Z-Dedup:A Case for Deduplicating Compressed Contents in Cloud

Zhichao Yan, Hong Jiang
University of Texas Arlington
zhichao.yan@mavs.uta.edu
hong.jiang@uta.edu

Yujuan Tan
Chongqing University
tanyujuan@gmail.com
Corresponding Author

Stan Skelton
NetApp
Stan.Skelton@netapp.com

Hao Luo
Twitter
hluo@twitter.com

Abstract

Lossless data reduction techniques, particularly compression and deduplication, have emerged as effective approaches to tackling the combined challenge of explosive growth in data volumes but lagging growth in network bandwidth, to improve space and bandwidth efficiency in the cloud storage environment. However, our observations reveal that traditional deduplication solutions are rendered essentially useless in detecting and removing redundant data from the compressed packages in the cloud, which are poised to greatly increase in their presence and popularity. This is because even uncompressed, compressed and differently compressed packages of the exact same contents tend to have completely different byte stream patterns, whose redundancy cannot be identified by comparing their fingerprints. This, combined with different compressed packets mixed with different data but containing significant duplicate data, will further exacerbate the problem in the cloud storage environment. To address this fundamental problem, we propose Z-Dedup, a novel deduplication system that is able to detect and remove redundant data in compressed packages, by exploiting some key invariant information embedded in the metadata of compressed packages such as file-based checksum and original file length information. Our evaluations show that Z-Dedup can significantly improve both space and bandwidth efficiency over traditional approaches by eliminating 1.61% to 98.75% redundant data of a compressed package based on our collected datasets, and even more storage space and bandwidth are expected to be saved after the storage servers have accumulated more compressed contents.

I. Introduction

Information explosion has significantly increased the amount of digital contents in the big data era [1] [2]. For example, the world's technological capacity to store information grew from 2.6 (optimally compressed) exabytes (EB) in 1986 to 15.8 EB in 1993, over 54.5 EB in 2000, and to 295 (optimally compressed) EB in 2007. In fact, we create 2.5 quintillion bytes (EB) of data every day, and 90% of the current world's digital data are created in the last two years alone [3]. These ever-increasing digital contents require a large amount of storage capacity to keep them available for future accesses. To meet performance, reliability, availability, power, and cost efficiency requirements, cloud storage service is becoming a core infrastructure to host these data. However, with the Cloud, network bandwidth becomes a bottleneck because of the need to upload and download massive amounts of data between clients and the servers, and the upload bandwidth is usually an order of magnitude smaller than the download bandwidth. Therefore, how to manage such incredible growth in storage demand has become the most important challenge to the cloud storage service providers [2].

Lossless data reduction techniques, most notably, compression and deduplication, have emerged to be an effective way to address this challenge. Compression finds repeated strings within a specific range of a given file and replaces them with a more compact coding scheme with intensive computations. For example, the deflation algorithm used in gzip checks the repeated strings within a limited $32KB$ window and the longest repeated string's length is limited to 258 bytes [4]. On the other hand, deduplication divides a file into fixed-size (e.g., 4KB in fixed chunking) or variable-size chunks (e.g., content defined chunking), identifies (i.e., lookup the hash table) and removes the duplicate chunks across all existing files by comparing chunks' unique fingerprints (e.g., secure hash values), and reassembles the chunks to serve the subsequent access operations. Therefore, compression can achieve the best data reduction ratio but at both high computation and memory costs, rendering it most suitable for reducing individual files and data items at small volumes locally. On the other hand, deduplication can remove redundant data at a much coarser granularity to obtain a good data reduction ratio with a much lower computation cost, thus making it most effective in eliminating much larger volumes of redundant data across files and data items globally. In order to design an efficient cloud storage system, designers need to combine these two techniques to detect and remove redundant data by exploiting both global and local redundancies with acceptable computation costs.

However, existing deduplication approaches cannot detect and remove the redundancy wrapped by different compression approaches because they only rely on bitstream fingerprint-based redundancy identification. As a result, they are unable to identify redundant data between the compressed and uncompressed versions of the exact same contents as the compressed contents, being encoded by a compression algorithm, will have very different bitstream patterns from their uncompressed counterparts. In fact, we find that they also cannot detect any redundant data compressed by different compression algorithms because they will generate different compressed data at the bitstream level of the same contents. Furthermore, different versions or input parameters of the same compression tool may generate different bitstreams of the same contents (e.g.,

different metadata information or compressed bitstream will be embedded in the compressed files), whose redundancy cannot be identified by fingerprint-based detection. Finally, very similar but slightly different digital contents (e.g., different versions of an open source software), which would otherwise present excellent deduplication opportunities, will become fundamentally distinct compressed packages after applying even the exact same compression algorithm.

In a multi-user cloud storage environment, the aforementioned compression scenarios are arguably widespread, which prevent the cloud servers from detecting and removing any data redundancy among these compressed packages. There are more complicated scenarios that chain multiple compression algorithms and pre-processing filters together to gain higher compression ratios [5]. All these scenarios indicate that several factors limit the applicability of deduplication in the cloud storage environment, leading to failures in identifying and removing potentially significant data redundancy across different compressed packages. These factors include compression algorithms, compression parameters, input file stream, etc. In general, two users can generate the same compressed packages only when they compress the same files with the same compression parameters by the same version of a compression tool. Obviously, it is impractical to assign the exact same specific compression mode to all the users in the cloud storage environment. This is also the reason why traditional data deduplication systems try to skip deduplicating compressed archives because they may share little redundant data in their binary representations with the original uncompressed or other differently compressed files, even though they may have the same semantic contents [6]. If there are 10 compression tools that are widely used for a cloud storage environment, where each has 10 versions and 5 sets of compression parameters, then a given file can in theory have up to 500 different compressed instances in the cloud, yet with no redundancy detectable by conventional deduplication. Moreover, if this file is compressed in conjunction with other different files (e.g., by solid compression detailed in Section II-B1), we can expect even more compressed instances of this particular file stored in the cloud. This kind of data redundancy is concealed by different compressed packages. As cloud storage is poised to become a digital content aggregating point in the digital universe, it is arguable that this type of hidden data redundancy already exists widely in deduplication-based storage systems and will likely increase with time. Thus, it is necessary to detect and remove this kind of data redundancy for a more efficient cloud storage ecosystem.

In our recent work [7], we proposed X-Ray Dedup, reporting the preliminary results on deduplicating the redundant contents concealed only in the non-solid compressed packages, and without addressing any of the solid compression, hash collision and security issues. In this paper, we propose Z-Dedup that expands and improves X-Ray Dedup substantially to support deduplication for both non-solid and solid compressed packages. In addition to evaluating the data deduplication benefits under these two compression schemes, Z-Dedup fundamentally addresses the hash collision problem of the CRC code used in X-Ray Dedup by replacing it with SHA-256. This enables Z-Dedup to be seamlessly integrated into the cloud storage environment at an acceptable hash collision rate. Moreover, we added an ownership verification scheme to defend against possible known-hash attacks at the client side. Our experimental results show that Z-Dedup can significantly improve both space and bandwidth efficiency over traditional approaches by eliminating 1.61% to 98.75% more redundant data among various compressed packages, and more storage space and network bandwidth are expected to be saved after the cloud storage servers hosting enough compressed contents.

TABLE I: Broadband speed greater than 10 Mbps, 25 Mbps and 100 Mbps (2015-2020) [8].

Region	>10 Mbps		>25 Mbps		>100 Mbps	
	2015	2020	2015	2020	2015	2020
Global	53%	77%	30%	38%	4%	8%
Asia Pacific	53%	83%	30%	52%	4%	8%
Latin America	27%	39%	10%	15%	1%	2%
North America	64%	88%	38%	52%	5%	9%
Western Europe	54%	74%	32%	43%	5%	11%
East-Central Europe	58%	83%	33%	41%	3%	6%
Middle East & Africa	17%	20%	7%	8%	0.3%	1%

The rest of the paper is organized as follows. Section II describes the necessary background to motivate this work. Section III elaborates on the design and implementation of the Z-Dedup system. We present the evaluation and analysis of Z-Dedup in Section IV and conclude the paper in Section V.

II. Motivation

A. Background

There are two main motivations for both users and cloud storage providers to compress their data, leading to more compressed packages in the cloud. One is to reduce the amount of data actually stored in the system and the other is to reduce the time spent transferring a large amount of data over the networks. While the former helps reduce cost, both hardware and operational, the latter improves performance, both upload and download, and network bandwidth utilization.

1) Cloud Storage in Face of Slow Networks: Table I lists the percentages of broadband connections faster than 10 Mbps, 25 Mbps, and 100 Mbps by region in 2015 and 2020, indicating the broadband speeds are still quite low considering that each person accounts for a considerable amount of digital contents. Moreover, upload and download bandwidths are usually asymmetric, with the former being one order of magnitude smaller than the latter. As a result, network bandwidth remains one of the main limiting factors in using cloud storage.

2) Compressions Are Increasingly Common: In order to deal with the incredible digital explosion, the application of data compression technologies, especially lossless data compression, is expected to become a commonplace throughout the life cycle of digital contents. In fact, a lot of systems and applications already perform compression and decompression without the users even being aware that it has occurred. However, from a storage perspective, this also means that there will be increasingly more compressed packages as more compressions of different formats and algorithms are performed.

TABLE II: Data profile of sunsite.org.uk [9].

Rank	Popularity		Storage Space	
	Ext.	% Occur.	Ext.	% Storage
1	.gz	32.50	.rpm	29.30
2	.rpm	10.60	.gz	20.95
3	.jpg	7.54	.iso	20.40
4	.html	4.83	.bz2	6.26
5	.gif	4.43	.tbz2	5.65
6	-	4.16	.raw	4.44
7	.lsm	3.74	.tgz	2.66
8	.tgz	2.90	.zip	2.53
9	.tgz2	2.35	.bin	2.00
10	.Z	2.12	.jpg	0.94
11	.asc	1.84	.Z	0.65
12	.zip	1.59	.gif	0.43
13	.rdf	1.39	.tif	0.31
14	.htm	1.21	.img	0.21
15	.o	1.06	.au	0.19
Total	-	82.26	-	96.92

Table II shows the compiled results by Calicrates Policroniades and Ian Pratt [9] of popular files' extensions and their relative popularity in an Internet server, indicating that a significant amount of different compressed contents occupy most of the storage space. This is consistent with our observation and the reason lies in the fact that the network bandwidth, particularly of wide-area networks, has become much more expensive than computing resources, leading more and more data being compressed before transmitting to the cloud.

With the increasing numbers of compressed packages of different formats in the cloud, there will be even greater numbers of redundant files concealed in different compressed packages. How to organize and store these compressed packages becomes a challenge as we face a mounting demand to further reduce storage capacity requirement in light of the explosive growth of digital contents.

B. Data Compression and Deduplication

1) Dictionary-Based Compression: We focus on dictionary-based compression, especially the sliding dictionary compression algorithms (LZ77 and LZ78), which were proposed by Abraham Lempel and Jacob Ziv [10] [11]. These two algorithms form the basis for a plethora of their variations, including LZW, LZSS, LZMA and others, that are collectively referred to as the LZ-family algorithms.

By their initial designs, compression tools only compress individual files because the compression algorithm only focuses on a single input data stream. Then some compression tools allow for the compression of a directory structure that contains multiple files and/or subdirectories into a single compressed package. There are two common ways to compress the directory structure, namely, compressing each file independently and concatenating a group of uncompressed files into a single file for compression. The former is called *non-solid compression* and widely supported by the zip format [12], while the latter is called *solid compression* and natively used in the 7z and rar formats [5] [13]. Solid compression is also indirectly used in tar-based formats such as .tar.gz and .tar.xz, where users archive a directory containing multiple files first as a single file that is then compressed. In this paper, we only focus on these two common compression schemes while considering out of scope schemes that chain multiple compression algorithms and filters to gain higher compression ratios.

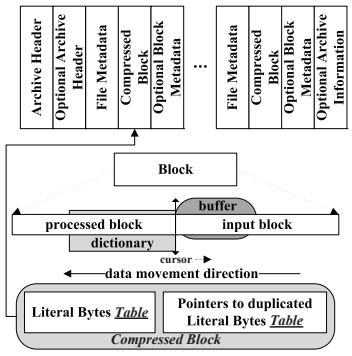


Fig. 1: Sliding dictionary compression.

As shown in Figure 1, a compressed archive is organized in a data package consisting of an archive header, an optional archive header, several compressed blocks with the corresponding file metadata and optional block metadata, and optional archive information. Archive header maintains the basic information on this compressed package. Block is the basic, atomic unit of compression, which can be of variable size and contain either some parts of a file or some files. Optional archive information contains the necessary information about directory's structure. A cursor will scan the input block and some parts of an already processed block to constitute a dictionary window that is used for the subsequent read buffer to find the longest repeated literal bytes. This process will generate an adaptive dictionary, as illustrated in Algorithm 1. After the cursor has scanned a whole block, it will generate another table containing literal bytes and a table containing pointers to duplicated literal bytes. Finally, these two tables will be compressed into a compressed block.

Algorithm 1 Adaptive dictionary compression.

```
while Not the end of block do
    Word = readWord( InputFile )
    DictionaryIndex = lookup( Word, Dictionary )
    if (DictionaryIndex IS VALID) then
        output( DictionaryIndex, OutputFile )
    else
        output( Word, OutputFile )
        addToDictionary ( Word, Dictionary )
    end if
end while
```

2) Deduplication: Deduplication is a special kind of lossless data compression, which aims to detect and remove the extremely long repeated strings/chunks across all files. It becomes popular in storage systems, particularly in backup and archiving systems, because of the datasets in these systems have significant amount of duplicative data redundancy.

Deduplication's basic idea is to divide the data stream into independent units, such as files or data chunks, and then use some secure hash values of these units, often referred to as fingerprints, to uniquely represent them. If the fingerprints of any two units match, these two data units are considered duplicate of each other. With this compare-by-hash approach,

duplicate contents within a single storage system or across multiple systems can be quickly detected and removed.

Deduplication relies on secure hash algorithms to limit the hash collision rate to avoid false positives, where units of different data contents produce the same hash values and thus are mistakenly considered duplicates. It adds extra operations, such as hash computation and hash index table lookup, to the I/O critical path. Another key issue of deduplication is its heavy reliance on the format of the digital content itself for the hash value calculation. For example, two identical files in different formats (e.g., due to compression or encryption) will generate essentially different hash values, completely preventing their data redundancy being detected and removed.

C. Research Problems

In above analysis, more and more compressed archives are expected to be stored in the cloud. In fact, some systems, such as ZFS and Flash Array, have already integrated compression and deduplication [14] [15]. Meanwhile, the cloud will be a convergence point to which increasing numbers of end users and systems upload their compressed packages. Within these compressed archives, there can be a great number of redundant files across different users, particularly if they happen to have similar interests in certain specific topics. However, due to the different ways in which compressed archives are generated, which can substantially hide the data redundancy detectable by the conventional deduplication technology, these problems arise when dedupication interplays with compression to motivate this work. First, *how to construct a fingerprint for each compressed file across the different compressed packages to help detect redundant files?* Second, *how to remove such redundant files compressed in different packages?* Third, *how to integrate it into existing data deduplication system to detect and remove redundant data between compressed packages and uncompressed files?* Fourth, *how to deduplicate the redundant files among solid compression packages?*

D. Related Works

Deduplication was proposed to remove redundant data in backup application [16]. As more and more data migrates to the cloud, it has been integrated within the cloud platform [17]. Different from the backup storage systems where chunk-level deduplication dominates, there exists strong evidence indicating that file-level deduplication can achieve comparable compression ratio to chunk-level deduplication in the cloud environment [18] [19]. However, it remains a challenge to find redundant data within a compressed package or among different compressed packages because conventional methods for detecting data redundancy usually scan the compressed bitstream itself without touching its internal information. Existing approaches, such as Migratory compression [20] and Mtar [21], must reorganize the internal structure of the compressed block, which makes them more suitable for a non-cloud-like, controllable environment of a private company/organization. More importantly, to ensure deduplicability, they require the same compression algorithm and parameter-set to be used to generate the same compressed blocks from the duplicate input files, something Z-Dedup is designed to avoid. X-Ray Dedup [7] has correctly identified the problem of the hidden redundancy among compressed packages and proposed a preliminary solution that combines file's size and CRC-32 metadata as its signature to detect and remove potential redundant files only for the non-solid compression scheme. However, it fails to provide a complete and generalizable system design. More importantly, it lacks an in-depth and comprehensive study of the collision problem and completely ignores the security issue. For example, we find that the 32-bit CRC checksum is not sufficient to avoid hash collisions in a large-scale system. By the birthday paradox theory, the collision rate of X-Ray Dedup will be 0.1% when there are 1.9×10^8 different files stored in the system. It also may suffer from security vulnerabilities such as the known-hash attacks. Z-Dedup expands and improves X-Ray Dedup substantially to support deduplication for both non-solid and solid compressed packages. In addition to evaluating the data deduplication benefits under these two compression schemes, Z-Dedup fundamentally addresses the hash collision problem by adding SHA-256 as the new checksum for compression. This enables Z-Dedup to be seamlessly integrated into the cloud storage environment at an acceptable hash collision rate. Z-Dedup also adds an ownership verification scheme to defend against possible known-hash attacks at the client side, which can be extended to handle possible side-channel attacks like the "confirmation-of-a-file" attack by injecting random latencies.

III. Design and Implementation

A. Main Idea

A compression algorithm will encode an input stream into an essentially different output stream, which makes it difficult to generate a simple but unique indicator for mapping between the input stream and the output stream. However, a critical observation we gained through our analysis is that *most compression algorithms will pack files' metadata information within the compressed packages and, importantly, some metadata, such as original file's size and checksum, will be invariant across all compressed packages*. Therefore, the main idea here is to combine this file-level invariant information as a new and unique indicator for the mapping to help detect redundant files across all the compressed packages globally.

This approach can be integrated into existing deduplication systems to help remove redundant files if these files are compressed in the non-solid compression method (SectionII-B1). For solid compression, we can leverage this feature to estimate the content similarity with any existing compressed packages. If the incoming compressed package is considered sufficiently similar to existing compressed packages, meaning that many (but not all) of the individual files within the packages are possibly identical to the files in existing stored compressed packages, then the system can decide to deduplicate by first uncompressing and then detecting and removing redundancy in

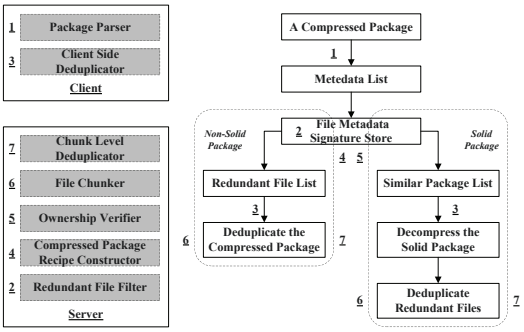


Fig. 2: Z-Dedup's main modules and workflow.

the conventional way. This approach helps data deduplication system quickly find out the solid compressed packages that contain significant data redundancy. Otherwise, the system will either ignore deduplicating solid compressed packages or uncompressing every solid compressed package to perform deduplication work.

B. System Design and Implementation

1) An System Overview of Z-Dedup: In Z-Dedup, once a compressed package is ready to be uploaded to the cloud, the client and the cloud server will exchange some metadata information and perform deduplication at the client side, which will generate a deduplicated compressed package. This scheme not only decreases the cloud capacity requirement, but also reduces the upload time from the client to the cloud server.

Figure 2 shows 7 main functional modules of Z-Dedup, each enclosed by a dashed gray box. The numerical numbers next to them both label them and indicate the order in which they process the incoming compressed package. *Package Parser* is responsible for parsing each compressed package to extract its file-level signature metadata list. *Redundant File Filter* detects duplicate files by looking up the file metadata signature store, and for solid package, it also helps query its similar package list. *Client Side Deduplicator* is responsible for either removing compressed files in a non-solid package or deciding whether or not to decompress a solid package and duplicate redundant files at the client side. *Compressed Package Recipe Constructor* books the necessary information for recovering a deduplicated compressed package back to its original format. *Ownership Verifier* verifies the client's ownership to prevent an attacker from illegally accessing the file by uploading a fake metadata list. *File Chunker* and *Chunk-level Deduplicator* perform conventional chunk-level deduplication.

2) Deduplicating Non-Solid and Solid Packages: Figure 2 also describes an overview of how Z-Dedup handles different packages generated by the non-solid or solid compression method to remove the redundant files. We list the main functional modules' labels (numbers) next to the structure maintained by Z-Dedup to indicate the interactive relationship between a particular functional module and its different operational steps. More details are discussed in Figure 3.

For the non-solid compression, each individual file is compressed independently. As a result, a non-solid compressed package has already embedded sufficient metadata information (i.e., each file's original length and checksum information) to easily locate a specific compressed file within the compressed package. Z-Dedup can directly detect and remove the redundant files within such a compressed package. On the other hand, for the solid compression scheme, multiple files are first combined into a single file by concatenating them in a certain order, then compression is performed on the single file to store the compressed stream into a compressed block. As such, a solid compressed package lacks adequate information to locate a specific compressed file within its compressed block because multiple files are scattered across the compressed block and boundaries between files are blurred at best.

In Z-Dedup, we inject metadata information (i.e., each file's original length and checksum information) per file into each compressed block, as detailed in Section III-B4, so as to help estimate the content similarity between any two different compressed packages by the percentage of the shared redundant files, (e.g., the higher the percentage the more similar to the two packages are to each other). The similarity detection in Z-Dedup entails estimating the likelihood of finding many duplicate files between the incoming package and those already stored in the cloud. Since extracting and separating the compressed stream of a specific file from a solid package incur very high overheads, Z-Dedup's similarity detection of solid compressed packages aims to ensure that an incoming solid compressed package is uncompressed only if it is likely to contain a sufficient number of duplicate files.

3) Workflow and Key Operations: Figure 3 illustrates an example of deduplicating and restoring a non-solid compressed package without considering the security issue that is discussed in Section III-C2. Initially, two non-solid compressed packages, P1 and P2, are stored in the cloud server, and their invariant metadata is stored in the *File Metadata Signature Store*. Now, a non-solid package P3 is ready to be stored to the cloud. P3 is parsed by the *Package Parser* to extract its files' metadata. Moreover, it also detects the locally redundant file (i.e., E) within P3. Meanwhile, files' compression order is implicitly maintained by the *Metadata List*. Based on *File Metadata Signature Store*, the *Redundant File Filter* can divide files into two different groups: redundant files' list (i.e., B, C and E) and unique files' list (i.e., A and D), where the former also records each file's order and pointer of the corresponding file. The *Client Side Deduplicator* will remove duplicated files to generate a *Deduplicated Compressed Package* that only contains A and D. At the server side, the compressed package recipe is generated and appended to the end of package by the *Compressed Package Recipe Constructor*. Finally, it will update the *File Metadata Signature Store* to incorporate P3's metadata information.

In order to restore a deduplicated non-solid package, Z-Dedup must reassemble a package whose files' order is the same as the original compressed package. It will scan from both sides of a deduplicated compressed package, in which the left side contains compressed unique files within itself and the right side contains the order and pointer information of the redundant files. These redundant files can be decompressed from other packages. Then based on files' order information, the o-

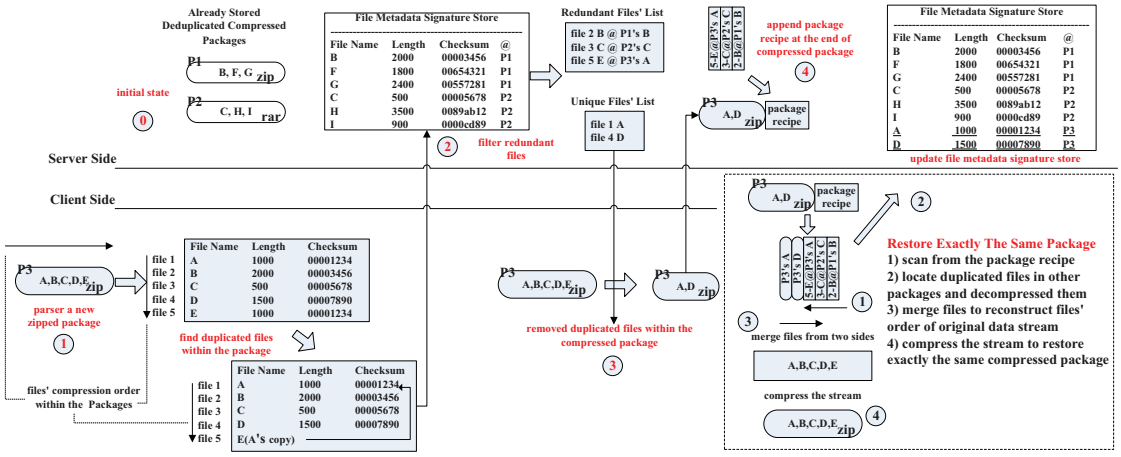


Fig. 3: An example on a non-solid package's deduplication and restoration.

riginal data stream will be reconstructed, and then compressed by the original set of compression parameters to restore the originally uploaded non-solid compressed package. We have summarized the main restoration process in Algorithm 2.

Now we assume that the incoming package P3 is a solid compressed one, whose file-level metadata information is already injected at the end of the package as described in Section III-B4. Once the redundant files' information is obtained, Z-Dedup will decide whether to remove the redundant files or not. This is done by estimating how much redundant data is contained within the incoming package P3 after querying the already stored contents in the cloud (Section III-B2). Based on the characteristics of source code packages, which are chosen to evaluate the effectiveness of Z-Dedup approach in handling the solid compression scenario, we find that it is simple but effective to find the most similar solid compressed packages to help remove the redundant files (i.e., find the most recent source code package to help remove redundant file). However, there is no way of quickly locating the compressed stream of a particular file in a solid package. Different from directly removing the redundant files within a non-solid package, once we have verified that the incoming solid package contains sufficient redundancy with the existing contents stored in the cloud, we need to decompress the solid package to locate and remove the redundant files first and then re-compress the remaining content into a solid package to send it to the cloud server. Moreover, it also maintains the concatenation order of compressed files in the original solid package. During the restore process, similar to the non-solid case, Z-Dedup will first restore the file stream and then solid compress these files to restore the original solid package.

4) Extending Compressed Packages' Layout: In most compression algorithms, files' metadata information such as the sizes of the original files and their checksums is already packed within the compressed packages for the purpose of data integrity. In Z-Dedup's implementation, we choose the combination of a file's SHA-256 value and its size to construct the file's signature. For those compressed packages that do not pack this information, we need to extend their data layout to record this information. It can be implemented by adding an optional feature to the compression tools to calculate and store

Algorithm 2 Restore a deduplicated compressed package.

```
FileCursor = 0
while FileStream is NOT FULL do
  Left = decompressFile(Package[LeftIndex])
  Right = findRedundantFile(Package[RightIndex])
  if (toMerge(Package[RightIndex]) is LEFT) then
    FileStream[FileCursor++] = Left
    LeftIndex++
  else
    FileStream[FileCursor++] = Right
    RightIndex++
  end if
end while
Parameter = getCompress(Package)
RestoredPackage = Compress(FileStream, Parameter)
```

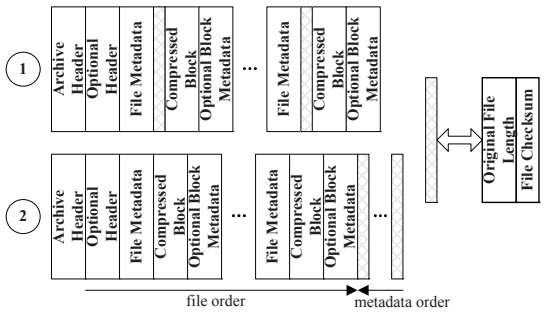


Fig. 4: Extending data layout of a compressed package.

this information in the compressed packages.

As shown in Figure 4, there are two different ways to pack this information. One is inserting it to file's metadata slot for each compressed file, which is similar to the structure of some compression algorithms that have already packed such metadata in their packages. The other is appending it at the end of a compressed package with a reverse order of the input stream. We choose the second way to pack files' metadata in this paper because it does not incur extra data movements within a compressed stream, which is compatible with existing compressed packages. Moreover, at the server side, each deduplicated compressed package will append its package recipe at the end of the deduplicated compressed package to help recover the compressed package, where more details are given in Section III-B1.

TABLE III: Collision rates for fingerprints of various lengths.

fingerprint		number of hashed elements such that {probability of at least one hash collision $\geq$ p}								
checksum	length	p=10^{-18}	p=10^{-15}	p=10^{-12}	p=10^{-9}	p=10^{-6}	p=0.1%	p=1%	p=25%	p=50%
32	64	6.1	1.9×10^2	6.1×10^3	1.9×10^5	6.1×10^6	1.9×10^8	6.1×10^8	3.0×10^9	4.3×10^9
64	96	4.0×10^5	1.3×10^7	4.0×10^8	1.3×10^{10}	4.0×10^{11}	1.3×10^{13}	4.0×10^{13}	2.0×10^{14}	2.81×10^{14}
128	160	1.7×10^{15}	5.4×10^{16}	1.7×10^{18}	5.4×10^{19}	1.7×10^{21}	5.4×10^{22}	1.7×10^{23}	8.5×10^{23}	1.2×10^{24}
160	192	1.1×10^{20}	3.5×10^{21}	1.1×10^{23}	3.5×10^{24}	1.1×10^{26}	3.5×10^{27}	1.1×10^{28}	5.6×10^{28}	7.9×10^{28}
224	256	4.8×10^{29}	1.5×10^{31}	4.8×10^{32}	1.5×10^{34}	4.8×10^{35}	1.5×10^{37}	4.8×10^{37}	2.4×10^{38}	3.4×10^{38}
256	288	3.2×10^{34}	1.0×10^{36}	3.2×10^{37}	1.0×10^{39}	3.2×10^{40}	1.0×10^{42}	3.1×10^{42}	1.6×10^{43}	2.2×10^{43}
384	416	5.8×10^{53}	1.8×10^{55}	5.8×10^{56}	1.8×10^{58}	5.8×10^{59}	1.8×10^{61}	5.8×10^{61}	2.9×10^{62}	4.1×10^{62}

C. Other Design Issues

1) Collision Analysis: Data deduplication requires a hash function that can generate a unique fingerprint for each unique data block. Suppose that a system contains N different files where each file is represented by a b-bit hash value with a uniform distribution, the probability p that there will be at least one collision is bounded by the number of pairs of blocks multiplied by the probability that a given pair will collide, which is $p \leq \frac{N(N-1)}{2} \times \frac{1}{2^b}$ [22].

Table III shows our estimated lower bounds on the numbers of hashed elements with different collision rates under various lengths of the checksum. For example, we can expect that, if the total number of unique files (i.e., hashed elements) is around several millions and the fingerprint is of 64 bits in length (32-bit chechsum plus 32-bit file length), the collision rate will be less then 10^{-6}. From this table, we can deduce that in a large-scale system, we might need to choose a checksum value whose length is larger than 128. In Z-Dedup, we propose to store files' SHA-256 checksum information at the end of each compressed package, which is explained in Section III-B4. The extra space overhead is negligible as explained in the example next. Assuming that the average file size is 32 KB with an average compression rate of 4, a 288-bit signature only adds 0.45% extra space overhead to the compressed package. Moreover, we can integrate the file metadata signature store into the existing lookup table of deduplication systems.

2) Security of Client Side Dedup: Z-Dedup performs deduplication at the client side without transferring all compressed data to the server side, which exposes a common security vulnerability in which an untruthful client may submit a faked metadata list to mislead the cloud server into believing that it owns the specific duplicated files already stored in the cloud server, enabling it to illegally access such duplicated files.

In order to overcome this security problem, the *Ownership Verifier* is added to verify if the client really owns the duplicated files as indicated by its metadata list. The key idea is to test whether the client really has the digital contents of the redundant files by asking the client to send a randomly selected sample segment of each claimed redundant file as follows. In Step 2 of Figure 3, when a server has identified redundant files that are claimed by a client's metadata list, it will randomly generate an address range within the size of each redundant file, i.e., a randomly selected sample segment of the file, and send this address information to the client. Meanwhile, it will decompress the corresponding sample data of the redundant files at the server side. In Step 3 of Figure 3, in addition to performing client side deduplication, the client needs to decompress the corresponding sample data of the redundant files specified by the sample segment address information, and send this data to the cloud server to verify the ownership of the corresponding files. A last step is to compare the sample data at the server side, where a match indicates the true ownership of the client and a fraudulent claim to ownership otherwise. Once the client has passed this test, its deduplicated compressed package will be accepted by the server.

IV. Evaluation

A. Experimental Environment

We use three desktops to emulate a simple cloud storage scenario. Two different clients run under $Ubuntu14.04$ and $Windows7$ with the EXT4 and NTFS file systems respectively. Both have installed some common compression tools that are listed in Table V. Throughout this work, we have collected the compressed packages generated from both the Linux and Windows platforms by using $Ubuntu14.04$ and $Windows7$ to emulate such a cloud storage scenario because they represent the most popular client platforms that will likely generate and store compressed contents to the cloud. One storage server runs under $Ubuntu16.04$ with all necessary compression tools to access the compressed packages received from different clients. All these three machines are equipped with one $i7-6700K$ processor, four 16GB DDR4 main memory, and one 6TB WD black HDD. We use Destor [23] as the traditional chunk-level deduplication engine. It is designed for backup applications with various chunk-level data deduplication algorithms. We have added a file-level deduplication process to implement Z-Dedup in this simulated environment.

Due to the privacy issue, we are not able to collect any real users' data in the cloud storage environment. In order to proof our Z-Dedup concept, we have collected several publicly available source code packages (i.e., coreutils, gcc, Linux kernel and Silesia corpus) plus some binary files, which include jpg, pdf, mp3, ppt, doc, and exe files, to form our datasets to test the effectiveness of integrating Z-Dedup into an existing deduplication-based storage system to detect and remove the data redundancy concealed by differently compressed packages. The reason why we collect these binary files is because these source code packages have a large number of small files, lack large files and all are represented by English characters, whose features make it suitable for them to be compressed into a solid compression package. Moreover, we randomly select some binary files to synthetically generate a compressed package workload (syn_data) to represent the scenario of multiple users sharing various types of files compressed by different tools. These binary files are usually used to comprise

TABLE IV: Sizes (KiB) of different compression formats under the Linux and Windows platforms, where the first number in each entry represents the size under Ubuntu14.04 and the second number represents the size under Windows7.

	coreutils-8.25	gcc-5.3.0	linux-4.5-rc5	silesia corpus
tar	49990/49990	601480/601480	642550/642550	206980/206990
xz	5591/5591	73815/73815	86287/86287	48047/48054
gz	12784/12784	121432/120154	132608/132609	66636/66640
7z	6169/5723	72256/71434	93561/89437	48563/48279
rar	12402/12401	148255/148177	156310/155135	53454/53451

the lossless data compression corpora, which include various types of files in different sizes, to evaluate the effectiveness of compression algorithms. We use several common compression tools to translate these compressed packages into various compression formats, which include both non-solid and solid compression schemes.

B. Results and Analysis

1) Analysis on Compressed Content: Table IV lists the various sizes of different formats for a specific version of selected datasets under both the Ubuntu and Windows platforms. We use the default parameters to convert the data package downloaded from the Internet into different package formats. We find that compression tools can significantly reduce the original digital contents' sizes. Especially, rar generates the non-solid compressed packages by default while 7z generates the solid compressed packages. Solid compression has achieved significantly higher compression ratios than non-solid compression. Moreover, source code packages have gained much higher compression ratios than binary files.

As shown in Table VI, we find that nearly all pairs of compressed packages have 0% data redundancy between them, a few pairs have 0.05%-7.63% data redundancy. The only exception is "tar.xz", which has generated 100% identical compressed packages from both the Ubuntu and Windows platforms, indicating xz-5.2.2 and xz-5.1.0α share the same core algorithm. We further verify that these packages share 0% redundancy with the compressed packages generated by xz-5.0.8. Although most packages have similar sizes across the two platforms, our study shows that: (1) except for "tar.xz", the compressed packages are fundamentally different from one another even under the same compression algorithm; (2) for the same digital contents, different compression algorithms will generate fundamentally different data streams; (3) a compressed package itself has very low data redundancy (0-0.05%) at the chunk level. *All these results indicate that traditional data deduplication methods cannot detect data redundancy in the compressed packages*.

2) Hidden Content Redundancy in Compressed Packages: In order to evaluate the real data redundancy among different compressed packages, we decompress various versions of compressed packages and apply the chunk level duduplication engine on them. As shown in Figure 5, we plot both local and global data redundancies, where the former represents the redundancy within the current version and the latter represents the redundancy among all versions. We find that most packages have very low local data redundancy within themselves. Although both the local and global data redundancy rates

TABLE V: Compression tools in the two client sites.

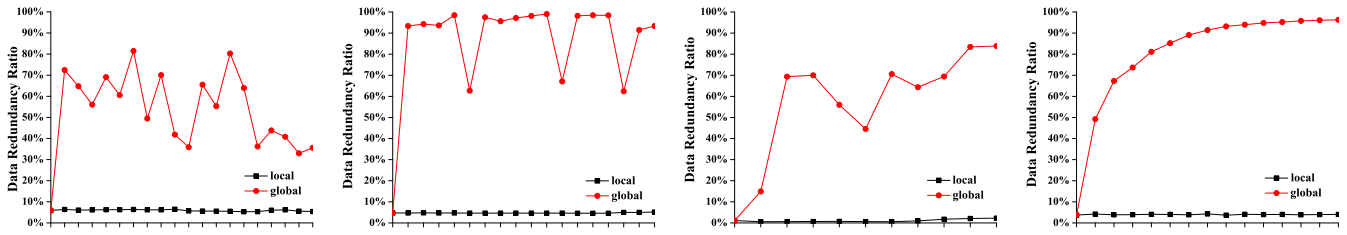
	tar	gzip	xz	zip	7z	rar
windows 7	1.28-1	1.6	5.2.2	3.0	15.09β	5.31
ubuntu 14.04	1.27.1	1.6	5.1.0α	3.0	9.20	4.20
ubuntu-16.04	1.27.1	1.6	5.1.0α	3.0	9.20	5.30β2

TABLE VI: Redundancy ratio (the total size of detected duplicate chunks divided by the total size of a compressed package) between any pair of compressed packages of the same content, indicates the compression one by the corresponding compression algorithm in the Ubuntu platform (row) and the other by the corresponding algorithm in the Windows platform. A zero or near-zero entry in the table means that there is no or nearly no detectable redundancy between the two compressed packages of the exact same content.

		coreutils				linux			
		xz	gz	7z	rar	xz	gz	7z	rar
coreutils	xz	100	0	0	0	0	0	0	0.05
	gz	0	7.6	0	0	0	0	0	0.05
	7z	0	0	0	0	0	0	0	0.05
	rar	0	0	0	1.0	0	0	0	0.05
linux	xz	0	0	0	0	100	0	0	0.05
	gz	0	0	0	0	0	5.6	0	0.05
	7z	0	0	0	0	0	0	0	0.05
	rar	0	0	0	0	0	0	0	0.24

vary over different versions, the global data redundancies are very high across different versions, indicating that high data redundancy exists among these packages. The test data sets are selected in such a way that they represent the most common types of modifications to data stored in the cloud. As such, we chose coreutils that selects continuous releases (including major, minor and patch) to represent small and frequently modified data sets, gcc that selects a major release after several minor or patch releases to represent the mixture of slightly and signifcantly modified data sets, and Linux that chooses discontinuous major or minor releases to represent the significantly modified data sets. In particular, the gcc and linux datasets contain a lot of stale and legacy code compared to coreutils, so their data redundancy is higher than the coreutils dataset. As illustrated in Figure 6, coreutils's data redundancy is lower than that of gcc and linux, gcc's data redundancy will drop when encountering any major release packages, and linux's data redundancy is low in the first few packages and increases in the last few packages, because the last few packages come from some minor releases that do not have too many modifications. *All these results indicate that there are a lot of duplicate files within these compressed packages, which can be detected and removed by Z-Dedup.*

3) Performance of Z-Dedup: File level redundancy ranges from 1.39% to 26.46% in coreutils, 95.21% to 97.55% in gcc, 9.04% to 55.55% in Linux, and 48.50% to 90.75% in syn_data. Z-Dedup is designed to identify and remove this kind of hidden redundant data. Our experiments show that Z-Dedup is able to reduce the size of compressed packages by 1.61%-35.78% in coreutils, 83.12%-98.75% in gcc, 11.05%-65.59% in Linux and 38.28% to 84.25% in syn_data with its file-level data deduplication. We find that some coreutils versions have major modifications made to most files, leading to a very low file-level redundancy. As a result, Z-Dedup can scan an incoming compressed package, to estimate the level of redundancy in the package by checking its metadata



(a) coreutils (20 versions) (b) gcc (18 versions) (c) linux (11 versions) (d) syn_data (15 versions)

Fig. 5: Real data redundancy throughout different versions of decompressed packages.

TABLE VII: Various checksum algorithms' computation overheads with Intel 6700K processor.

Algorithms	CRC32	CRC32C	MD5	SHA-1	SHA-256	SHA3-224	SHA3-256	RIPEMD-160	RIPEMD-256	BLAKE2s	BLAKE2b
MiB Per Second	610	5271	742	695	341	475	453	302	584	769	1128
Cycles Per Byte	6.3	0.7	5.1	5.5	11.2	8.0	8.4	12.6	6.5	5.0	3.4

information and opt out performing file level deduplication when there is very little file level redundancy for this package. However, extracting metadata information from these low-redundancy compressed packages is still necessary and in fact important because there could be high file-level redundancy in the compressed packages of the subsequent versions.

After evaluating the effectiveness of Z-Dedup, we want to evaluate the extra overheads that come with this approach. In Table VII, we obtain the computation overheads of several popular hash algorithms used to generate the checksum values within the compressed packages with Intel's 4GHz 6700K processor. We find that SHA-256 incurs $1.79\times$ computation overhead than the traditional CRC32 algorithm. By choosing SHA-256 as the checksum algorithm for Z-Dedup, we can also reuse the existing fingerprint table, which contains the SHA-256 hash values, in existing data deduplication systems.

We use the source code packages that contain a large number of small files to evaluate the overheads on maintaining and processing the necessary metadata in the Z-Dedup approach. In general, using a source code package as a target to simulate deduplicating a non-solid compressed package can incur the worst overheads in deduplicating compressed contents because it requires a number of operations proportional to the number of files in the package but saving very little space because all the files are small in size. At the same time, we use the synthesized packages that contain various types of large size files to more realistically evaluate the overheads on deduplicating non-solid compressed packages. We find that Z-Dedup will slightly increase the size of compressed packages because it needs to inject the SHA-256 checksums within the compressed packages. In fact, this space overhead is found to be less than 0.45% of the total compressed package's size. *All these results indicate that Z-Dedup can reduce a significant amount of redundant data in compressed packages in a traditional deduplication-based storage system without significant overheads.*

4) Network Traffic Reduction: We show the amounts of data transferring to the server side under different workloads in Figure 6, while each data is normalized to its corresponding uncompressed package's size on both non-solid and solid compression modes to learn the benefits of data reduction for network transmission. In particular, the 7z and rar compressions will reduce the amount of coreutils data to 10.69% and 23.76% of their original sizes in geometric average, while Z-Dedup can further reduce these to 4.16% and 9.82% by exploiting the data redundancy embedded in these compressed packages. Moreover, the 7z and rar compressions will reduce the amount of gcc data to 11.30% and 23.28%, linux data to 13.32% and 23.01%, syn_data data to 36.83% and 41.37% in geometric average; while Z-Dedup can further reduce these data to 0.62% and 1.28% in gcc, 4.69% and 8.30% in linux, and 4.16% and 4.70% in syn_data. Note, binaries files in syn_data show different redundant data curves than source code packages because these data usually have lower compression ratios than plain-text files. *From this evaluation, we further verify that Z-Dedup can reduce most data transferring to the network*

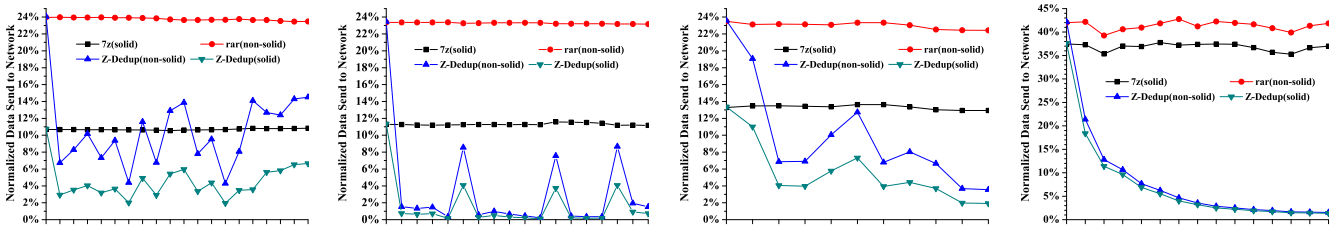
Besides the reduced data network traffics, we have also evaluated Z-Dedup's total processing times that include the local processing time and network transmission time. Z-Dedup can reduce the processing time by 10.71% to 31.52% without verifying file's ownership. It will reduce this time by 2.74% to 21.32% after adding the ownership verification.

V. Conclusion

In this paper, we designed and implemented Z-Dedup, a novel deduplication technique designed to detect and remove data redundancy in compressed packages for which conventional chunk-fingerprint based deduplication approaches have failed. The main idea of Z-Dedup is to exploit some key invariant information embedded in the metadata of compressed packages, such as file-based checksum and original file lengths, as the unique signatures of individual compressed files. The design and implementation of Z-Dedup also address a potential security vulnerability due to client-site compression, as well as packages generated by both non-solid and solid compression methods. Our evaluation of a Z-Dedup prototype shows that it reduces up to 98.75% more redundant data in compressed packages than traditional deduplication system.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable feedback and constructive suggestions. This research is partially supported by Chongqing High-Tech Research Program (cstc2016jcyjA0274), National Natural Science Foundation of China(61402061), Fundamental Research Funds for the Central Universities (2018CDXYJSJ0026), research grant from NetApp and the U.S. National Science Foundation (NSF)



(a) coreutils (20 versions) (b) gcc (18 versions) (c) linux (11 versions) (d) syn_data (15 versions)

Fig. 6: Data to be sent to the cloud server, while each number is normalized to the total size of its uncompressed package's size.

under Grant Nos. CCF-1704504 and CCF-1629625. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] J. Gantz and D. Reinsel, "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east," https://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf, 2012.

[2] EMC, "Managing storage: Trends, challenges, and options(2013-2014)," https://education.emc.com/content/_common/docs/articles/Managing_Storage_Trends_Challenges_and_Options_2013_2014.pdf, 2013.

[3] B. Walker, "Every day big data statistics - 2.5 quintillion bytes of data created daily," http://www.vcloudnews.com/wp-content/uploads/2015/04/big-data-infographic1.png, 2015.

[4] P. Deutsch, "Rfc1951 deflate compressed data format specification version 1.3," https://www.ietf.org/rfc/rfc1951.txt, 1996.

[5] I. Pavlov, "7-zip," http://www.7-zip.org/.

[6] Y. Tan, H. Jiang, D. Feng, L. Tian, Z. Yan, and G. Zhou, "Sam: A semantic-aware multi-tiered source deduplication framework for cloud backup," in *Proceedings of the 39th International Conference on Parallel Processing*, 2010, pp. 614–623.

[7] Z. Yan, H. Jiang, Y. Tan, and H. Luo, "Deduplicating compressed contents in cloud storage environment," in *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems*, 2016.

[8] Cisco, "Cisco visual networking index: Forecast and methodology, 2015c2020," http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html, 2016.

[9] C. Policroniades and I. Pratt, "Alternatives for detecting redundancy in storage systems data," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATC)*, 2004, pp. 6–6.

[10] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[11] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[12] P. Inc., ".zip file format specification (version 6.3.4)," https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT, 2014.

[13] A. Roshal, "Rar," http://www.rarlab.com/.

[14] J. Bonwick, "Zfs deduplication," https://blogs.oracle.com/bonwick/entry/zfs_dedup, 2009.

[15] P. Storage, "Pure storage flasharray," http://info.purestorage.com/rs/225-USM-292/images/Pure_Storage_FlashArray_Datasheet.pdf, 2015.

[16] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, 2008, pp. 18:1–18:14.

[17] M. Vrable, S. Savage, and G. M. Voelker, "Cumulus: Filesystem backup to the cloud," in *Proccedings of the 7th Conference on File and Storage Technologies*, 2009, pp. 225–238.

[18] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, 2011, pp. 1–1.

[19] Y. Tan, H. Jiang, D. Feng, L. Tian, and Z. Yan, "Cabdedupe: A causality-based deduplication performance booster for cloud backup services," in *2011 IEEE International Parallel Distributed Processing Symposium*, 2011, pp. 1266–1277.

[20] X. Lin, G. Lu, F. Douglis, P. Shilane, and G. Wallace, "Migratory compression: Coarse-grained data reordering to improve compressibility," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, 2014, pp. 257–271.

[21] X. Lin, F. Douglis, J. Li, X. Li, R. Ricci, S. Smaldone, and G. Wallace, "Metadata considered harmful ... to deduplication," in *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems*, 2015, pp. 11–11.

[22] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," in *Proceedings of the Conference on File and Storage Technologies (FAST)*, 2002, pp. 89–101.

[23] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan, "Design tradeoffs for data deduplication performance in backup workloads," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015, pp. 331–344.