ELEKTROTEHNIŠKI VESTNIK 82(5): 233–242, 2015
ORIGINAL SCIENTIFIC PAPER

Deduplication in unstructured-data storage systems

Andrej Tolič[1,†], **Andrej Brodnik**[1,2]

[1] *University of Ljubljana, Faculty of Computer and Information Science, Večna pot 113, 1000 Ljubljana, Slovenia*
[2] *University of Primorska, Andrej Marušič Institute, Muzejski trg 2, 6000 Koper, Slovenia*

[†] *E-mail: andrej.tolic@fri.uni-lj.si*

Abstract. The paper addresses the issue of deduplication, a process of identifying and eliminating redundancy in large data sets of unstructured data. Storage systems for the unstructured data handle an ever increasing amount of information, a large portion of which may be redundant. While the well-known methods, such as entropy encoding, solve the issue to a certain extent, they fail to detect and eliminate the redundant data more than a few gigabytes apart. The basics of deduplication are explained and a detailed description is given of the steps involved. The state-of-the-art deduplication techniques are described.

Keywords: deduplication, redundancy elimination, storage systems, distributed systems, Bloom filter

Deduplikacija v shranjevalnih sistemih za nestrukturirane podatke

Deduplikacija, proces razpoznavanja in odstranjevanja redundance v velikih množicah nestrukturiranih podatkov, je predstavljena v tem članku. Shranjevalni sistemi za nestrukturirane podatke obravnavajo vedno večje količine informacij, velik del katerih je lahko podvojen. Čeprav dobro znane metode, kot je entropijsko kodiranje, rešujejo problem do neke mere, ne zmorejo zaznati in odstraniti odvečnih podatkov več kot nekaj gigabajtov narazen. To težavo naslavlja deduplikacija. V članku razložimo osnove deduplikacije in podamo podroben opis korakov izvedbe. Na poti opišemo najsodobnejše tehnike deduplikacije.

1 INTRODUCTION

Modern storage systems for unstructured data such as distributed file systems [1], key-value stores [2] and archival/backup systems in many cases exhibit data redundancy. The issue is addressed by deduplication, a process of identifying and eliminating duplicated data [3]. Often, whole files are duplicated in enterprise and cloud environments, while the subfile-level redundancy is also common.

While redundancy elimination with entropy encoding algorithms (e.g. ZIP compression) is efficient in eliminating the redundant data within logical groups of files up to a few gigabytes in the size, there still remains duplicated data across the storage system. The amount of data in these systems is typically measured in terabytes or petabytes. Detecting and eliminating the remaining redundancy across the whole system can thus lead to significant space savings – depending on the data, a deduplicated size of less than ten percent of the original size is not uncommon. Furthermore, in the light of the cloud-based services and inter-connectivity, large amounts of data are moved across networks all over the world. If redundancy could be eliminated, the network bandwidth would also be preserved [4]. However, any additional operation the storage system must perform when the data is accessed can lead to a significant performance degradation. Therefore, the challenge is to design a deduplication technique to detect as much redundancy as possible while having the minimal performance footprint.

The paper is organized as follows. In Section 2 we present the basics of the unstructured storage systems. Section 3 is an introduction to deduplication where different characteristics of deduplication are presented. A general architecture of the deduplication systems is introduced in Section 4, with individual steps described further in Sections 5, 6 and 7 along with examples of the well-known deduplication techniques. Section 6 details two of the steps for which we believe they should be explained together. Section 8 describes how deduplication techniques affect and are affected by reading operations. Section 9 gives a short overview of deduplication in a distributed setting. We conclude in Section 10 where we mention some open problems tackled by the current research.

2 UNSTRUCTURED STORAGE

When talking about the storage systems for unstructured data, we refer to the systems that are not aware of the structure of the data they store beyond what is implied by the storage API. This is in contrast with the structured

Received 1 June 2015
Accepted 16 September 2015

storage, like relational databases, where the system is (at least to some extent) aware of the structure of the data it holds in order to provide the users with higher-level operations.

A typical example of the unstructured-data storage are the POSIX file systems. To the users they are mainly known by the operations their API [5] exposes, such as *create, delete, open, close, write, read,* etc. Another type of the unstructured-data storage with a simpler interface than POSIX FS is the *object-based storage* [6], also known as the *key-value* storage [2]. It could be described as an advanced block-level storage, where the objects (blocks) have a variable length, a more flexible naming scheme and possibly support advanced functions, such as object attributes, replication, snapshots, etc. Compared to POSIX FS, the main difference is the lack of a tree-like naming structure based on directories. Instead, the objects are referenced only by their names (keys).

Certain object-based systems are classified as the NoSQL databases which usually expose structured interfaces. Most NoSQL systems offer higher-level data models on top of a key-value model, although the difference can be blurred with some systems. We also note that the object-based storage should not be confused with the object database systems which replace the relational model with the one based on the objects as they are known in object-oriented programming.

A distinction is also made between the systems intended for the **primary workloads** and those for the **secondary workloads**. The latter are characterized by an infrequent access consisting mostly of write operations, which is the case with the archival and backup storage systems. The primary-workload storage systems, on the other hand, are known for their frequent data access where read operations usually prevail. Even today, most of the deduplication techniques are intended for the secondary workloads, but research into the techniques appropriate for the primary workloads is increasing.

As with many other computer systems, the unstructured storage systems can be either **centralized** or **distributed**.

3 DEDUPLICATION BASICS

The discussion applies to both the centralized as well as the distributed storage systems unless stated otherwise. An example of the former are the local file systems (NTFS, Ext4, BTRFS), and of the latter the distributed file systems such as NFS, Google file system [1], Ceph [7]. It should be evident from the context where the distinction is important, specifically when we present a distributed deduplication.

3.1 File- and subfile-level deduplication

A division could be made between the **file-level** deduplication and the **subfile-level** (sometimes called block-level) deduplication. Note that we use the word file to mean a finite-length array of bytes. It is thus any unit of storage with respect to the storage API, such as an object in an object-based storage. The literature typically refers to these units as files since POSIX FS is probably the most widely recognized storage-system interface for the unstructured data.

The file-level approach checks for identical files. It is also known as a content-addressable storage (CAS). It is a simple technique but effective in certain scenarios [8], especially cloud stores, where many files are stored multiple times. Consider a cloud-based storage service. Users all over the world store the same music files and computer-science students store the same textbooks and so on. If the storage system detects these files as duplicates, only the first copy must be stored and later referenced when users (attempt to) write the same files to the system. In fact, the network bandwidth can also be preserved since the users do not even have to write the file to the system if the client software first sends a **hash of the file** to the system, upon which it receives a reply that such a file is already stored. This applies if the client has an access to the whole file when it attempts to write it to the system. The system then only stores a reference to this file.

The deduplication techniques usually do not employ a byte-by-byte comparison in order to identify identical files (or parts of files as we will see later), but rather rely on the hash functions with sufficiently large hash values and collision resistance.

The subfile-level deduplication tries to detect redundancy within files. Consider office documents within an organization, all having the same logo embedded within them or virtual-machine disk-image files containing the same or similar virtual files. This kind of the files are not identical and so redundancy would not be detected by the file-level methods. The subfile-level methods attempt to partition the files into continuous subarrays, usually called **chunks** or segments, and detect identical ones (see Figure 1). We consider two files similar if they can be partitioned so that they share many of the resulting chunks. Every deduplication technique takes a slightly different approach on how to identify the identical data, but they all rely on **features**, the values calculated from the partitioned data and used to query (during redundancy detection) and later possibly update (after redundancy detection) the **chunk index**. The most common approach present in practically all the techniques is to use the hash values of the chunks as features (as in the example in Figure 1), but additional features can be calculated as well, which we will see in Section 6. The features therefore depend on partitioning and affect the chunk index, a data structure (or several of them) holding information about the already stored data. This information typically contains a list of the hash values of

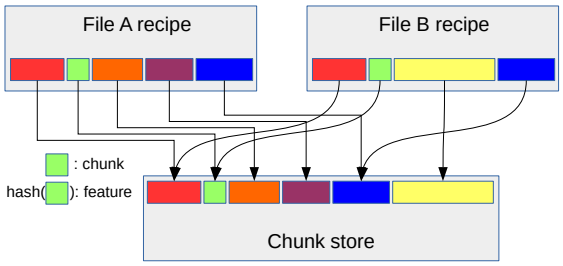

Figure 1. Typical subfile-level deduplication technique identifies the redundant parts of files by partitioning them into chunks and calculating the features from chunks. In the simplest and most common case, the features are the hash values of the chunks and are used to identify the identical already stored data. Unique chunks are stored in a chunk store while the file recipes contain pointers to the chunks used to reassemble the files when reading them.

the stored chunks (which are almost always the features as well), reference counts for every chunk, and possible additional features. The storage system keeps the chunks in the **chunk store** and maintains the **file recipes**, the metadata containing references (pointers) to the chunks that comprise the file in question. These references also take space, so the chunks of the sizes smaller than a kilobyte are usually not reasonable.

3.2 Inline and offline deduplication

The **inline deduplication** (also called online) is done as the data is written to the storage system. The word *inline* is used since in the storage context there is a data path or a data line by which the data is written to the system. In the **offline deduplication** (also called out-of-line), the data is first written to the system and then deduplication is done on the data "at rest" in the batch. The downside of the offline deduplication is the need for a temporary storage where the redundant data is stored before being deduplicated. The upside is a better performance, since deduplication can be performed in the background when the user access to the data is minimal (e.g. during the night).

4 General Architecture of the Deduplication Systems

The deduplication techniques are often designed as part of a storage system and tailored to specific workloads. This results in a heterogeneous architecture and intrinsic properties. Despite this, the general architecture, which is roughly followed by most systems, can be described.

Since deduplication mainly interferes with the write operations (it is then when redundancy detection and elimination happens), we define the general architecture as a sequence of steps performed on the data written to the storage system. These steps are performed either as the data is being written (inline deduplication) or at some later time (offline deduplication). In Section 8 we will describe how deduplication interferes with the read operations as well as how the read performance considerations affect the deduplication design.

Deduplication is thus performed in the following steps (cf. Figure 1):

1) **Partition** the incoming file into either the fixed-length chunks or variable-length content-defined chunks.
2) **Calculate the features** for a file from the partitioned chunks. In almost all cases this means calculating the hash values of the chunks. However, different techniques improve on this with several approaches as we will see in Section 6.
3) **Look up features** of a file in an appropriate data structure (also called the chunk index) to identify the matching data.
4) **Commit write** by storing the non-duplicated data chunks to a permanent storage and updating the chunk data structure. Lastly, a file recipe, the metadata describing contents of the file, is generated and stored.

The described steps represent a chain where the output of one operation is used as an input to the next. It is essential for consistency that the data chunks are stored and their data structures updated before the file recipes are generated and stored, otherwise the recipes would reference the non-existing data. Furthermore, the file-recipe generation and storage must be done atomically, since this is the only information on what is contained in the file. If this information is incorrect, it is possible to detect an error (some systems store the whole-file hashes or file length), but it cannot be recovered from, although the contents themselves (in the form of chunks) are stored in the system.

5 Partitioning

Partitioning is affected by the feature computation and granularity of the redundancy detection (the file-level vs. the subfile-level). The file-level deduplication skips this step since the second step (feature computation) is done on a file as a whole by simply calculating the hash value of a file.

On the subfile-level, one possibility is to partition the data into the **fixed-length chunks**, which is appropriate for certain scenarios [9]. The problem with the fixed-length chunks is that a simple insertion or deletion at the beginning causes the chunks that follow to shift, thus producing different hash-based features for almost identical data. This is depicted in Figure 2. The fixed-size chunking can, however, be used efficiently in the similarity-based techniques as we will describe in Section 6.2. The issue is addressed by the **content-defined**

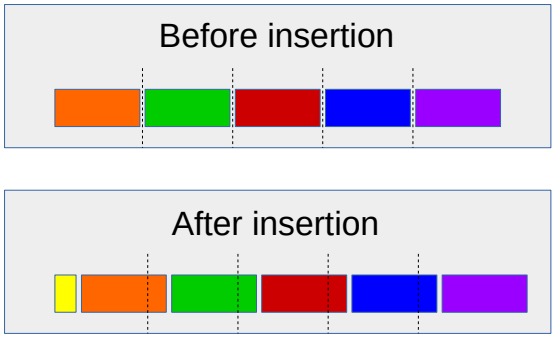

Figure 2. Data-shift problem with the fixed-length chunking after insertion at the beginning.

partitioning using the sliding window technique [10], [4]. The idea of this technique is to identify the variable-length chunks whose boundaries are selected whenever a certain pattern is observed. The hash value is computed over a k-byte window, which is slided byte-by-byte through the file. Whenever the lower d bits of the hash value equal a predefined constant r ($f \equiv r \pmod{D}$), a chunk boundary is marked at the start or the end of the window (see Figure 3). The value $D = 2^d$ is the divisor and r is the remainder, usually set to 0. Assuming the uniformly random data, this will happen every $D = 2^d$ bytes, which is thus the expected average size of a chunk. Pathological cases do exist. For example, a long sequence of the zero bytes would probably (depends on the hash function and values r and d) never match the boundary condition. Conversely, the k-byte chunks all matching the boundary condition can be near each other, thus producing the small-size chunks. This is usually addressed by setting the minimum and maximum chunk sizes as proposed in [4]. If the boundary is found sooner than the minimum size, it is skipped. If the boundary is not found when scanning the maximum size, it is selected even if not matching the boundary condition. We usually want the average chunk size from one to several tens of kilobytes, making the typical choice for d somewhere between 10 and 15.

To further reduce the chunk size variations and improve detection of the identical chunks, a TTTD (two thresholds two divisor) partitioning approach is proposed [11]. As the name suggests, the algorithm uses two thresholds (minimum and maximum sizes of the chunks) as already proposed in [4] and just described. Additionally, the approach introduces a second divisor D' which is roughly half the size (one bit shorter) of the main divisor D. The purpose of the second divisor is to find the backup boundaries in case the main divisor fails to do so. In that case, if the second divisor succeeds, the resulting chunk is closer to the expected size, as well as being a better candidate for deduplication since its boundary is defined by observing a pattern as opposed to being cut off at the maximum threshold.

For a file of length n bytes, a hash is computed at $n - k + 1$ different but overlapping positions of a window. Typical window sizes k are several tens of bytes. For example, the authors in [4] use a 48-byte window. We should emphasize that these hash values only serve the purpose of identifying the variable-length content-defined chunks and are in no way connected with the hash values computed later on these chunks (or groups of chunks as in some examples) for the purpose of feature computations.

Since the typical choice of the value for d is slightly over 10 and $n-k+1$ hash computations are required to partition a single file, we are interested in the fast hash functions with a uniformly random output and possibly short (but of the size at least D) output values. Any hash function can be used for this, but since we slide a window byte-by-byte, we are calculating the hash values of many overlapping subarrays.

Rolling hashes allow for a faster computation of the new hash value given only the old hash value, the bits that are removed from and added to the window. Since we slide the window one byte at a time, eight bits are removed and added in each step. The *Rabin fingerprint* [12] is a type of the rolling hash function operating on bit strings. An n-bit message $m_0, \ldots, m_{n-1}$ is represented as a polynomial

$$p(x) = m_0 + m_1 x + \ldots + m_{n-1} x^{n-1}$$

of degree $n-1$ over a finite field GF(2). We then calculate the k-bit hash value as a remainder of the division of $p(x)$ by irreducible polynomial $i(x)$ of degree k over a finite field GF(2). The remainder polynomial is of degree $k-1$ and interpreted as a k-bit value. Even though

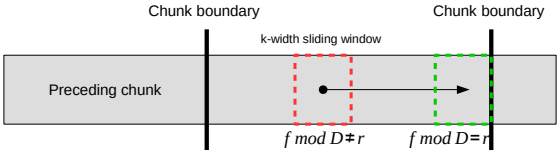

Figure 3. Sliding-window technique for the chunk-boundary detection.

the Rabin fingerprints operate on bit strings, it is possible and even desirable (due to the better performance) to process more than one bit at a time. When operating on larger window sizes, fast implementations consider the register size of the underlying architecture, which today is usually 32 or 64 bits. For our purposes, eight bits are processed at a time. If l bits are processed at a time, two precomputed tables with 2^l entries (representing all possible l-bit strings) are used to avoid a repeated computation of the l rightmost and l leftmost terms of the polynomial. These terms represent the bits that enter

and leave the sliding window over which the fingerprints are calculated.

The first well-known system using deduplication with the content-defined chunking is LBFS [4], a low-bandwidth file system. The system does not use deduplication to preserve only the storage space but the network bandwidth as well. It provides the same semantics as the well-known network file systems, such as NFS or AFS, but uses less network bandwidth to transmit changes between the clients and servers. This is achieved by a heavy client-side caching and deduplication based on the content-defined chunking. Before side A (client or server) sends data to side B (server or client), side A partitions the file into chunks, calculates the hash values of these chunks and sends these hash values to side B which checks if it already has them stored. Side B reports back the hash values of the missing chunks (can be all of them for a completely new data), and only these are actually sent over the network by side A, thus preserving the bandwidth. LBFS is therefore an early example of the use of deduplication as one of the core **WAN optimization** techniques. In the LBFS case, deduplication is specific to the application layer protocol (network file system), but the WAN optimization devices use the same approach when deduplicating the packet payloads at the network layer independent of the upper layer protocols (HTTP, NFS, SMTP, *etc.*).

6 Feature computation and lookup

Although we listed the feature computation and lookup as two separate steps, we describe them together for better understanding.

The basic operation of the **feature computation** is hashing. The features are computed for each file separately. In the file-level deduplication, a feature is simply the hash value of the whole file, since there is no partitioning. There are nuances between different techniques, but the fundamental part where the redundancy detection happens is in checking if a specific hash value (which serves as a feature) is already present in the system. Once a duplicate chunk is detected, most techniques do not proceed with a byte-by-byte comparison, although two different chunks or files can be hashed to the same value. This is known as *hash collision*. Due to the birthday paradox, hash collisions become likely when $2^{\frac{n}{2}}$ different values are hashed by an n-bit hash function [13]. If we were to use a 128-bit hash function for the file-based deduplication and the average file size were 1 kilobyte, such a system could reliably identify the duplicates by hashing, as long as less than 2^{64} files were stored. For example, the 2^{50} one kilobyte files amount to 1000 petabytes of data.

Feature lookup is supported by an index data structure holding the chunk metadata (hash value, reference count, *etc.*). Since this data structure in its naive form takes the space linear in the total size of the stored data, it is quickly too large to fit into RAM, so it needs to be stored on the disk. This is a problem, since deduplication should cause as little additional disk accesses as possible, otherwise it degrades the performance. The random access times to RAM, SSD (solid-state drive) and HDD (hard-disk drive) are measured in tens of nanoseconds, tens of microseconds and milliseconds, respectively. The SSD drives are thus a "middle ground" between RAM and HDD and are actually employed as large caches in some deduplication techniques. However, limiting most accesses to RAM is still desirable.

The following two subsections present two approaches that tackle the problem by reducing the number of disk lookups. The first relies on a probabilistic data structure called the Bloom filter described at the beginning of the subsection. The second approach relies on multiple (usually two) granularities of feature computation, where at a higher level, the *similar* (not identical) data is identified and then deduplicated at a finer granularity. Example techniques are described for both approaches.

6.1 Avoiding disk I/O with Bloom filters

The Bloom filter [14] is a probabilistic data structure used to support set membership queries. Insertion and lookup take a constant time, while removal is not possible with an ordinary Bloom filter. The structure takes the $O(m)$ space as a bit array of length m, where m is a parameter set by the user. An empty filter has all the bits set to 0. When an element is added to the filter, it is hashed by k different hash functions returning values h_i, where $0 \leq h_i < m, \quad i \in [1, k]$. Each of the k values is used as an index into an array of bits (we assume zero-based indexing), so that the bits at the corresponding index positions are set to 1. The lookup operation is performed in a similar manner. An element whose membership we wish to check is hashed by the same k hash functions returning the index values into a bit array. If the bits at all the positions are 1, the filter returns `true`, otherwise `false`. This means there are no false negatives, since if an element was added to the filter, the corresponding bits would be definitely set to 1. However, false positives are possible, since the bit values at the k positions could be set to 1 by a different combination of other elements. In fact, the probability of a false positive after inserting the n elements is approximated by $(1 - e^{\frac{-kn}{m}})^k$ [15].

Probably, the most cited approach to the unstructured data deduplication is described in [15]. When the files are written to the system they are partitioned using a variable-length content-defined chunking. The feature computation results in chunk descriptors comprised of the chunk SHA-1 hash value, its size and some optional information. The feature lookup is accelerated by the *summary vector*, an in-RAM Bloom-filter structure that

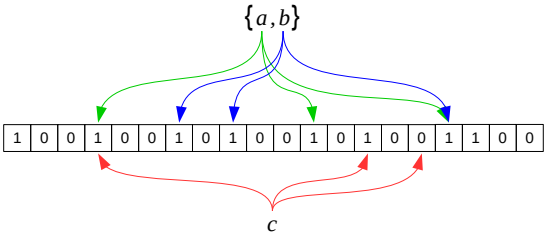

Figure 4. Example of the Bloom filter with a and b as members of the set, and c not a member.

reduces the number of accesses to the chunk index stored on the disk when a chunk is not duplicated.

The proposed technique was designed for the backup workloads, where the spatial locality of chunks is very common, since the same files (consisting of a sequence of chunks) are repeatedly written to the system. A stream-informed chunk layout is thus used to preserve the spatial locality of chunks and their metadata. This means that the neighbouring chunks as well as their descriptors are stored sequentially, which enables a locality-preserving cache. Whenever a retrieval from the on-disk chunk index for a single chunk descriptor is needed, a neighbouring group of the chunk descriptors is transferred to a cache, expecting that other chunk descriptors from the group will be queried soon.

6.2 *Similarity-based techniques*

Another solution to the problem of the large chunk-index data structures are similarity-based techniques. We first present the general idea and some theoretical background followed by a description of two such techniques.

Instead of only doing exact matching (looking for identical chunks) at a single granularity, the similarity-based techniques take a hierarchical approach, where the features are computed in at least two levels, with the higher level(s) using similarity matching to identify groups of similar data (often called superchunks) to focus on at a lower-level (finer) granularity. How are these groups (superchunks) formed and how are the higher-level features computed on them will be discussed with each of the two presented techniques. The features at the lowest level (chunk-hash values) are then exactly matched. Since the higher-level features are computed at a coarser granularity on groups of data, they are fewer. This enables us to store parts of the index in RAM (higher level) and parts on the disk (lower level). Usually, two levels are used. The choice of a similarity-based technique also affects the partitioning step, which can be carried out with either the content-defined or fixed-size chunking. See Figure 5 for an example of the two-level hierarchy.

The *Jaccard index* (also called the Jaccard similarity coefficient) is a measure of similarity between two sets, A and B, and is defined as

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}.$$

To avoid calculating the intersection and union for every pair of sets, Broder [16] defines the notion of resemblance (similarity) between the two data sets based on an estimate of the Jaccard index. Let h denote a hash function that maps the elements of some set S to integers (a binary string could be interpreted as an integer). Let $h_{\min}(S)$ denote element x of S with the minimum value of $h(x)$. Broder shows that the Jaccard index of sets A and B could reliably be estimated as

$$\Pr[h_{\min}(A) = h_{\min}(B)].$$

Stated otherwise, the estimate of the Jaccard index is defined as a probability that two sets have the same minimum hash element, the element that maps to the lowest hash value. This is also known as **MinHashing** [16]. The idea behind this is that if two sets share many of the same elements, when permutation is applied that maps the elements to the set $1 \ldots 2^n$, it is likely the same element from both sets will be mapped to 1. In practice, the n-bit random hash functions are used in place of permutations.

In order to evaluate the set resemblance, one hash function and only the minimum hash value have a too high variance, since random variable r in the above probability can only take two values, i.e. the minimum hash value is either the same for both sets ($r = 1$) or not ($r = 0$). This is addressed by two approaches. The first one is to use k different hash functions instead of just one. The resemblance is then estimated to be $\frac{m}{k}$, where m is the number of hash functions for which the minimum hash value is equal for both sets. This is actually proposed by Broder in [17]. The second one is to use one hash function, but calculate the k different values for each set instead of just one. For example, the k least-hash values instead of just the minimum. This is proposed in the Broder's first paper [16]. Again, the resemblance is estimated to be $\frac{m}{k}$, where m is the number of the hash values which are equal for both sets. However, in this case the resulting k values are not random, which might be a problem, but can be mitigated. This is done in the technique presented in [18] described in the next paragraph. In both approaches, the k values – obtained by either storing the minimum element for the k hash functions or storing the k smallest elements for one hash function – can be computed and stored for each set independently, and later compared as necessary. These k values are also called a *sketch* of the set. A more thorough and mathematically precise treatment of MinHashing can be found in [16] and [17]. The similarity-based deduplication techniques use the

variants of MinHashing to detect the similar data before exact matching is done.

The authors in [18] present a similarity-based deduplication technique. For the purpose of a higher-level similarity detection, the input files are partitioned into fixed-length superchunks of size 2^{24} bytes (16 MB). A 512-byte wide window is slided byte-by-byte over a superchunk to efficiently compute the Rabin hashes of the $2^{24} - 511$ different (but overlapping) chunks. We denote k_i to be the Rabin hash value calculated over the chunk starting at byte position i within the superchunk, with $0 \leq i \leq 2^{24} - 512$. Let i_1, i_2, i_3 and i_4 denote the index positions of the chunks for which hash values k_{i_1}, k_{i_2}, k_{i_3} and k_{i_4} are the largest. One could use $k_{i_1}, \ldots, k_{i_4}$ as the superchunk features in order to identify similar superchunks, that is, the superchunks that have at least one of the four features in common. However, the authors notice that since values $k_{i_1}, \ldots, k_{i_4}$ are the largest among all the $2^{24} - 511$ computed values within a superchunk, they are not uniformly distributed over the output range of the hash function, but tend to belong to a small set at the higher end of the output range. This would lead to collisions, with different chunks having the same hash values. The identical features would not correspond to the identical data, which would lead to a similarity detection where there would not be any. Therefore, the authors propose to shift index positions $i_1, \ldots, i_4$ by a small amount m (they suggest $m = 8$) and use $k_{i_1+m}, \ldots, k_{i_4+m}$ as features of the superchunk. Even though, these four hash values are computed over similar chunks – in each of the four pairs, the shifted and original chunk differ in (at most) the first and last m bytes, thus having (at least) $512-2m$ bytes in common – they generally differ significantly due to the nature of the hash functions. The authors propose to use a Rabin hash with a 56-bit output. Along with every 7 byte feature, a pointer to the on-disk location of one or more of the corresponding superchunk(s) is listed (also 7 bytes for a total of $2^{56}2^{24} = 2^{80}$ bytes), which amounts to at most (since a single feature may be shared by several superchunks) 14×4 bytes per a 16 MB superchunk. Due to this ratio of $\frac{56 \text{ bytes}}{16 \text{ MB}}$, the higher-level features (and the accompanying metadata) can be stored in RAM and upon arrival of new data used to identify the similar already stored superchunks. The latter are then loaded from the disk and exactly matched for the identical 512-byte lower-level chunks.

Sparse indexing [19] is another well-known similarity-based technique. The input files are partitioned into the variable-length content-defined superchunks as opposed to the previous example [18]. The sliding window technique with a TTTD improvement [20] is used first for finding the lower-level chunk boundaries and then to find the higher-level superchunk boundaries. In the latter case, the window is slided chunk-by-chunk instead of byte-by-byte. The divisors are chosen so that the average size of the superchunks is a few megabytes and the average size of the chunks is a few kilobytes. Every superchunk has a corresponding *manifest* stored on the disk. A manifest lists the sequence of the chunks that comprise the superchunk, storing for each chunk its SHA1 hash value, pointer to its location in the chunk store on the disk, and the chunk length. After an input file is partitioned into chunks and superchunks, every superchunk is treated separately. First, a few representative chunks, called *hooks*, are sampled from the incoming superchunk. The hash values of these hooks are the features of a superchunk. The authors propose to sample the chunks whose hash values have the first n bits zero. Thus, given a random hash function and random data, the average sampling rate is 2^{-n}. Then, a few already stored superchunks most similar to the incoming superchunk are identified. These superchunks are called *champions*. They are identified by querying the in-RAM dictionary data structure called *sparse index* using the features of the incoming superchunk. The keys for searching the sparse index are the features (hash values of hooks), and for every feature a list of the pointers to the superchunk manifests (one hook could be present in more than one superchunk) is returned. However, in practice, if RAM is limited, only one pointer is stored per hook – that of the last stored superchunk. For an incoming superchunk we thus make a constant (with regard to the average number of the chunks in a superchunk and n) number of the sparse-index queries – one for each hook in the incoming superchunk. Each query takes a constant time on average, since the sparse index is implemented as a hash table with a chained hashing. The counters for each reported manifest pointer for every queried feature are maintained, so we can keep track which of the

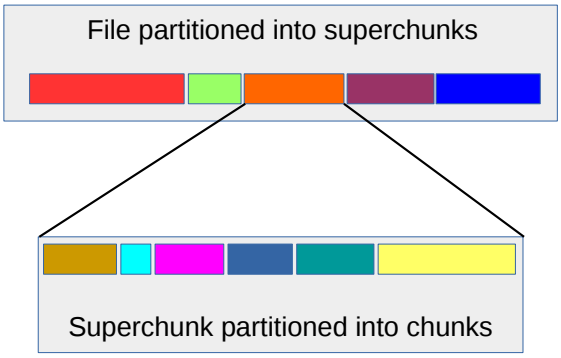

Figure 5. Example of a two-level hierarchical partitioning into the superchunks at the higher level and the chunks at the lower level. In this example, both the superchunks and chunks are of a variable length.

already stored superchunks contain the most hooks of an incoming superchunk and are thus the most similar. Note that a query into the sparse index could return nothing in case of features (hooks) that are new. The number of the champions is limited (the authors suggest a limit of 10) and not always achieved. When the champions are identified, their manifests are loaded into RAM from the disk. At this point, the SHA1 hash values of all the chunks (not just hooks) are compared to the chunk hash values of all the chunks from all the previously identified champions in order to identify the duplicates. The chunks from an incoming superchunk found not to be duplicated are stored to the chunk store on the disk. A manifest is then created for the incoming superchunk with the pointers to the chunks. Lastly, the sparse index is updated. An entry is created for each new hook, and the entries for the existing hooks are updated. If the sparse index stores only one manifest pointer per hook (as is suggested to preserve RAM), then the existing manifest pointer is overwritten with the pointer to the manifest of the incoming superchunk. However, if several manifest pointers are allowed, the pointer to the new manifest is added and, if the maximum is exceeded, the pointer to the oldest manifest is removed.

7 WRITE COMMIT

The final step of deduplication is storing the non-duplicated chunks in a structure often called the *chunk store*, updating the chunk metadata data structures and storing the file recipes.

The details on this step are often vague or even omitted in papers. This is especially true when a technique is designed with a specific storage system (including hardware specifications) in mind, which results in intrinsic optimizations. These techniques are usually designed by R&D departments of storage product companies. For example, when a storage system only uses the RAID-6 arrays with specific stripe sizes, consideration is taken of how often and in what layout new chunks should be written to a persistent storage. We should note, however, that most system software takes the underlying system architecture into account at least to some extent.

Deduplication techniques have been designed that take additional actions in the final step to further reduce the required space. The possible additional actions are *entropy encoding* (sometimes called classical compression) and *delta encoding*. The basic idea of the delta encoding is to encode one data object as a set of differences (deltas) with regard to the other data objects. The main issue with using this for redundancy elimination in large storage systems is how to efficiently find groups of similar objects, so that one of them can be used as a reference object, while the others are encoded as a (as small as possible) set of differences to this reference object. These techniques are sometimes described as **hybrid**. A good example is the technique presented in [21]. The method first applies a variable-length chunk deduplication similar to LBFS [4] (see Section 5). The m chunks not identified as duplicated are further analyzed. Similar to the approaches described in Section 6.2, several features are computed for each chunk using the Rabin hashing. These additional features are then coalesced into a single *superfeature* as suggested in [17] by grouping the existing features (hashes of the chosen chunks) and calculating the hashes over their concatenation – the process is similar to building the Merkle tree [22]. This makes it possible to identify the groups of similar chunks in the $O(m \log m)$ time instead of brute forcing every pair of chunks in the $O(m^2)$ time (see [17] for a detailed treatment). When such groups of similar chunks are identified, a single chunk is chosen randomly in every group to act as a reference, while the other chunks in the group are delta-encoded against the reference chunk. Furthermore, the chunks that are not handled by any of the above procedures are entropy-encoded.

8 READING THE DATA

Deduplication changes how the storage system writes the data. However, reading is also affected, at least by the fact that the file recipes must be checked before a file can be reconstructed and returned. For the file-level deduplication we can imagine a recipe containing a single-chunk hash value. Storing the file recipes efficiently is not trivial, but is beyond the scope of this paper. Let us rather explain an additional issue that also affects how writing is done.

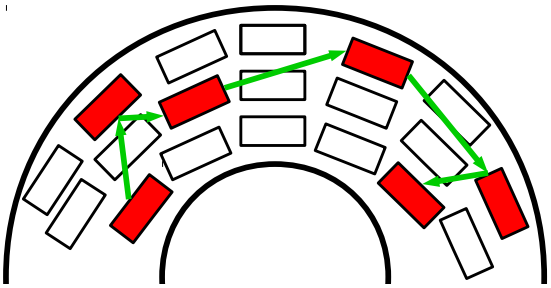

Figure 6. Random access pattern when reading a deduplicated file.

Consider writing a file that is almost completely deduplicated, i.e. most of the file is comprised of chunks already stored in the system. Since these chunks could be stored by different write operations, they can be scattered all over the disk. When this file is later read from the system, what should have been a sequential access turns into a random access due to the scattered chunks as is depicted in Figure 6. The same effect of a non-sequential access is seen in local file systems, only

there it is caused by fragmentation. This non-sequential access is not a problem by itself, but is a consequence of the fact that the system is using the storage devices optimized for a sequential access. The problem is less obvious in storage systems with multiple disks or even multiple machines, where the chunks stored on different disks or nodes can be accessed in parallel. A trivial technological solution is to use the devices with a better random access performance, such as the SSD disks, but this could be costly. The deduplication technique designers try to mitigate this issue by introducing read caches as well as content-aware layouts. We will not discuss this in detail. An example solution can be found in [23] where a deduplication technique for a primary-workload storage system is presented. These are especially sensitive to the read-performance degradation, since the reads prevail over the writes in the primary workloads.

9 Deduplication in Distributed Storage Systems

Due to the cheaper and more powerful commodity hardware as well as affordable high-performance networks, many modern computer systems are distributed with storage systems as no exception. Several solutions for deduplication in the distributed storage systems have been proposed. According to [24], there are two approaches to deduplication in the distributed storage systems. One is *local deduplication with routing*. Basically, deduplication is done locally at the nodes of a distributed system, but to achieve better space savings, routing based on some sort of similarity is used so that the similar data is stored at the same node and thus the chance for deduplication increases. The other approach is *global deduplication* which supports cross-node references. This could lead to better space savings, since the redundancy in the data stored on different nodes could be eliminated, but is quite difficult to implement efficiently. The system described in [24] uses both approaches.

Local deduplication with routing can be further divided:

1) *Stateless* data routing assigns the chunks to the nodes based only on the contents of the chunks. This approach is followed by [25] and enables uniformly distributed data without the need to query into additional data structures.
2) *Statefull* routing assigns the chunks to the nodes based also on a previous assignment of the similar data. This achieves a better deduplication, but causes performance and load-balancing issues. This approach is taken in [26].

In both cases, since routing is based on similarity, the data is first partitioned on a coarser granularity into the superchunks (see Section 6.2). Partitioning is mostly done at the clients or specific nodes in a centralized fashion. The superchunks are then routed to specific nodes that hopefully contain similar superchunks, that is, superchunks containing many identical chunks.

Let us look at the stateless-routing technique extreme binning [25] in more detail. The idea is to store the similarity index of every file in the main memory. This is called the primary (higher level) index. The content-defined chunking [4] is used to partition the incoming files. The feature contains a SHA-1 hash value of the whole file, ID of the representative chunk (the chunk with the minimum hash value as per the MinHashing approach [16]) and a pointer to the bin stored on the disk to which this chunk belongs. Similar files (the ones having the same representative chunk) are grouped into bins stored on the disks. The bins represent the second tier (lower level) index. In this way, the duplicated files are detected already in RAM, while the duplicated chunks are detected inside each bin once it is loaded from the disk. The technique should be classified as a similarity-based deduplication approach that can be used in either a centralized or a distributed setting. In the latter, the K nodes of a distributed system are numbered from 0 to $K-1$. The representative-chunk hash value (which has the role of the chunk ID) is used for stateless routing, with the files being routed to the node $chunkID \mod K$.

10 Conclusion

Deduplication is a useful addition to approaches for tackling an ever increasing amount of data produced and stored by the today's computer systems. It is especially appropriate for the archival and backup, cloud stores, virtual machine storage, enterprise file servers and any other case where data exhibits redundancy that cannot be eliminated by other means, such as entropy encoding which tackles much smaller data sets. If done efficiently using advanced techniques, such as those presented in this paper, space saving far outweighs a potential performance degradation.

Although much has already been done in the area of the unstructured-data deduplication, there is still room for improvement. Some of the challenges are:

- Write the data in such a way that the reads will result in more sequential and less random accesses.
- Other improvements for the use in the primary-workload storage systems where the reads prevail over the writes.
- Better approaches for the distributed-data deduplication.

REFERENCES

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, p. 29, Dec. 2003.

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, p. 205, Oct. 2007.

[3] D. Geer, "Reducing the Storage Burden via Data Deduplication," *Computer*, vol. 41, no. 12, pp. 15–17, Dec. 2008.

[4] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, p. 174, Dec. 2001.

[5] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts, Seventh Edition*. John Wiley & Sons, 2007.

[6] M. Mesnier, G. Ganger, and E. Riedel, "Storage area networking - Object-based storage," *IEEE Communications Magazine*, vol. 41, no. 8, pp. 84–90, Aug. 2003.

[7] S. A. B. Sage A. Weil, "Ceph: A scalable, high-performance distributed file system," *Proceedings of the 7th symposium on Operating systems design and implementation - OSDI '06*, p. 14, 2006.

[8] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur, "Single instance storage in Windows® 2000," in *WSS'00 Proceedings of the 4th conference on USENIX Windows Systems Symposium*. USENIX Association, Aug. 2000, p. 2.

[9] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *ACM Transactions on Storage*, vol. 7, no. 4, pp. 1–20, Jan. 2012.

[10] U. Manber, "Finding similar files in a large file system," in *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference (WTEC'94)*. USENIX Association, Jan. 1994, p. 2.

[11] K. Eshghi and T. Khuern, "A Framework for Analyzing and Improving Content-Based Chunking Algorithms," Hewlett-Packard Laboratories, Tech. Rep., 2005.

[12] M. O. Rabin, "Fingerprinting by Random Polynomials," Center for Research in Computing Technology , Harvard University, Tech. Rep., 1981.

[13] D. R. Stinson, *Cryptography (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, 2005.

[14] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[15] 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 (FAST'08)*. USENIX Association, Feb. 2008, p. 18.

[16] A. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*. IEEE Comput. Soc, 1997, pp. 21–29.

[17] A. Z. Broder, "Identifying and filtering near-duplicate documents," in *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, ser. COM '00. London, UK, UK: Springer-Verlag, 2000, pp. 1–10.

[18] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein, "The design of a similarity based deduplication system," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference on - SYSTOR '09*. New York, New York, USA: ACM Press, May 2009, p. 1.

[19] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, "Sparse indexing: large scale, inline deduplication using sampling and locality," in *Proccedings of the 7th conference on File and storage technologies (FAST'09)*. USENIX Association, Feb. 2009, pp. 111–123.

[20] T.-S. Moh and B. Chang, "A running time improvement for the two thresholds two divisors algorithm," in *Proceedings of the 48th Annual Southeast Regional Conference on - ACM SE '10*. New York, New York, USA: ACM Press, Apr. 2010, p. 1.

[21] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey, "Redundancy elimination within large collections of files," in *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC'04)*. USENIX Association, Jun. 2004, p. 5.

[22] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," pp. 369–378, Aug. 1987.

[23] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti, "iDedup: latency-aware, inline data deduplication for primary storage," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, Feb. 2012, p. 14.

[24] P. Efstathopoulos, "File routing middleware for cloud deduplication," in *Proceedings of the 2nd International Workshop on Cloud Computing Platforms - CloudCP '12*. New York, New York, USA: ACM Press, Apr. 2012, pp. 1–6.

[25] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge, "Extreme Binning: Scalable, parallel deduplication for chunk-based file backup," in *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*. IEEE, Sep. 2009, pp. 1–9.

[26] D. Frey, A.-M. Kermarrec, and K. Kloudas, "Probabilistic deduplication for cluster-based storage systems," in *Proceedings of the Third ACM Symposium on Cloud Computing - SoCC '12*. New York, New York, USA: ACM Press, Oct. 2012, pp. 1–14.

Andrej Tolič received his B.Sc. degree in Computer Science and Mathematics from the University of Ljubljana, Slovenia. He is currently a Ph.D. student at the same university working on deduplication in distributed storage systems. His research interests include systems software, distributed and networking systems, cybersecurity, and cryptography.

Andrej Brodnik received his Ph.D. degree from the University of Waterloo, Ontario, Canada, in 1995. After graduation he worked as a head of research and CTO in industry (IskraSistemi and ActiveTools). In 2002 he joined the University of Primorska and also worked as a researcher and adjoined professor at the University of Technology in Luleå, Sweden. He has authored several tens of various scientific papers and is an author and co-author of patents in Sweden and the USA. The CiteSeer and ACM Digital Library list over 200 citations of his works. He is also a recipient of a number of awards (National award for exceptional achievements in higher-education teaching; Boris Kidrič award; Fulbright scholarship; IBM Faculty Award, 2004; Golden Plaque of the University of Primorska; etc.). Currently he holds a teaching position at the University of Primorska and the University of Ljubljana.