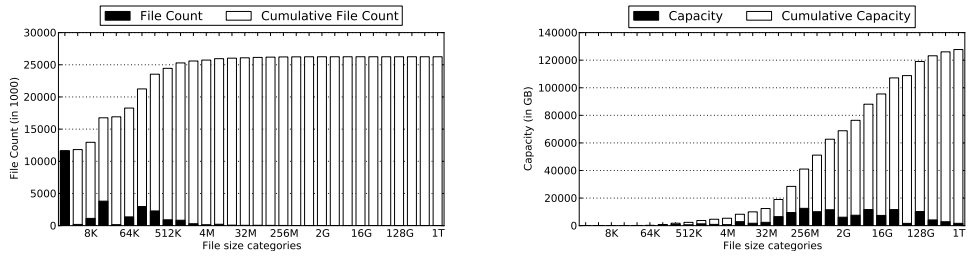
(a) File Size Count (b) File Size Capacity

Figure 160: File statistics for the DKRZ-I2 dataset



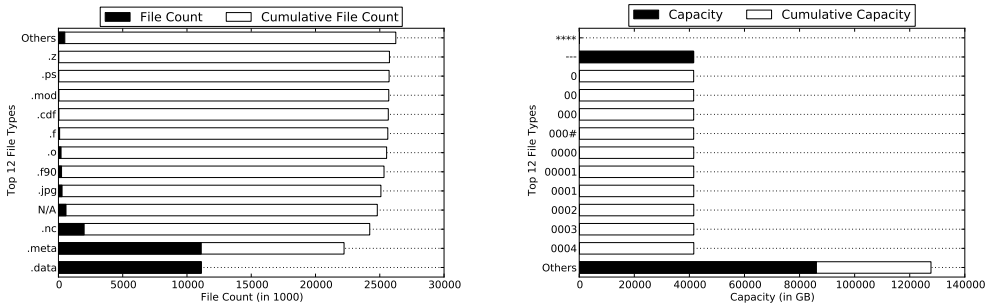
(a) File Type Count (b) File Type Capacity

Figure 161: File type statistics for the DKRZ-I2 dataset



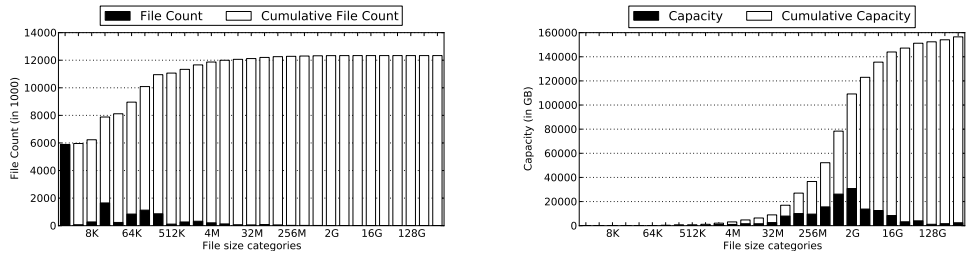
(a) File Size Count (b) File Size Capacity

Figure 162: File statistics for the DKRZ-U1 dataset



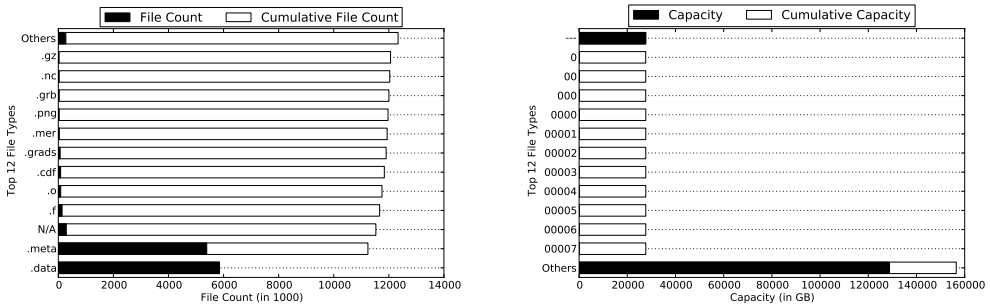
(a) File Type Count (b) File Type Capacity

Figure 163: File type statistics for the DKRZ-U1 dataset



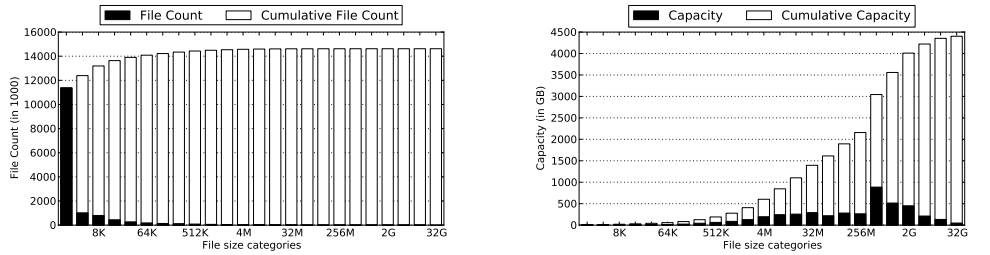
(a) File Size Count (b) File Size Capacity

Figure 164: File statistics for the DKRZ-U2 dataset



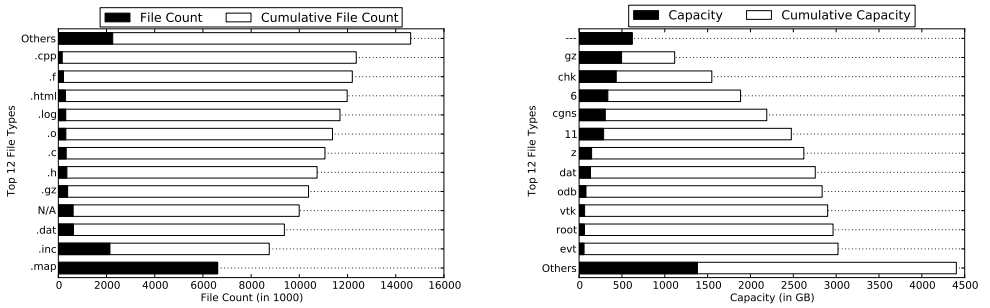
(a) File Type Count (b) File Type Capacity

Figure 165: File type statistics for the DKRZ-U2 dataset



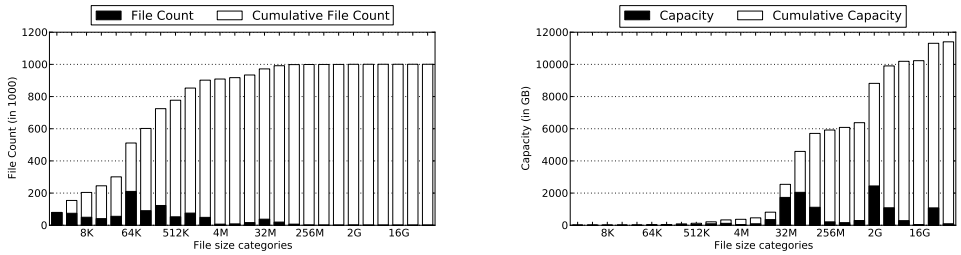
(a) File Size Count (b) File Size Capacity

Figure 166: File statistics for the RWTH dataset



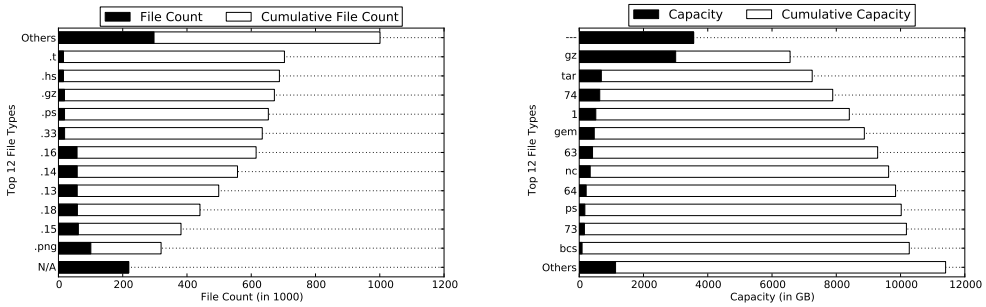
(a) File Type Count (b) File Type Capacity

Figure 167: File type statistics for the RWTH dataset



(a) File Size Count (b) File Size Capacity

Figure 168: File statistics for the RENCI dataset



(a) File Type Count (b) File Type Capacity

Figure 169: File type statistics for the RENCI dataset

B.2 DEDUPLICATION RESULTS

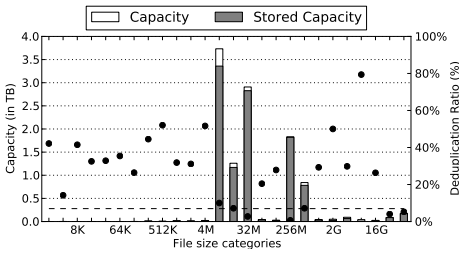


Figure 170: Deduplication results grouped by file size categories, BSC-BD data set

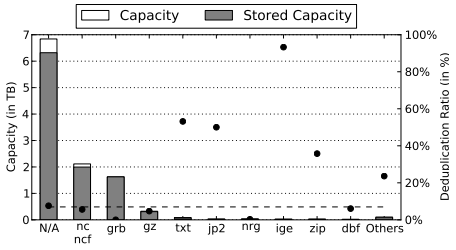


Figure 171: Deduplication results grouped by file types, BSC-BD data set

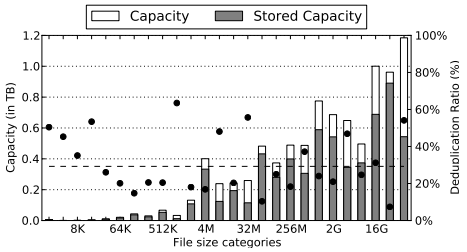


Figure 172: Deduplication results grouped by file size categories, BSC-PRO data set

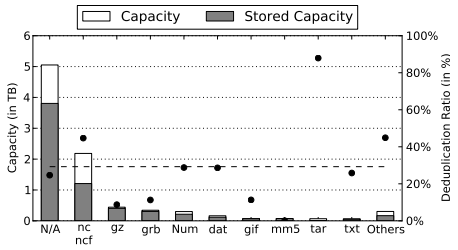


Figure 173: Deduplication results grouped by file types, BSC-PRO data set

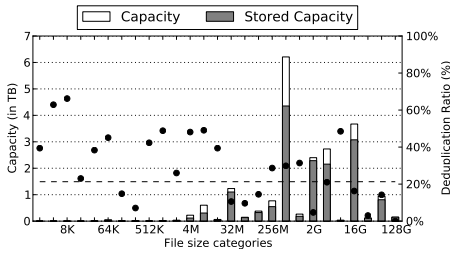


Figure 174: Deduplication results grouped by file size categories, BSC-MOD data set

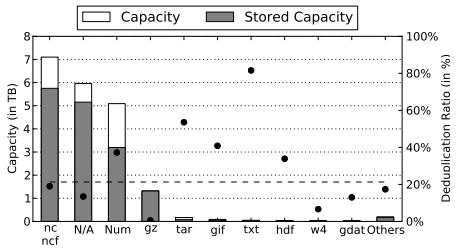


Figure 175: Deduplication results grouped by file types, BSC-MOD data set

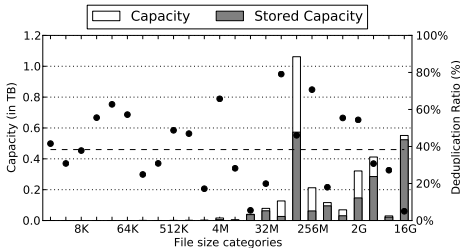


Figure 176: Deduplication results grouped by file size categories, BSC-SCRA data set

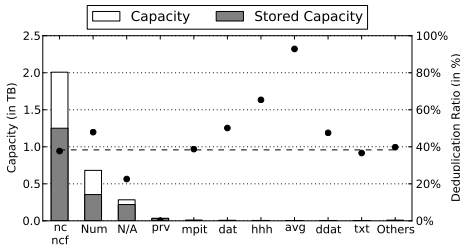


Figure 177: Deduplication results grouped by file types, BSC-SCRA data set

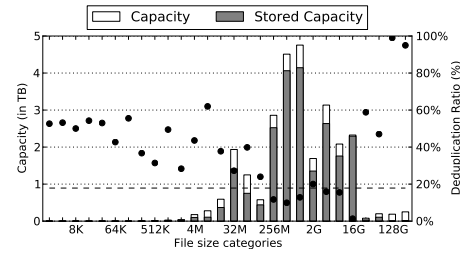


Figure 178: Deduplication results grouped by file size categories, DKRZ-A data set

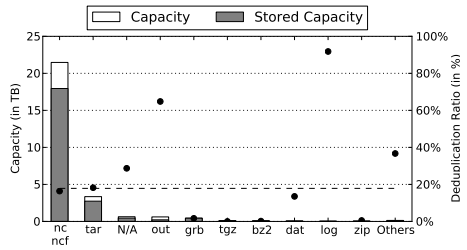


Figure 179: Deduplication results grouped by file types, DKRZ-A data set

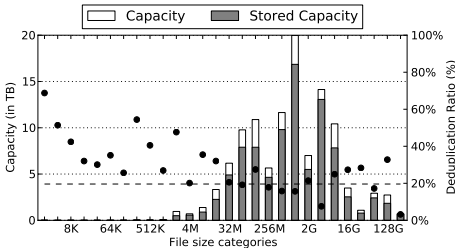


Figure 180: Deduplication results grouped by file size categories, DKRZ-B1 data set

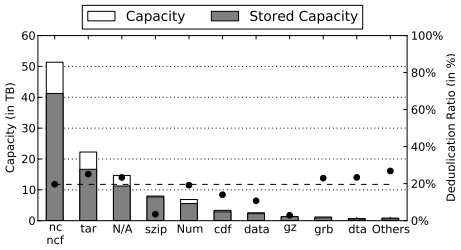


Figure 181: Deduplication results grouped by file types, DKRZ-B1 data set

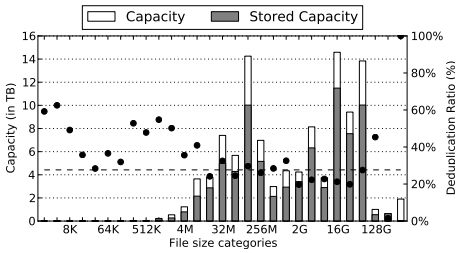


Figure 182: Deduplication results grouped by file size categories, DKRZ-B2 data set

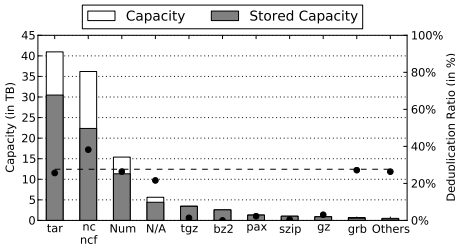


Figure 183: Deduplication results grouped by file types, DKRZ-B2 data set

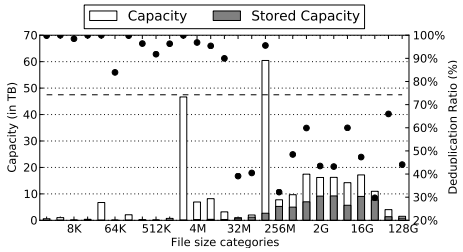


Figure 184: Deduplication results grouped by file size categories, DKRZ-B3 data set

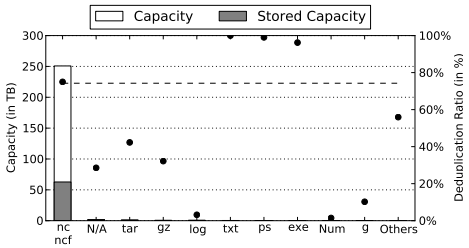


Figure 185: Deduplication results grouped by file types, DKRZ-B3 data set

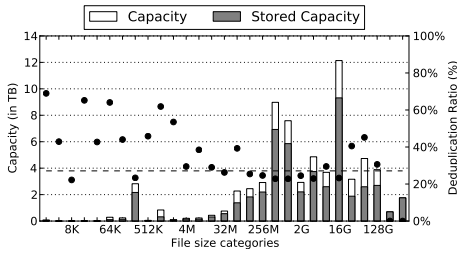


Figure 186: Deduplication results grouped by file size categories, DKRZ-B4 data set

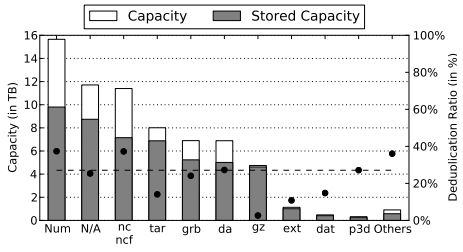


Figure 187: Deduplication results grouped by file types, DKRZ-B4 data set

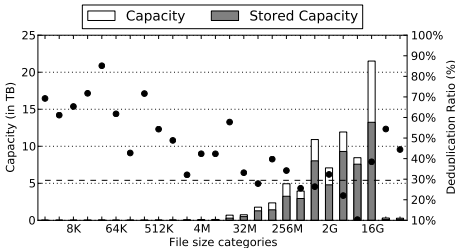


Figure 188: Deduplication results grouped by file size categories, DKRZ-B5 data set

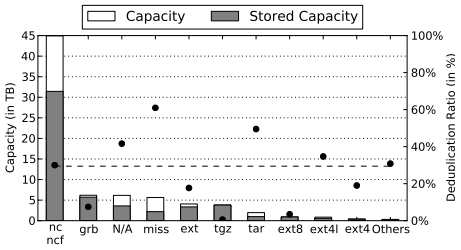


Figure 189: Deduplication results grouped by file types, DKRZ-B5 data set

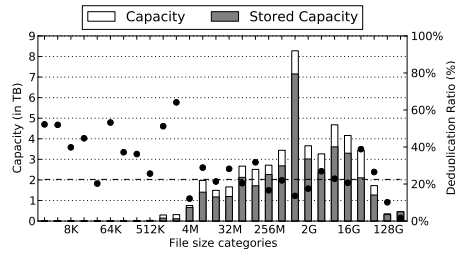


Figure 190: Deduplication results grouped by file size categories, DKRZ-B6 data set

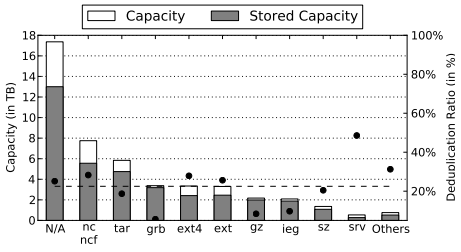


Figure 191: Deduplication results grouped by file types, DKRZ-B6 data set

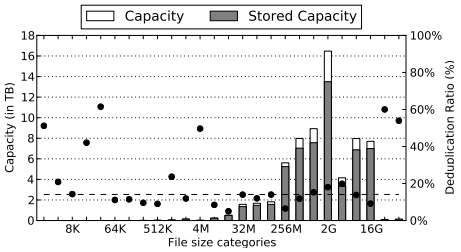


Figure 192: Deduplication results grouped by file size categories, DKRZ-B7 data set

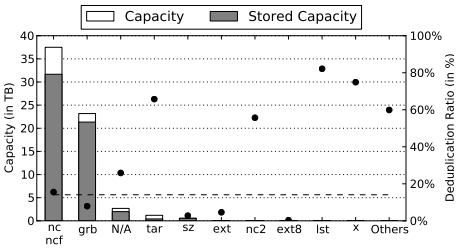


Figure 193: Deduplication results grouped by file types, DKRZ-B7 data set

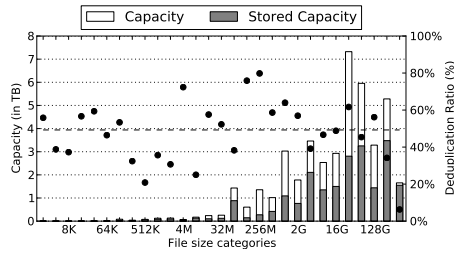


Figure 194: Deduplication results grouped by file size categories, DKRZ-K data set

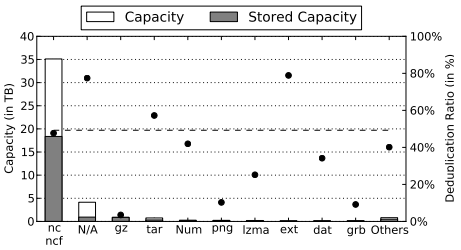


Figure 195: Deduplication results grouped by file types, DKRZ-K data set

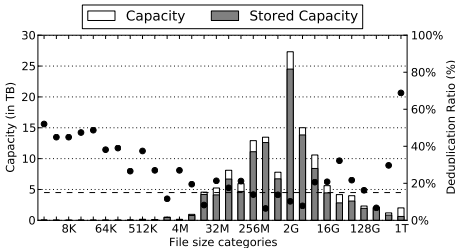


Figure 196: Deduplication results grouped by file size categories, DKRZ-M1 data set

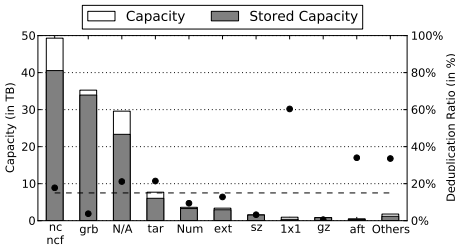


Figure 197: Deduplication results grouped by file types, DKRZ-M1 data set

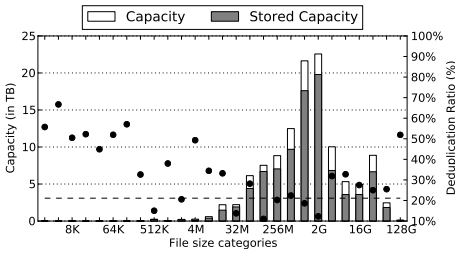


Figure 198: Deduplication results grouped by file size categories, DKRZ-M2 data set

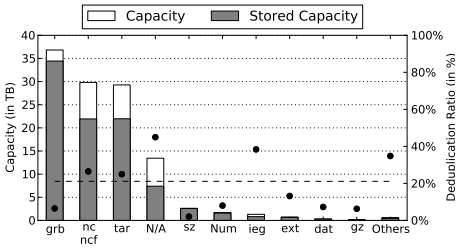


Figure 199: Deduplication results grouped by file types, DKRZ-M2 data set

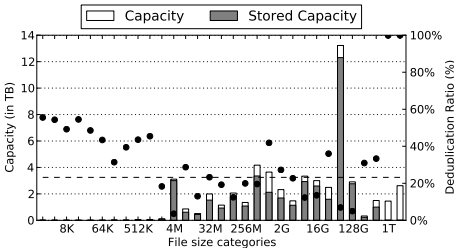


Figure 200: Deduplication results grouped by file size categories, RWTH data set

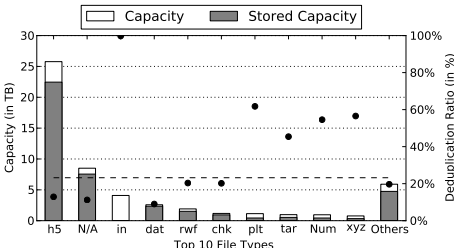


Figure 201: Deduplication results grouped by file types, RWTH data set

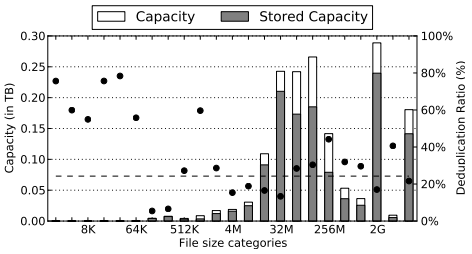


Figure 202: Deduplication results grouped by file size categories, RENCI data set

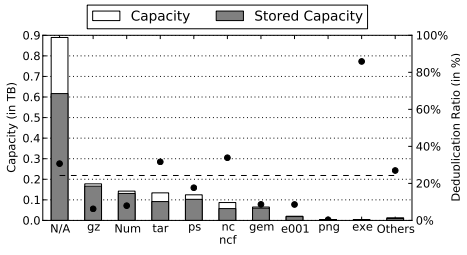


Figure 203: Deduplication results grouped by file types, RENCI data set

C

FS-C TRACE DATA AND SOURCE CODE

In this chapter, I will describe how to receive the trace data and the major source code used in this thesis.

The FS-C data sets *UPB*, *JGU*, and *ENG* data sets as well as all HPC data sets are archived ZDV department of the Johannes Gutenberg University Mainz. The FS-C data sets have a total size of more than 4 TB.

The data sets need to be considered confidential unless notes otherwise. The DKRZ has granted permission to release the data sets not starting with the prefix work_mh to other researchers for research purposes. The trace files can be received on request to the ZDV department.

The source code used for the experiments and simulations is archived by the ZDV department. Parts of the used software has also been published as open source. The following table states, which software has been used, the corresponding git revision, and if and where the software is available.

FS-C: The FS-C tool suite is available `https://github.com/dmeister/fs-c`. The most recent version at time of the thesis is 0.3.13 corresponding the revision `d45a4ec73869b4cf9448e728c3c8164d00f8fcfd`.

DEDUPV1: The dedupv1 system is available at `https://github.com/dedupv1/dedupv1`. The public version does not include the compression and BLC prototype.

The BLC prototype is available at request from the ZDV department. The revision used for the evaluation presented in Part III is `e7795c5414405443216d8b4157c032ad70d85bad`.

WORKLOAD GENERATION: The workload generation tool is available at `https://github.com/dmeister/dedupbenchmark`. The version used in the evaluation has the revision id `9ec2c66a70f754f2ad7b630b17531ac55cbdb705`.

DI-DEDUPV1: The di-dedupv1 prototype used in the evaluation of Part II is not publicly available. It can be requested by a interested party from the ZDV department. The version used in the evaluation has the revision id `aaec75a431d11c11d29e6e975361-7472669de1bf`.

SIMULATIONS OF RECIPE COMPRESSION AND BLC: The trace-based simulation of block and file recipe compression and of the Block Locality Cache are available at `https://github.com/dmeister/fs-c-analytics`.

The released version has been commented and slightly refactor-ed version of the software used in the evaluation. The version with revision id `cfbbc994dc82f91b8d-4b6a66a69d0300509ccfb5` can be used to repeat the evaluation presented in Part III.

I can not make long-term guarantees on the availability of public links.

BIBLIOGRAPHY

[AAB+09] Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel T. Klein. The design of a similarity based deduplication system. In *Proceedings of the 2nd Israeli Experimental Systems Conference (SYSTOR)*, pages 6:1–6:14. ACM, 2009.

[AAH+12] Lior Aronovich, Ron Asher, Danny Harnik, Michael Hirsch, Shmuel T. Klein, and Yair Toaff. Similarity based deduplication with small data chunks. In *Proceedings of the Prague Stringology Conference 2012*, page 3, 2012.

[ABDL07] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS)*, 3(3), 2007.

[ABF+02] Miklos Ajtai, Randal Burns, Ronald Fagin, Darrell DE Long, and Larry Stockmeyer. Compactly encoding unstructured inputs with differential compression. *Journal of the ACM (JACM)*, 49(3):318–367, 2002.

[Aga05] Pratul K. Agarwal. Role of protein dynamics in reaction rate enhancement by enzymes. *Journal of the American Chemical Society*, 127(43):15248–15256, 2005.

[AGM+90] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403 – 410, 1990.

[Alv11] Carloz Alvarez. NetApp technical report: NetApp deduplication for FAS and V-Series deployment and implementation guide. NetApp Technical Report TR-3505, Version 8, February 2011.

[AMF06] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. ACM, 2006.

[Apa] Apache Software Foundation. Hadoop. `http://hadoop.apache.org/`.

[ASM12] Ian Adams, Mark W. Storer, and Ethan L. Miller. Analysis of workload behavior in scientific and historical long-term data repositories. *ACM Transactions on Storage (TOS)*, 8(2):6, 2012.

[BADAD+08] Lakshmi N Bairavasundaram, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Garth R Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. *ACM Transactions on Storage (TOS)*, 4(3):8, 2008.

[BC08] Subashini Balachandran and Cornel Constantinescu. Sequence of hashes compression in data de-duplication. In *Proceedings of the Data Compression Conference (DCC)*, page 505. IEEE, 2008.

[BCGD00] William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. Single instance storage in Windows 2000. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium*. USENIX, 2000.

[BDGM95] Sergey Brin, James Davis, and Hector Garcia-Molina. Copy detection mechanisms for digital documents. In *ACM SIGMOD Record*, volume 24, pages 398–409. ACM, 1995.

[BE07] André Brinkmann and Sascha Effert. Inter-node communication in Peer-to-Peer storage clusters. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies (MSST)*. IEEE, 2007.

[BELL09] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, pages 1–9. IEEE, 2009.

[BGG+09] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, page 21. ACM, 2009.

[BGKM11] André Brinkmann, Yan Gao, Miroslaw Korzeniowski, and Dirk Meister. Request load balancing for highly skewed traffic in P2P networks. In *Proceedings of the 6th IEEE International Conference on Networking, Architecture, and Storage (NAS)*. IEEE, 2011.

[BHM+04] André Brinkmann, Michael Heidebuer, Friedhelm Meyer auf der Heide, Ulrich Rückert, Kay Salzwedel, and Mario Vodisek. V:Drive - costs and benefits of an out-of-band storage virtualization system. In *Proceedings of the 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST)*. IEEE, 2004.

[Big07] Heidi Biggar. Experiencing data de-duplication: Improving efficiency and reducing capacity requirements. White paper February, The Enterprise Strategy Group, 2007.

[Bla06] John Black. Compare-by-hash: a reasoned analysis. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 85–90. USENIX, 2006.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BM72] Rudolf Bayer and Edward Meyers McCreight. Organization and maintenance of large ordered indexes. *Acta informatica*, 1(3):173–189, 1972.

[BM99] Jon Bentley and Douglas McIlroy. Data compression using long common strings. In *Proceedings of the Conference on Data Compression Conference (DCC)*, pages 287–295. IEEE, 1999.

[Bol12] Vladislav Bolkhovitin. SCST: Generic SCSI target subsystem for linux. http://scst.sourceforge.net/, 2012.

[Bra02] Peter Braam. Lustre: A scalable, high performance file system. Available at http://www.lustre.org/docs/whitepaper.pdf, 2002.

[Bra11] Peter Braam. Bridging the peta-to exa-scale i/o gap. Keynote talk at IEEE Conference on Mass Storage Systems and Technologies (MSST), May 2011.

[Bro93] Andre Z. Broder. Some applications of rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science*, volume 993, page 143152, 1993.

[Bro97] Andre Z. Broder. On the resemblance and containment of documents. In *Proceedings of the Conference on Compression and Complexity of Sequences*, pages 21–29. IEEE, 1997.

[Bro00] Andre Z. Broder. Identifying and filtering near-duplicate documents. In *Combinatorial Pattern Matching*, pages 1–10. Springer, 2000.

[BSGH13] Fabiano C. Botelho, Philip Shilane, Nitin Garg, and Windsor Hsu. Memory efficient sanitization of a deduplicated storage system. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, February 2013.

[CAVL09] Austin T. Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized deduplication in san cluster file systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX, 2009.

[CBT07] Mingming Cao, Suparna Bhattacharya, and Ted Tso. Ext4: The next generation of ext2/3 filesystem. In *Proceedings of the 2007 Linux Storage & File System Workshop*. USENIX, 2007.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[CGC11] Cornel Constantinescu, Joseph Glider, and David Chambliss. Mixing deduplication and compression on active data sets. In *Data Compression Conference (DCC), 2011*, pages 393–402. IEEE, 2011.

[CHA+11] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage (TOS)*, 7(3):8, 2011.

[CK06] J. Chu and V. Kashyap. RFC 4391: Transmission of ip over infiniband (IPoIB). Technical report, Internet Engineering Task Force - Network Working Group, April 2006.

[CL01] Calvin Chan and Hahua Lu. Fingerprinting using polynomial (rabin's method). *Faculty of Science, University of Alberta, CMPUT690 Term Project*, 2001.

[CLI+00] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference (ALS)*, pages 317–327. USENIX, 2000.

[CMN02] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy. *ACM SIGOPS Operating Systems Review*, 36(SI):285–298, 2002.

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, December 1960.

[CPL09] Cornel Constantinescu, Jan Pieper, and Tiancheng Li. Block size optimization in deduplication systems. In *Proceedings of the Data Compression Conference*, 2009.

[CTA02] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(5):561–580, 2002.

[Day08] Shobhit Dayal. Characterizing HEC storage systems at rest. Technical report, Carnegie Mellon University Parallel Data Lab, 2008.

[DB99] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM, 1999.

[DBQS11] Fred Douglis, Deepti Bhardwaj, Hangwei Qian, and Philip Shilane. Content-aware load balancing for distributed backup. In *Proceedings of the 25th Large Installation System Administration Conference (LISA)*. USENIX, 2011.

[DCR08] Christophe De Canniere and Christian Rechberger. Preimages for reduced sha-0 and sha-1. *Advances in Cryptology–CRYPTO 2008*, pages 179–202, 2008.

[DDL+11] Wei Dong, Fred Douglis, Kai Li, Hugo Patterson, Sazzala Reddy, and Philip Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 2011.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI)*. USENIX, 2004.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[DG11] Jeffrey Dean and Sanjay Ghemawat. LevelDB: A fast and leightweight key/-value database library by google. `https://code.google.com/p/leveldb/`, 2011.

[DGH+09] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: A scalable secondary storage. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 2009.

[DI03] Fred Douglis and Arun Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 113–126. USENIX, 2003.

[DKM96] Murthy Devarakonda, Bill Kish, and Ajay Mohindra. Recovery in the calypso file system. *ACM Trans. Comput. Syst.*, 14, 1996.

[DSL10] Biblob Debnath, Sudipta Sengupta, and Jin Li. Chunkstash: speeding up inline storage deduplication using flash memory. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 16–16. USENIX, 2010.

[DUK04] Cezary Dubnicki, Cristian Ungureanu, and Wojciech Kilian. FPN: A distributed hash table for commercial applications. In *Proceedings of the 13th IEEE International Symposium on High performance Distributed Computing*, pages 120–128. IEEE, 2004.

[Dut08] Mike Dutch. Understanding data deduplication ratios. In *SNIA Data Management Forum*, 2008.

[DWG11] Umesh Deshpande, Xiaoshuang Wang, and Kartik Gopalan. Live gang migration of virtual machines. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC)*, pages 135–146. ACM, 2011.

[Eff11] Sascha Effert. *Verfahren zur redundanten Datenplatzierung in skalierbaren Speichernetzen*. PhD thesis, Heinz Nixdorf Institut und Institut für Informatik, University of Paderborn, June 2011.

[EG10] Petros Efstathopoulos and Fanglu Guo. Rethinking deduplication scalability. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*. USENIX, 2010.

[EKJP10] Kaoutar El Maghraoui, Gokul Kandiraju, Joefon Jann, and Pratap Pattnaik. Modeling and simulating flash based solid-state disks for operating systems. In *Proceedings of the 1st joint WOSP/SIPEW International Conference on Performance Engineering*, pages 15–26. ACM, 2010.

[ELW+07] Kave Eshghi, Mark Lillibridge, Lawrence Wilcock, Guillaume Belrose, and Rycharde Hawkes. Jumbo store: Providing efficient incremental upload and versioning for a utility rendering service. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 2007.

[EMC12] EMC Cooperation. Data domain boost software. http://www.emc.com/collateral/software/data-sheet/h7034-datadomain-boost-sw-ds.pdf, 2012.

[ET05] Kave Eshghi and Husi Khuern Tang. A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR*, 30, 2005.

[Eva10] Mark Evans. SCSI block command 3 (SBC-3) revision 24. T10 Committee of the INCITS, 2010.

[FEC05] George Forman, Kave Eshghi, and Stephane Chiocchetti. Finding similar files in large document repositories. In *Proceedings of the 11th International Conference on Knowledge Discovery in Data Mining (SIGKDD)*, pages 394–400. ACM, 2005.

[FZJZ08] Dan Feng, Qiang Zou, Hong Jiang, and Yifeng Zhu. A novel model for synthesizing parallel i/o workloads in scientific applications. In *IEEE International Conference on Cluster Computing*, pages 252–261. IEEE, September 2008.

[GE11] Fanglu Guo and Petros Efstathopoulos. Building a highperformance deduplication system. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 25–25. USENIX, 2011.

[GMB10] Yan Gao, Dirk Meister, and André Brinkmann. Reliability analysis of declustered-parity RAID 6 with disk scrubbing and considering irrecoverable read errors. In *Proceedings of the 5th IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 126–134. IEEE, 2010.

[GRI94] Grib format edition 1. Technical report, World Meteorological Organization, 1994.

[HDF11] HDF5 user's guide. Technical report, The HDF Group, November 2011.

[Hen03] Val Henson. An analysis of compare-by-hash. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems*. USENIX, 2003.

[HH05] Val Henson and Richard Henderson. Guidelines for using compare-by-hash. http://valerieaurora.org/review/hash2.pdf, 2005.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*. USENIX, 2010.

[HL04] Bo Hong and Darrell D. E. Long. Duplicate data elimination in a san file system. In *Proceedings of the 21th IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*. IEEE, 2004.

[HMN+12] Danny Harnik, Oded Margalit, Dalit Naor, Dmitry Sotnikov, and Gil Vernik. Estimation of deduplication ratios in large data sets. In *Proceedings of the 28th IEEE Conference on Mass Storage Systems and Technologies (MSST)*. IEEE, 2012.

[HRSD07] Allison L. Holloway, Vijayshankar Raman, Garret Swart, and David J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 2007.

[Huf52] David Albert Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute for Radio Engineers (IRE)*, 40(9):1098–1101, 1952.

[IAFB12] Dewan Ibtesham, Dorian Arnold, Kurt Ferreira, and Patrick Bridges. On the viability of checkpoint compression for extreme scale fault tolerance. In *Proceedings of the Euro-Par 2011 Parallel Processing Workshops*, pages 302–311. Springer, 2012.

[iee86] IEEE standard glossary of mathematics of computing terminology. Standard, 1986.

[JM09] Keren Jin and Ethan L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of the 2nd Israeli Experimental Systems Conference (SYSTOR)*. ACM, 2009.

[Jon10] Stephanie N Jones. Online de-duplication in log-structured file system for primary storage. Master's thesis, University of California, Santa Cruz, 2010.

[JPZ+11] K. R. Jayaram, Chunyi Peng, Zhe Zhang, Minkyong Kim, Han Chen, and Hui Lei. An empirical analysis of similarity in virtual machine images. In *Proceedings of the Middleware 2011 Industry Track Workshop*, pages 6:1–6:6. ACM, 2011.

[JSPW+11] Stephanie N. Jones, Christina R. Strong, Aleatha Parker-Wood, Alexandra Holloway, and Darrell D.E. Long. Easing the burdens of HPC file management. In *Proceedings of the 6th Workshop on Parallel Data Storage (PDSW)*, pages 25–30. ACM, 2011.

[KA01] Kimberly Keeton and Eric Anderson. A backup appliance composed of high-capacity disk drives. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*, page 171. USENIX, 2001.

[Kai11] Jürgen Kaiser. Distributed fault-tolerant inline deduplication. Master's thesis, University of Paderborn, March 2011.

[KBKD12] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR)*, page 11. ACM, 2012.

[KDLT04] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 59–72. USENIX, 2004.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC)*. ACM, 1997.

[KMB+11] Matthias Keller, Dirk Meister, André Brinkmann, Christian Terboven, and Christian Bischof. eScience cloud infrastructure. In *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 188–195. IEEE, 2011.

[KMBE12] Jürgen Kaiser, Dirk Meister, André Brinkmann, and Sascha Effert. Design of an exact data deduplication cluster. In *Proceedings of the 28th IEEE Conference on Mass Storage Systems and Technologies (MSST)*. IEEE, May 2012.

[KR10] Ricardo Koller and Raju Rangaswami. I/O deduplication: Utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):13, 2010.

[KRVG08] Samer Al Kiswany, Matei Ripeanu, Sudharshan S. Vazhkudai, and Abdullah Gharaibeh. stdchk: A checkpoint storage system for desktop grid computing. In *Proceedings of the 28th International Conference on Distributed Computing Systems (ICDCS)*, 2008.

[KTSZ09] Rachita Kothiyal, Vasily Tarasov, Priya Sehgal, and Erez Zadok. Energy and performance evaluation of lossless file data compression on server systems. In *Proceedings of the 2nd Israeli Experimental Systems Conference (SYSTOR)*, pages 4:1–4:12. ACM, 2009.

[KUD10] Erik Kruus, Cristian Ungureanu, and Cezary Dubnicki. Bimodal content defined chunking for backup streams. In *Proceedings of the 8th USENIX Conference on File and storage technologies (FAST)*, page 18. USENIX, 2010.

[LCGC12] Maohua Lu, David Chambliss, Joseph Glider, and Cornel Constantinescu. Insights for data reduction in primary storage: a practical analysis. In *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR)*, page 17. ACM, 2012.

[LEB+09] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 2009.

[LEB13] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, February 2013.

[LGLZ12] Zhichao Li, Kevin M. Greenan, Andrew W. Leung, and Erez Zadok. Power consumption in enterprise-scale backup storage systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, pages 6–6. USENIX, 2012.

[LH08] Anthony Liguori and Eric Van Hensbergen. Experiences with content addressable storage and virtual disks. In *Proceedings of the Workshop on I/O Virtualization (WIOV'08)*, 2008.

[LJD10] Guanlin Lu, Yu Jin, and David H.C. Du. Frequency based chunking for data de-duplication. In *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS),*, pages 287–296. IEEE, 2010.

[LLS+08] Chuanyci Liu, Yingping Lu, Chunhui Shi, Guanlin Lu, David H.C. Du, and Dong-Sheng Wang. ADMAD: Application-driven metadata aware de-duplication archival storage system. In *Proceedings of the Fifth IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, pages 29–35. IEEE, 2008.

[LND12] Guanlin Lu, Youngjin Jin Nam, and David H.C. Du. BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In *Proceedings of the 28th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2012.

[LPB11] Y. V. Lokeshwari, B. Prabavathy, and Chitra Babu. Optimized cloud storage with high throughput deduplication approach. In *Proceedings of the International Conference on Emerging Technology Trends (ICETT)*, 2011.

[LPG+11] Jay Lofstead, Milo Polte, Garth Gibson, Scott Klasky, Karsten Schwan, Ron Oldfield, Matthew Wolf, and Qing Liu. Six degrees of scientific data: reading patterns for extreme scale science io. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 49–60. ACM, 2011.

[LT96] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. *ACM SIGOPS Operating Systems Review*, 30(5):84–92, December 1996.

[Man94] Udi Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*. USENIX, 1994.

[Mat04] Jeanna Neefe Matthews. The case for repeated research in operating systems. *ACM SIGOPS Operating Systems Review*, 38(2):5–7, April 2004.

[MB09] Dirk Meister and André Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *Proceedings of the 2nd Israeli Experimental Systems Conference (SYSTOR)*, pages 8:1–8:12. ACM, 2009.

[MB10a] Dirk Meister and André Brinkmann. dedupv1: Improving deduplication throughput using solid state drives. In *Proceedings of the 26th IEEE Conference on Mass Storage Systems and Technologies (MSST)*. IEEE, May 2010.

[MB10b] Dirk Meister and André Brinkmann. dedupv1: Improving deduplication throughput using solid state drives (technical report). Technical report, University of Paderborn, May 2010.

[MB12] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. *Trans. Storage*, 7(4):14:1–14:20, February 2012.

[MBS13] Dirk Meister, André Brinkmann, and Tim Süß. File recipe compression in data deduplication systems. In *Proceedings of 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 175–182. USENIX, 2013.

[MBV08] Henry M Monti, Ali R Butt, and Sudharshan S Vazhkudai. Timely offloading of result-data in HPC centers. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ISC)*, pages 124–133. ACM, 2008.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[MCM01] Athicha Muthitacharoen, Benjie Chen, and D. Mazières. A low-bandwidth network file system. *SIGOPS Oper. Syst. Rev.*, 35(5):174–187, October 2001.

[Mei08] Dirk Meister. Daten-Deduplizierung mit Hilfe von Flash-basierten Indizes. Master's thesis, University of Paderborn, December 2008.

[MG82] Jayadev Misra and David Gries. Finding repeated elements. Technical report, Cornell University, Ithaca, NY, USA, 1982.

[MKB+12] Dirk Meister, Jürgen Kaiser, André Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. A study on data deduplication in HPC storage systems. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 7:1–7:11. IEEE, November 2012.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[Mor78] Robert Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, October 1978.

[MOS+05] N. H. Margolus, E. Olson, M. Sclafani, C. J. Coburn, and M Fortson. A storage system for randomly named blocks of data. Patent WO 2006/042019A2, 2005.

[MPR+03] Jai Menon, David A. Pease, Robert M. Rees, Linda Duyanovich, and B. L. Hillsberg. IBM Storage Tank – a heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2):250–267, 2003.

[MRHBS06] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference (ATC)*, pages 43–56, 2006.

[MRS09] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2009.

[MYW11] Jaehong Min, Daeyoung Yoon, and Youjip Won. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers*, 60(6):824–840, 2011.

[MZSU08] Nagapramod Mandagere, Pin Zhou, Mark A Smith, and Sandeep Uttamchandani. Demystifying data deduplication. In *Proceedings of the Middleware Conference Companion*, 2008.

[NCL+09] Sumit Narayan, John A Chandy, Samuel Lang, Philip Carns, and Robert Ross. Uncovering errors: the cost of detecting silent data corruption. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, pages 37–41. ACM, 2009.

[Nel95] Mark Nelson. *The Data Compression Book*. John Wiley and Sons, 2. edition edition, December 1995.

[NKO+06] Partho Nath, Michael A. Kozuch, David R. O'Hallaron, Jan Harkes, M. Satyanarayanan, Niraj Tolia, and Matt Toups. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *Proceedings of the 2006 USENIX Annual Technical Conference (ATC)*, pages 6–6. USENIX, 2006.

[NKP+96] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael L. Best. File-access characteristics of parallel scientific workloads. *IEEE Trans. Parallel Distrib. Syst.*, October 1996.

[NSM04] David Nagle, Denis Serenyi, and Abbie Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC)*. IEEE, 2004.

[OCGO96] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008.

[Pat06] Hugo Patterson. Data Storage using Identifiers. US Patent 7,143,251, November 2006.

[PB61] William Wesley Peterson and DT Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, 1961.

[PBB+99] Kenneth W. Preslan, Andrew P. Barry, Jonathan E. Brassow, Grant M. Erickson, Erling Nygaard, Christopher J. Sabol, Steven R. Soltis, David C. Teigland, and Matthew T. O'Keefe. A 64-bit, shared disk file system for Linux. In *Proceedings of the 16th IEEE Symposium on Symposium on Mass Storage Systems (MSS)*, pages 22–41. IEEE, 1999.

[PBM09] Rajesh Krishna Panta, Saurabh Bagchi, and Samuel P Midkiff. Zephyr: Efficient incremental reprogramming of sensor nodes using function call indirections and difference computation. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2009.

[Per12] Permabit. Permabit albireo data optimization software. White paper, `http://permabit.com/resources/permabit-albireo-data-optimization-software/`, January 2012.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Rec.*, 17(3):109–116, June 1988.

[PGLP07] Swapnil V. Patil, Garth A. Gibson, Sam Lang, and Milo Polte. GIGA+: scalable directories for shared file systems. In *Proceedings of the 2nd International Workshop on Petascale Data Storage (PDSW)*, pages 26–29. ACM, 2007.

[PL10] Nohhyun Park and David J. Lilja. Characterizing datasets for data deduplication in backup applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, December 2010.

[PP04] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the 2004 USENIX Annual Technical Conference (ATC)*, pages 73–86. USENIX, 2004.

[PRPS12] J. Paulo, P. Reis, J. Pereira, and A. Sousa. DEDISbench: A benchmark for deduplicated storage systems. In *On the Move to Meaningful Internet Systems: OTM 2012*, Lecture Notes in Computer Science, pages 584–601. Springer, 2012.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[QD02] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, pages 7–7. USENIX, January 2002.

[Rab81] Michael O. Rabin. Fingerprinting by random polynomials. Technical report, Center for Research in Computing Technology, Harvard University, 1981.

[RBK92] K. K. Ramakrishnan, Prabuddha Biswas, and Ramakrishna Karedla. Analysis of file I/O traces in commercial computing environments. In *Proceedings of the 1992 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, 1992.

[RCP08] Sean Rhea, Russ Cox, and Alex Pesterev. Fast, inexpensive content-addressed storage in Foundation. In *Proceedings of the USENIX 2008 Annual Technical Conference (ATC)*. USENIX, 2008.

[RDE+11] Russ Rew, Glenn Davis, Steve Emmerson, Hervey Davies, Ed Harnett, and Dennis Heimigner. The netcdf users guide. User guide, University Corporation for Atmospheric Research, Unidata Program Center, June 2011.

[RLC+08] Robert Ross, Walt Ligon, Phil Carns, Robert Latham, Neill Miller, Frank Shorter, Harish Ramachandran, Dale Whitchurch, Mike Speth, and Brad Settlemyer. Orange file system. `http://www.orangefs.org/`, 2008.

[RMP11] Pierre Riteau, Christine Morin, and Thierry Priol. Shrinker: improving live migration of virtual clusters over WANs with distributed data deduplication and content-based addressing. *Proceedings of the 17th International Conference on Parallel processing (Euro-Par 2011)*, pages 431–442, 2011.

[Rot07] Philip C. Roth. Characterizing the I/O behavior of scientific applications on the Cray XT. In *Proceedings of the 2nd International Workshop on Petascale Data Storage (PDSW)*, pages 50–55. ACM, November 2007.

[RS98] Martin Raab and Angelika Steger. "balls into bins" - a simple and tight analysis. In *Proceedings of the 2nd International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*, 1998.

[SAHI+13] Przemyslaw Strzelczak, Elzbieta Adamczyk, Urszula Herman-Izycka, Jakub Sakowicz, Lukasz Slusarczyk, Jaroslaw Wrona, and Cezary Dubnicki. Concurrent deletion in a distributed content-addressable storage system with global deduplication. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 161–174. USENIX, 2013.

[Say00] Khalid Sayood. *Introduction to Data Compression*. Morgen-Kaufmann, 2000.

[SBGV12] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 2012.

[SDG10] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding latent sector errors and how to protect against them. *Trans. Storage*, 6(3):9:1–9:23, September 2010.

[Sep10] Septon. DeltaStor Software Datasheet. `http://go.sepaton.com/rs/sepaton/images/2011%20DeltaStor%20Datasheet.pdf`, 2010.

[SG07] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1), 2007.

[SH02] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, pages 231–244, 2002.

[SHWH12] Philip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu. WAN optimized replication of backup datasets using stream-informed delta compression. *ACM Transactions on Storage (TOS)*, 8(4):5–5, November 2012.

[SHWK76] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. The design and implementation of INGRES. *ACM Trans. Database Syst.*, 1(3):189–222, September 1976.

[Sie06] Jerry Sievert. Iometer: The i/o performance analysis tool for servers. `http://www.iometer.org/`, July 2006.

[SJPW+11] Christina Strong, Stephanie Jones, Aleatha Parker-Wood, Alexandra Holloway, and Darrell D. E. Long. Los Alamos National Laboratory interviews. Technical Report UCSC-SSRC-11-06, University of California, Santa Cruz, September 2011.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *Proceedings of the 26ths IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2010.

[SMS+04] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. RFC 3720, internet small computer systems interface (iSCSI). Technical report, Internet Engineering Task Force - Network Working Group, April 2004.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems (TOCS)*, 1(3):222–238, August 1983.

[Sto86] Michael Stonebraker. The case for shared nothing. *Database Engineering*, 9(1):4–9, 1986.

[SXM+00] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, T. Rytina, M. Kalla, L. Zhang, and V. Paxson. RFC 4960: Stream control transmission protocol. Technical report, Internet Engineering Task Force - Network Working Group, October 2000.

[SXM+04] Thomas JE Schwarz, Qin Xin, Ethan L Miller, Darrell DE Long, Andy Hospodor, and Spencer Ng. Disk scrubbing in large archival storage systems. In *Proceedings of the 12th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, pages 409–418. IEEE, 2004.

[TBZS11] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking file system benchmarking: It *is* rocket science. In *Proceedings of the 13th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2011.

[TJF+11] Yujuan Tan, Hong Jiang, Dan Feng, Lei Tian, and Zhichao Yan. CABdedupe: A causality-based deduplication performance booster for cloud backup services. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2011.

[TKS+03] Niraj Tolia, Michael Kozuch, Mahadev Satyanarayanan, Brad Karp, Thomas Bressoud, and Adrian Perrig. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the 2003 USENIX Annual Technical Conference (ATC)*, 2003.

[TMB+12] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. Generating realistic datasets for deduplication analysis. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*. USENIX, 2012.

[TML97] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 224–237. ACM, December 1997.

[TS07] Niraj Tolia and M. Satyanarayanan. Consistency-preserving caching of dynamic database content. In *Proceedings of the 16th international conference on World Wide Web*, pages 311–320. ACM, 2007.

[TW11] Yoshihiro Tsuchiya and Takashi Watanabe. DBLK: Deduplication for primary block storage. In *Proceedings of the 27th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5. IEEE, May 2011.

[TYFS12] Yujuan Tan, Zhichao Yan, Dan Feng, and Edwin Sha. Reducing the delinearization of data placement to improve deduplication performance. In *Proceedings of the International Workshop On Data-Intensive Scalable Computing Systems (DISCS)*, 2012.

[TZJW08] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS)*, 4(2):5, May 2008.

[UAA+10] Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Całkowski, Cezary Dubnicki, and Aniruddha Bohra. HydraFS: a high-throughput file system for the hydrastor content-addressable storage system. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*, pages 17–17. USENIX, 2010.

[Vit85] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, March 1985.

[VMw09] VMware, Inc. VMware VMFS product datasheet. `http://www.vmware.com/pdf/vmfs_datasheet.pdf`, 2009.

[Vog99] Werner Vogels. File system usage in Windows NT 4.0. *ACM SIGOPS Operating Systems Review*, 33(5):93–109, December 1999.

[WBD+89] B. Welch, M. Baker, F. Douglis, J. Hartman, M. Rosenblum, and J. Ousterhout. Sprite position statement: Use distributed state for failure recovery. In *Proceedings of the 2nd Workshop on Workstation Operating Systems (WWOS)*, pages 130–133, September 1989.

[WBM+06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–320. USENIX, 2006.

[WDQ+12] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 2012.

[Wil08] Andrew Wilson. The new and improved filebench. In *Proceedings of 6th USENIX Conference on File and Storage Technologies (FAST), Poster*, 2008.

[WJZF10] Jiansheng Wei, Hong Jiang, Ke Zhou, and Dan Feng. MAD2: A scalable high-throughput exact deduplication approach for network backup services. In *Proceedings of the 26th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, May 2010.

[WKB+08] Youjip Won, Rakie Kim, Jongmyeong Ban, Jungpil Hur, Sangkyu Oh, and Jangsun Lee. PRUN: Eliminating information redundancy for large scale data backup system. In *Proceedings of the International Conference on Computational Sciences and Its Applications (ICCSA)*, pages 139–144. IEEE, July 2008.

[WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes*. Morgan Kaufmann, 1999.

[WMR12] Avani Wildani, Ethan L. Miller, and Ohad Rodeh. HANDS: A heuristically arranged non-backup in-line deduplication system. Technical Report UCSC-SSRC-12-03, University of California, Santa Cruz, March 2012.

[WUA+08] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, volume 8, pages 2:1–2:17. USENIX, 2008.

[XJFH11] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. SiLo: a similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, pages 26–28. USENIX Association, 2011.

[YCT+12] Alan G. Yoder, Mark Carlson, David Thiel, Don Deel, and Eric Hibbard. *2012 SNIA Dictionary*. SNIA, 2012.

[YJF+10] Tianming Yang, Hong Jiang, Dan Feng, Zhongying Niu, Ke Zhou, and Yaping Wan. DEBAR: A scalable high-performance de-duplication storage system for backup and archiving. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–12, April 2010.

[YPL05] Lawrence L. You, Kristal T. Pollack, and Darrell D.E. Long. Deep store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 804–815. IEEE, 2005.

[ZHMM10] Xiang Zhang, Zhigang Huo, Jie Ma, and Dan Meng. Exploiting data deduplication to accelerate live virtual machine migration. In *Proceedings of the*

2010 IEEE International Conference on Cluster Computing (CLUSTER), pages 88–96. IEEE, September 2010.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[ZLP08] Benjamin. Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 2008.