SAP: Similarity-Aware Partitioning for Efficient Cloud Storage

Bharath Balasubramanian[1], Tian Lan[2], and Mung Chiang[1]
[1]Princeton University, NJ, USA, [2]George Washington University, DC, USA
{bharathb, chiangm}@princeton.edu, tlan@gwu.edu

***Abstract*—Given a set of files that show a certain degree of similarity, we consider a novel problem of deduplicating them (eliminating redundant chunks) across a set of distributed servers in a manner that is: (i) space-efficient: the total space needed to deduplicate and store the files is minimized and, (ii) access-efficient: each file can be accessed by communicating with a bounded number of servers, thereby minimizing network-access times in congested data center networks. A space-optimal solution in which we first deduplicate all the files and then distribute them across the servers (referred to as chunk-distribution), may require communication with many servers to access each file. On the other hand, an access-efficient solution in which we randomly partition the files cross the servers, and then store their unique chunks on each server may not exploit the similarities across files to reduce the space overhead. In this paper, we first show that finding an access-efficient, space optimal solution is an NP-Hard problem. Following this, we present the similarity-aware-partitioning (SAP) algorithms that find access-efficient solutions within polynomial time complexity and guarantees bounded space overhead for arbitrary files. Our experimental verification on files from Dropbox and CNN confirm that the SAP technique is much more space-efficient than random partitioning, while maintaining compression ratio close to the chunk-distribution solution.**

I. Introduction

Data volumes are increasing at an unprecedented rate due to the advent of social networks such as Facebook, on-demand cloud-based video services like Netflix, and file sharing applications such as Dropbox. These cloud-based services must satisfy two important concerns (among many others): (i) They should be *access-efficient* in terms of minimizing network accesses to read or write to the files in the system. This is a crucial factor with increasing network congestion in the underlying data center networks [14], [4]. (ii) They should be *space-efficient* to manage the high volumes of data [11]. Given a set of files that show a certain degree of similarity or redundancy, we consider the problem of storing them across a set of distributed servers and present solutions that are both access-efficient and space-efficient.

Space efficiency is often achieved by the process of *deduplication* [19], [6], [17], [2], [9], which splits all the files in the system into chunks and maintains only a unique copy of each chunk. Deduplication operates on server file systems and typically achieves 30% - 50% compression in various applications. In Fig. 1(i), we illustrate a simplified version of deduplication. To store the files $A_1, \ldots, A_5$, without deduplication, we require ten chunks (assuming chunk size of 3 bits). However, if we eliminate duplicate chunks, we need to store only six chunks, $B_1 \ldots B_6$. Typically, most deduplication solutions have focused on reducing the space-overhead *within* a single server [7], [21], [26]. These techniques do not consider the problem of distributed deduplication due the cost of network accesses and file maintenance in a distributed solution. However, files with a high degree of similarity may be deduplicated on different servers. In this paper, we show that it is possible to achieve both space and access-efficiency in a distributed solution.

Consider the example in Fig. 1. Given a set of distributed servers, each of capacity 9 bits, how can we store the chunks of the files in $A_1 \ldots A_5$ across the servers? We illustrate three different approaches to this problem. In the first approach, as shown in Fig. 1(ii), we first deduplicate the files in a central location and distribute the unique file chunks $B_1, \ldots, B_6$ sequentially across each server. A chunk-location mapping can be stored at any of the servers to allow file access. This particular deduplication technique is space-optimal and requires only two servers. However, this approach suffers from several drawbacks. First, to access file A_3, we need to retrieve its chunks from two servers across the network. When there are many servers, in the worst case, we may need to retrieve chunks from a prohibitively large number of servers across the network. With network congestion being a significant concern, this is not an access-efficient solution in terms of either network-access times or message overhead. Second, there is considerable engineering overhead associated with reconstructing files from chunks obtained from multiple servers and is hence rarely used, especially when the data needs to be accessed frequently. Finally, since all the files need to be deduplicated first, the centralized server performing deduplication needs to have a large amount of physical memory to enable efficient deduplication.

In Fig. 1(iii), we randomly partition the set of files across the servers and then deduplicate them to maintain unique chunks. Due to the server space constraints, while A_1 and A_2 "fit" on the first server, only A_3 with three chunks each of three bits can reside on the second server. This forces the need for a third server for A_4 and A_5. This solution is access-efficient, since to access any file we need to communicate with only one server, but it is not as space-efficient as the

978-1-4799-3360-0/14/$31.00

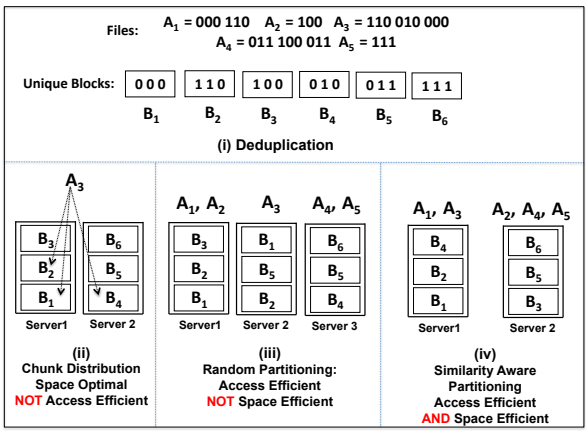


Fig. 1. Space and Access-Efficient SAP

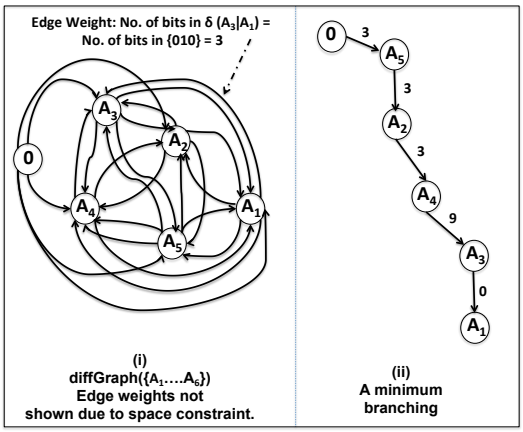


Fig. 2. Delta-Encoding for File Similarity

first solution.

We present a *similarity-aware-partitioning* (SAP) approach that is both *access-* and *space-efficient* in Fig. 1(iv). The key insight, in this illustrative example, was to identify that A_1 and A_3 have similar chunks, and so do A_2 and A_4. Hence by partitioning the files as $\{A_1, A_3\}$ and $\{A_2, A_4, A_5\}$, and then deduplicating them on each server, we require only two servers. Clearly, the chunks of any one file are contained locally on each server. Unlike the solution in Fig. 1(ii), the load of performing deduplication is shared by the servers (this is true even for the random partitioning solution). In Fig. 1, the SAP approach and chunk-distribution both require the same number of servers. In general, this may not be the case and we may have additional redundancy in the SAP approach. However, in the process, we reduce the network-overhead for file access.

Our main goal is to minimize the space required by the access-efficient SAP solution. For this small example, we are able to determine the optimal partition simply by inspection. For a general set of files it is a very computationally expensive task to compute all possible partitions, and then identify the ones which capture file similarity better. In this paper, we present polynomial time algorithms to partition the given set of files and store them across the servers in a space and access-efficient manner.

In the following paragraphs, we summarize the contributions in this paper.

Definition of (k, C)-distribution: First, we need to define the distributed storage problem in a way that captures the notion of access-efficiency. Consider a set of k servers each of capacity C and a set of files $\mathcal{A} = \{A_1, \ldots, A_n\}$, where the size of any file is less than C. Informally, a (k, C)-distribution of $\mathcal{A}$ is a set of k elements $\mathcal{D} = \{D_1, \ldots, D_k\}$, such that: (i) D_i contains chunks belonging to the files in $\mathcal{A}$, (ii) the chunks of any $A_i \in \mathcal{A}$ is contained in exactly one of the D_is in $\mathcal{D}$ and, (iii) the size needed to represent any D_i is less than or equal to C. The solution in Fig. 1(iv) is $(2, 9)$-distribution of $\mathcal{A} = \{A_1 \ldots A_5\}$ with $D_1 = \{B_1, B_2, B_4\}$ and $D_2 = \{B_3, B_5, B_6\}$. Similarly, the solution in 1(iii) is a $(3, 9)$-distribution of $\mathcal{A}$. However, the solution in figure 1(ii) is *not* a (k, C)-distribution of $\mathcal{A}$ since the chunks belonging to A_3 are across servers 1 and 2.

We define access-efficiency based on communication with *exactly* one server, mainly to understand the performance trade-offs of a solution that completely minimizes network accesses. In future work, we will extend this to the case where we consider communication with multiple servers. This will enable us to quantify the trade-off between the speed due to the concurrent retrieval of files against the overhead of network accesses. In the technical report [3], we show that the problem of generating a space-optimal (k, C)-distribution for a given set of files is NP-Hard. We address this problem, using the delta-encoding function, to present polynomial time algorithms that generate space-efficient solutions.

Delta-Encoding Approximation for Deduplication: Delta-encoding is commonly used to find the differences among two files [8], [5], [15]. There are many popular tools such as the UNIX *diff*, *vcdiff* and *gdiff* to perform this function. In [20], the authors present a technique to compress a set of files $\mathcal{A}$ using just the delta-encoding function between each pair of files. Their key intuition is to construct a directed graph, which we denote *dGraph*$(\mathcal{A})$, in which each node corresponds to one of the files in $\mathcal{A}$ and the weight of an edge $A_i \rightarrow A_j$ is the size of A_j delta-encoded over A_i, i.e., the amount of information needed to represent A_j given A_i. They show that the minimum directed spanning tree (branching) of such a graph represents the space-optimal way of storing a set of files, given just the delta-encoding function.

In this paper, we present an analytic framework to compare: (i) total size of files in $\mathcal{A}$ compressed using deduplication, denoted $S_\Delta(\mathcal{A})$ and, (ii) total size of files compressed using delta-encoding, denoted $S_\delta(\mathcal{A})$. We show the surprising result that this compression ratio is bounded tightly as $\frac{S_\delta(\mathcal{A})}{S_\Delta(\mathcal{A})} \leq \Gamma(\rho)$, where ρ is the *maximum chunk popularity*, i.e., the maximum number of unique files in $\mathcal{A}$ any chunk appears in, and $\Gamma(\rho)$ is a quadratic function whose value

TABLE I
NOTATION USED IN THE PAPER

$\mathcal{A}$	Given set of files	n	Number of files
k	Number of servers	C	Size or capacity of each server
Δ	Deduplication function	δ	Delta-Encoding function
P	Bit-size function	*dGraph*	Delta-encoding based graph
s	Maximum file size	ρ	Maximum chunk popularity

is bounded by ρ. Since for practical data sets, ρ is a small constant, the compression-ratio $\frac{S_\delta(\mathcal{A})}{S_\Delta(\mathcal{A})}$ is bounded, even for arbitrary files. Based on this intuition, we present efficient algorithms to generate the (k, C)-distribution for a set of files.

SAP Algorithms for (k, C)-distribution: We present the efficient, bounded, similarity-aware-partitioning (SAP) algorithms to generate a (k, C)-distribution for a given set of files $\mathcal{A}$. (i) For a fixed number of servers k, we balance storage-load among the servers. Let the space required by the optimal deduplication-based (k, C)-distribution be S_{opt}. Let the maximum file size be denoted by s. The SAP algorithm has time complexity $O(n^2 s + k)$ and generates a (k, C)-distribution that requires space at most $\Gamma(\rho)S_{opt} + (k-1)s$. (ii) For fixed server capacity C, we minimize the number of servers needed. The SAP algorithm has time complexity $O(\frac{n^2 s \log(C) \log(n)}{C})$ and generates a (k, C)-distribution that requires at most $1 + \frac{\Gamma(\rho)N_{opt}}{\left[1-\log_2(\frac{C}{s})/(\frac{C}{s})\right]}$ servers, where N_{opt} is the servers required by the optimal deduplication-based (k, C)-distribution.

Implementation and Experiments: We performed basic experiments on user files chosen from *Dropbox* and random web files downloaded from *CNN*. For the Dropbox files, on average, our SAP technique achieves 22% compression (over the uncompressed set of files), while random partitioning (based on Fig. 1(iii)) and chunk-distribution (deduplication based on Fig. 1(ii)) achieve 15.0% and 23.3% compression respectively. For the CNN files, our technique on average achieves 28% compression while random partitioning and chunk-distribution achieve 15% and 30.1% compression, respectively. These results confirm that the SAP algorithms are much more space-efficient than random partitioning, while maintaining compression ratio close to the access-inefficient chunk-distribution solution.

In the following section, we describe the system model and assumptions, followed by the problem definition. In section III, we describe the delta-encoding based technique for capturing file similarities. In section IV, we present the SAP algorithms. The results of our experiments are presented in section V. Finally, we describe the related work and end with the conclusions and future work.

II. System Model and Definitions

In this section, we first describe the assumptions on our servers and the nature of file accesses. Following this, we describe the two operations that we use to measure the similarity among these files: deduplication and delta-encoding. Finally, we present the formal definition of a (k, C)-distribution for a given set of files. The notation used in this paper is summarized in Table I.

We are given files $\mathcal{A} = \{A_1, \ldots, A_n\}$ to store on a set of distributed servers with no shared state or memory. Each A_i can be broken down into fixed-size chunks. The servers in our system maintain a subset of these fixed size chunks. We consider a fault-free system in this paper. Dealing with faults in the storage servers is an important avenue of future work. Accesses to the files, is performed by an external client which knows both the location and order of the chunks for each file. When a server receives a read request for a file A_i, it replies to the client with all the information it has corresponding to A_i. Writes are more complicated since it may change the nature of chunk references and requires consistency management over distributed servers. While propagating updates in deduplicated or delta encoded systems is a well studied topic [2], [19], we focus on an offline algorithm for the partitioning and storage of the files. Addition and deletion of files is part of our future work.

Deduplication: Given a set of files $\mathcal{A}$, *deduplication* breaks each file $A_i \in \mathcal{A}$ into fixed-size q-bit chunks (with padding if necessary), and removes all redundant chunks through similarity check. Let $\mathcal{F}$ be a sample space, that is the set of all possible file chunks of q-bits. Each file A_i can be represented as an ordered tuple of q-bit chunks belonging to $\mathcal{F}$. For a set of files $\mathcal{A}$, and a value of q, the deduplicated representation of $\mathcal{A}$, denoted by $\Delta(\mathcal{A}) \in \mathcal{F}$, is the set of unique file chunks among all the files in $\mathcal{A}$. In Fig. 1(i), $q = 3$, $\Delta(\mathcal{A}) = \{000, 110, 100, 010, 011, 111\}$ while $\Delta(\{A_4\}) = \{011, 100\}$. The amount of space needed to represent any subset $F \subseteq \mathcal{F}$ is called the *bit-size* of F and is denoted by $P(F)$. For example, $P(\Delta(\mathcal{A})) = 18$.

Delta-Encoding: Given any two files A_i and A_j, the *delta-encoding* of A_j over A_i, denoted $\delta(A_j|A_i)$, is defined as, $\delta(A_j|A_i) = \Delta(\{A_j\}) - \Delta(\{A_i\})$ (in terms of set subtraction). It represents the amount of information needed to represent A_j given that A_i is available. We refer to A_i as the *base file* and A_j as the *target file*. In Fig. 1(i), $\delta(A_3|A_1)$ ={110,010,000} - $\{000, 110\}$ = $\{010\}$. Also, the bit-size of $\delta(A_3|A_1)$, denoted by $P(\delta(A_3|A_1)) = 3$. Note the following assumptions, inherent in our definitions of deduplication and delta-encoding:

- Deduplication in practical systems is much more complex to address concerns such as uneven file lengths and

Rabin-fingerprinting for shift resistant chunks [19]. Similarly, delta-encoding can be performed based on longest common string searches with different techniques for different file formats [23], [15]. Our simplified definitions conceptually represent both these operations and provide a common framework to compare their performance. For ease of comparison, we assume that the chunk size chosen for all the functions in the paper are identical.

- Given $\Delta(\mathcal{A})$, to regenerate any file $A_i \in \mathcal{A}$, we need the chunks in $\Delta(\mathcal{A})$ as well as meta information that specifies which chunks belong to A_i, as well as their exact order in A_i. We assume that the chunk size q relative to the file size is large enough so that the size occupied by the meta information is negligible. Therefore, $\Delta(\mathcal{A})$ represents a lower bound of the minimum information needed to construct all files in $\mathcal{A}$ using practical deduplication algorithms.
- We do not consider chains of delta-encoding, e.g., $\delta(A_i|A_j|A_l)$. Given a set of files, we assume that the delta-encoding function can eliminate identical chunks only among pairs of files, while the deduplication function can identify identical chunks across all the files in the system. Even without assuming chains of delta-encoding, we show that our solution is space-efficient.

We now present the definition of the main problem addressed in this paper. The main idea behind this definition is to describe the distributed storage problem in a way that captures access-efficiency.

Definition 1 ((k, C)-distribution): Given a set of files $\mathcal{A} = \{A_1, \ldots, A_n\}$ and a set of k servers each of capacity C, a (k, C)-distribution of $\mathcal{A}$ is a set $\mathcal{D} = \{D_1, \ldots, D_k\}$, such that:

- Each D_j is a multi-set of $\Delta(\mathcal{A})$ such that $P(D_j) \leq C$.
- The unique chunks of each file A_i is a subset of some $D_j \in \mathcal{D}$, i.e., $\Delta(A_i) \subseteq D_j$. In other words, each file can be locally regenerated from one of the elements in $\mathcal{D}$.

A (k, C)-distribution $\mathcal{D}$ of $\mathcal{A}$ is access-efficient by definition, since we can simply place each $D_i \in \mathcal{D}$ on a different server. This ensures that to regenerate any file in $\mathcal{A}$, we need information from only one server, thereby minimizing: (i) network access times, (ii) number of network messages, and (iii) consistency management across the distributed servers. The solution in Fig.1 (iv) with $D_1 = \Delta(\{A_1, A_3\})$ and $D_2 = \Delta(\{A_2, A_4, A_5\})$ is a $(2, 9)$-distribution of $\mathcal{A} = \{A_1, \ldots, A_5\}$. On the other hand, the solution in Fig.1(ii) is not a (k, C)-distribution since the chunks of A_3 are spread across the two servers.

The main goal of this paper is to generate a space-efficient (k, C)-distribution for the given set of files in $\mathcal{A}$. This is a hard problem, mainly due to the exponentially many partitions of $\mathcal{A}$ that we need to consider. In the following section, we present the approach of using the delta-encoding function to capture file similarities. This forms the basis of our polynomial time, bounded heuristics in section IV. Due to space constraints, we maintain all the proofs in our technical report [3].

III. File Similarities using Delta-Encoding

Given a set of n files, $\mathcal{A} = \{A_1 \ldots A_n\}$, in [20], the authors present a technique to compress a set of files using just the delta-encoding function between all pairs of files in $\mathcal{A}$. While using a similar technique, we present a theoretical framework to compare: (i) the space occupied by a set of files compressed using just the delta-encoding function, denoted by $S_\delta(\mathcal{A})$ (ii) the space occupied by a deduplication-based solution, denoted by $S_\Delta(\mathcal{A})$. Our analysis shows the surprising result that $\frac{S_\delta}{S_\Delta}$ is bounded, even for arbitrary files. It provides the key result that we leverage to quantify the performance of our SAP algorithms.

We define *dGraph*($\mathcal{A}$) as a directed weighted graph with n nodes, each corresponding to one of the files in $\mathcal{A}$. The weight of the directed edge $(A_i \rightarrow A_j)$, $j \neq 0$, denoted by $w(i, j)$, is calculated as $P(\delta(A_j|A_i))$. The edge-weight captures file-similarity using delta-encoding: if the weight of any edge is small, then the files are similar. Futher, we add an additional node labeled A_ϕ (the null node) and edges $(A_\phi \rightarrow A_i)$ with weights $P(\Delta(\{A_i\}))$ to allow the delta-encoding of single files. In Fig. 2(i), we show *dGraph*($\mathcal{A}$) for $\mathcal{A} = \{A_1 \ldots A_5\}$ in Fig. 1(i). Note that $w(1, 3) = 3$ since $\delta(A_3|A_1) = \{010\}$ (very similar), whereas $w(1, 4) = 6$ since $\delta(A_4|A_1) = \{011, 100\}$ (different).

Given a graph $G = (V, E)$, a directed spanning tree (DST) or branching of G rooted at r, is a subgraph T of G such that the undirected version of T is a tree and T contains a directed path from r to any other vertex in V . The cost $w(T)$ of a directed spanning tree T is the sum of the costs of its edges. A *minimum branching* rooted at r is a directed spanning tree rooted at r of minimum cost.

In the graph *dGraph*($\mathcal{A}$), the minimum branching represents a tree contained in the graph, in which very similar files are parents (or children) of each other. Consider the minimum branching shown in Fig. 2(ii). We denoted the null node A_ϕ as '0'. The most space-efficient way of maintaining the files in $\mathcal{A}$, using just the pair-wise delta-encoding function, is to maintain the files: A_ϕ, $\delta(A_5|A_\phi)$, $\delta(A_2|A_5)$, $\delta(A_4|A_2)$, $\delta(A_3|A_4)$ and $\delta(A_1|A_3)$. Given these files, we can regenerate all the files in $\mathcal{A}$. For example, assume that we need to retrieve A_4. Since $\delta(A_5|A_\phi) = \Delta(A_5)$, we can first obtain A_5; then A_2 using A_5 and $\delta(A_2|A_5)$ and then A_4 using A_2 and $\delta(A_4|A_2)$. We state this formally in the following observation.

Observation 1: Given a set of files, $\mathcal{A} = \{A_1, \ldots, A_n\}$, consider a minimum weight branching of *dGraph*($\mathcal{A}$) defined by the n edges: $\{A_{i_1} \rightarrow A_{j_1}, A_{i_2} \rightarrow A_{j_2}, \ldots A_{i_n} \rightarrow A_{j_n}\}$, for $A_{i_1}, \ldots, A_{i_n}, A_{j_1}, \ldots, A_{j_n} \in \mathcal{A} \cup \{A_\phi\}$. The most space-optimal way of maintaining the files in $\mathcal{A}$, using just the pair-

wise delta-encoding function, is to maintain $\Delta(\{A_{i1}\})$ (the root of the branching) followed by the chain of delta-encoded files, $\delta(A_{j_t}|A_{i_t})$ for all $t = 1, \ldots, n$. Here, A_{i_1} is called the *reference file*.

A. Delta-Encoding: An approximation for Deduplication

While the delta-encoding function can at best capture pair-wise similarity among the files in $\mathcal{A}$, deduplication can identify and remove similar chunks across all the files in $\mathcal{A}$. Hence, one would expect the deduplication function to achieve much more compression across a set of files as compared to the delta-encoding function. However, in Theorem 1, which is the main result of this section, we derive a uniformly-tight upper bound of the compression ratio and then show that the bound does not increase on the order of the number of files, n. To this end, we first provide an upper bound of $S_\delta(\mathcal{A})$ as follows.

Lemma 1: Consider a set of n arbitrary files $\mathcal{A} = \{A_1 \ldots A_n\}$. Let the size of the optimal delta-encoding based solution to maintain the set of files in $\mathcal{A}$ be denoted $S_\delta(\mathcal{A})$. We have

$$S_\delta(\mathcal{A}) \leq \sum_{i=1}^{n} P(\Delta(\{A_i\})) - \frac{2}{n} \sum_{i<j} P(\Delta(A_i) \cap \Delta(A_j)) \quad (1)$$

where $A_i \cap A_j$ denotes the set of common chunks between files A_i and A_j.

We prove Lemma 1, in the technical report [3], by replacing the minimum weight branching (which achieves $S_\delta(\mathcal{A})$) with an average over all possible branching of *dGraph*($\mathcal{A}$). The result, extending Bonferroni-type inequalities [10] in probability theory, validates our intuition that the optimal delta-encoding based solution can effectively reduce pair-wise duplications $P(\Delta(A_i) \cap \Delta(A_j))$.

Following this, we derive an asymptotically tight bound for $\frac{S_\delta(\mathcal{A})}{S_\Delta(\mathcal{A})}$ using a new technique which we refer to as *chunk popularity analysis*. We define the *popularity* of a chunk belonging to the files in $\mathcal{A}$, to be the number of unique files of $\mathcal{A}$ it appears in. For example, in Fig. 1, chunk B_1 has a popularity of two, since it appears in files A_1 and A_3. To state our bound in a general form, we define a chunk frequency function $f_\mathcal{A}(p)$, which is the fraction of chunks in $\mathcal{A}$ that have a popularity of p. An example of $f_\mathcal{A}(p)$ for randomly selected CNN and Dropbox files is discussed in Fig. 5 of Section V.

Theorem 1: Let $S_\Delta(\mathcal{A})$ be the size of the optimal deduplication-based solution to store the files in $\mathcal{A}$. For any n arbitrary files $\mathcal{A} = \{A_1 \ldots A_n\}$, we can bound the compression ratio of the delta-encoding based solution by,

$$1 \leq \frac{S_\delta(\mathcal{A})}{S_\Delta(\mathcal{A})} \leq \sum_{p=1}^{n} f_\mathcal{A}(p) \left[p - \frac{p^2 - p}{n} \right] \quad (2)$$

The upper bound is uniformly tight since there exist a set of files whose compression ratio achieves exactly the bounds.

In the proof, which is provided in the technical report [3], we separately analyze the contributions of chunks with different popularities and aggregate them to derive the bound in (2). It is also shown that the upper bound is tight in a symmetric *worst case* where every subset of p files share exactly one common chunk, a condition rarely possible for any realistic dataset. When chunk popularities are bounded by a constant, which we denote ρ, it is easy to show that the upper bound in Theorem 1 reduces to a constant.

Corollary 1: If $f_\mathcal{A}(p) = 0$ for all $p \geq \rho$, then we can uniformly bound the compression ratio of the delta-coding based solution by,

$$\frac{S_\delta(\mathcal{A})}{S_\Delta(\mathcal{A})} \leq \rho - \frac{\rho^2 - \rho}{n} \triangleq \Gamma(\rho). \quad (3)$$

Corollary 1 provides a constant upper bound of compression ratio for any files that have maximum chunk popularity ρ. For most practical data sets ρ is a constant, as illustrated by our graph in Fig. 5. In fact, for the data sets from CNN and Dropbox, ρ is as small as 8. Henceforth, in this paper, we assume that the maximum chunk popularity is a constant and use the bound, $\frac{S_\delta(\mathcal{A})}{S_\Delta(\mathcal{A})} \leq \Gamma(\rho)$. It is easy to see that even in the worst case, $\Gamma(\rho)$ outperforms an oblivious solution that stores and compress each of the n files independently. The bound achieved in this section provides the theoretical foundation for analyzing the performance of our SAP algorithms.

IV. SAP Algorithms for (k, C)-distribution

In this section, we present polynomial time algorithms (Fig. 3) to generate a (k, C)-distribution for a given set of files and show that our algorithms have a bounded optimality gap. In the technical report [3], we prove that this problem is NP-Hard.

A. Minimize Total Space Occupied

We first describe the `SAP-Space` algorithms for generating a (k, C)-distribution $\mathcal{D}$ of $\mathcal{A}$ for some $C > 0$, given a fixed number of servers k. In this algorithm, we generate a minimum branching for the given set of files, and then partition the branching into k sub-trees by removing any $k-1$ edges from this branching. Based on how we represent the files of the sub-trees in step 2 (or step 2') we consider two algorithms: (i) `SAP-Space-Delta` which maintains the files in each server using one reference file and a sequence of delta-encoded files and, (ii) `SAP-Space-Dedup` that deduplicates all the files belonging to that sub-tree. While the latter algorithm can be computationally more expensive, it can also be more space efficient since we use deduplication. Since the minimum branching keeps "near-similar" files in consecutive positions on the tree, the files in each of the servers exhibit good similarity.

Algorithm `SAP-Space`
Input: Set of files $\mathcal{A} = \{A_1, A_2, \ldots A_n\}$,
no. of servers k;
Output: (k, C)-distribution of $\mathcal{A}$ for some $C > 0$;
//Step 1: Identify "similar" files
Construct *dGraph*($\mathcal{A}$) ;
Find a minimum branching T_{opt} of *dGraph*($\mathcal{A}$) ;
Set of trees, $\mathcal{T} \leftarrow \phi$;
$\mathcal{D} \leftarrow \phi$;
$\mathcal{T} \leftarrow$ Remove $k-1$ edges randomly from T_{opt} to create k trees;

// Step 2: Store files in delta-encoded form
// `SAP-Space-Delta`
for each tree M **in** $\mathcal{T}$ **do**
Let the tree M be $A_{t1} \rightarrow A_{t2} \ldots \rightarrow A_{t|M|}$;
//root as reference and chain of deltas
$\mathcal{D} \leftarrow \mathcal{D} \cup \{\Delta(A_{t1}), \cup_{i=2}^{i=|M|} \delta(A_{ti}|parent(A_{ti}))\}$;

// Alternate Step 2': Store files in dedup form
// `SAP-Space-Dedup`
for each tree M **in** $\mathcal{T}$ **do**
Let the tree M be $A_{t1} \rightarrow A_{t2} \ldots \rightarrow A_{t|M|}$;
$\mathcal{D} \leftarrow \mathcal{D} \cup \{\Delta(\{A_{t2}, A_{t2}, \ldots, A_{t|M|}\})\}$;

return $\mathcal{D}$;(k, C)-distribution of $\mathcal{A}$;

Algorithm `SAP-Servers`
Input: Set of files $\mathcal{A} = \{A_1, A_2, \ldots A_n\}$,
server capacity C;
Output: (k, C)-distribution of $\mathcal{A}$ minimizing k;
//Step 1: Identify "similar" files
Construct *dGraph*($\mathcal{A}$) ;
Find a minimum branching T_{opt} of *dGraph*($\mathcal{A}$) ;
Set of trees, $\mathcal{T} \leftarrow \phi$;
$\mathcal{D} \leftarrow \phi$;
while (T_{opt} is not empty) **do**
Tree T_c=`size-partition`(T_{opt}, C);
Add T_c to $\mathcal{T}$;
Remove T_c from T_{opt}: $T_{opt} \leftarrow T_{opt} - T_c$;
/* step 2 and 2' are identical to `SAP-Space` */
return $\mathcal{D}$; //(k, C)-distribution of $\mathcal{A}$;

Subroutine `size-partition`
Input: Tree T, remaining deficit d;
Output: Largest tree T_c of size $\leq d$;
// Step 0: Terminating conditions
if (T is a single node)
return $T_c = T$ if $P(T) \leq d$ or $T_c = \{\cdot\}$ otherwise;
// Step 1: Find a candidate sub-tree
Find a sub-tree T_r (root r) in T such that:
$P(T_r) \geq d - P(r)$ and for each child tree L_m with root r_m, $P(L_m) < d - P(r_m)$;
Deficit $d \leftarrow d - P(r)$;//amount to be removed
// Step 2: Select child trees from T_r
Sort L_ms in descending order of $P(L_m)$: $[L_1, L_2, \ldots]$;
Initialize $m = 0$;
while ($\sum_{j=1}^{m} P(L_m) < d$) do
$m = m + 1$;
$\hat{L}_m$=`size-partition`$(L_m, d - \sum_{j=1}^{m-1} P(L_j))$;
//Step 3: Add r as root to all child trees and return
return $T_c = (r \rightarrow \hat{L}_m) \cup (r \rightarrow L_1) \ldots \cup (r \rightarrow L_{m-1})$;

Fig. 3. SAP Algorithm for (k, C)-distribution

For example, consider the files in Fig. 1 with $k = 2$. To divide these files among two servers, we construct their minimum branching (Fig. 2) and remove the edge $A_4 \rightarrow A_3$ to create two sub-trees. For `SAP-Space-Delta`, we generate the following $(2, C)$-distribution $\mathcal{D} = \{D_1, D_2\}$ (for some $C > 0$):

$$D_1 = \{\Delta(\{A_5\}), \delta(A_2|A_5), \delta(A_4|A_2)\}$$
$$D_2 = \{\Delta(\{A_3\}), \delta(A_1|A_3)\}$$

For `SAP-Space-Dedup` we generate:

$$D_1 = \{\Delta(\{A_5, A_2, A_4\}), D_2 = \{\Delta(\{A_3, A_1\})$$

The files in D_1 and D_2 are maintained on two independent servers. Note that, in either of these solutions, all files can be retrieved locally at each server, thereby ensuring access-efficiency. In the following theorem, we present bounds on the space-efficiency of the solution generated by our algorithms.

Theorem 2: Consider a set of files $\mathcal{A} = \{A_1 \ldots A_n\}$, whose sizes are bounded by s. Let the amount of space required by the optimal deduplication-based (k, C)-distribution be S_{opt}. Let the space required by the (k, C)-distribution generated by the `SAP-Servers-Delta` and `SAP-Servers-Dedup` algorithms be $S_{\texttt{SAP-Delta}}$ and $S_{\texttt{SAP-Dedup}}$ respectively. For arbitrary files, we have,

- $S_{\texttt{SAP-Dedup}} \leq S_{\texttt{SAP-Delta}} \leq \Gamma(\rho) S_{opt} + (k-1)s$
- Both `SAP-Space` algorithms have time complexity $O(n^2 s + k)$.

Remark To partition our minimum branching we can use other algorithms in the literature to achieve different trade-offs [13], [16]. For example, we can use the technique in [25] that bounds the ratio of edges among any two sub-trees to at most three. This can be used for better load-balancing among the servers.

B. Minimize Number of Servers

In this section, we describe the `SAP-Servers` algorithm that generates a (k, C)-distribution of $\mathcal{A}$ for a fixed server capacity C. The only difference between this algorithm and

the SAP-Space algorithms is in the way we partition the tree to satisfy server capacity constraints. We use the space-partition routine to split the minimum branching of the files into k subtrees, each of which has size as close to C as possible. Similar to step 2 (or 2') in SAP-Space, we have algorithms SAP-Servers-Delta and SAP-Servers-Dedup based on how the partitioned subtrees are stored.

We now focus on the space-partition routine. The bit-size of a tree T with edges, $\{A_{i_1} \rightarrow A_{j_1}, A_{i_2} \rightarrow A_{j_2}, \ldots\}$ is defined as, $P(T) = P(\delta(A_{j_1}|A_{i_1})) + P(\delta(A_{j_2}|A_{i_2})) + \ldots$. The goal of the partitioning routine is to create a subtree T_c as large as possible while ensuring that the size of the tree, i.e., $P(T_c)$ plus the cost of maintaining its root as a reference file, is no more than the remaining deficit d (i.e., remaining server capacity). In other words, T_c along with its reference file can be maintained on a single server. In Step 1, we identify a subtree of the minimum branching that has size greater than d, but with child trees each of size less than the d. We decrease the deficit d to ensure that the sub-tree we partiton "out" can accomodate a reference file as root. In step 2, we try to pack in as many child trees as we can within the space of d. If the total size goes beyond d after adding child tree m, we keep child trees $L_1, \ldots, L_{m-1}$ and fill in the remaining space by further partitioning child tree, L_m to generate $\hat{L}_m$. This is achieved by recursively calling the space-partition routine with $T = L_m$ and a renewed deficit. The algorithm terminates once T becomes a single node that cannot be further partitioned. Finally, we return the child trees selected along with their root edge as T_c.

Consider the minimum branching shown in Fig. 2 with $C = 9$. The only tree T_r which acts as a candidate tree is the one with $r = A_3$. Clearly, $P(T_r) + P(r) = 9 = C$ and its only child tree has size less than $C - P(A_1) = 9 - P(A_1) = 3$. Hence, $A_3 \rightarrow A_1$ with A_3 as the reference file is returned as T_c for $C = 9$ and placed on one server. The remaining tree is placed on the other server. The bound on the number of servers required by the SAP-Servers algorithm is crucially dependent on the number of reference files added by the algorithm. In the previous section, this was trivially calculated as $k - 1$, since we remove at-most $k - 1$ edges. In this section, this is more involved and we show in the following lemma that it is at most logarithmic in C/s. The key insight is that our space-partition routine recursively cuts down the deficit d (initially $d = C$) by at least 1/2 each time and returns once T_c becomes a single node.

Lemma 2: Given a set of files $\mathcal{A}$, the number of reference files added by the SAP-Servers-Delta algorithm is at most $g(\mathcal{A}) \leq \log_2(C/s)$.

Finally, we bound the number of servers required by SAP-Servers algorithms and prove their polynomial time complexity.

Theorem 3: Let the number of servers required by the optimal deduplication-based (k, C)-distribution be N_{opt} and the number of servers required by the (k, C)-distribution generated by the SAP-Servers-Delta and SAP-Servers-Dedup algorithms be $N_{\texttt{SAP-Delta}}$ and $N_{\texttt{SAP-Dedup}}$ respectively. For arbitrary files, we have,

- $N_{\texttt{SAP-Dedup}} = N_{\texttt{SAP-Delta}} \leq 1 + \frac{\Gamma(\rho) N_{opt}}{\lceil 1 - \log_2(\frac{C}{s})/(\frac{C}{s}) \rceil}$
- Both SAP-Servers algorithms have time complexity $O(\frac{n^2 s \log(C) \log(n)}{C})$.

As discussed in Corollary 1, $\Gamma(\rho)$, is a quadratic function bound by the maximum chunk popularity. Since its value is small for most practical cases, our SAP algorithms provide bounded space guarantees.

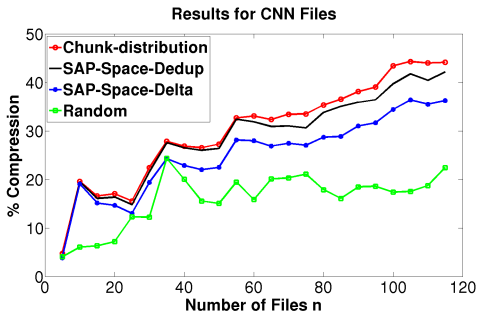
V. Experimental Results

We implemented our SAP-Space algorithms (for fixed number of servers k) in Java 1.6 and compared it with three other solutions: (i) *Chunk-distribution*, in which we identify the unique chunks in the given set of files $\mathcal{A}$ and spread them across the k servers. This solution is space-efficient but expensive in terms of the network accesses for each file. (ii) *Random partitioning*, in which we randomly partition the set of files in $\mathcal{A}$ across the k servers and then apply the deduplication function on each server. We performed basic experiments on two sets of files: (a) random web files downloaded from *CNN* (120 files) with file sizes varying from 22 KB to 182 KB and, (b) files belonging to two users in *Dropbox* (140 files) that contains sets of evolving file versions with file sizes varying from 13 KB to 132 KB.

For both these applications, we kept the chunk-size constant at 50 bytes and varied the number of files (n) in increments of 3 and measured the % compression achieved by each of the solutions w.r.t to the total size of the uncompressed files in $\mathcal{A}$. For each value of n, k was chosen to be $max(2, n/20)$ to make sure the number of servers increases slowly with the number of files. The results of our experiments are shown in Fig. 4.

For CNN files, on average, SAP-Space-Dedup achieves 28% compression, while SAP-Space-Delta achieves 25 % compression. Random partitioning and chunk-distribution achieve 15% and 30.1% compression respectively. For the Dropbox files, both SAP algorithms perform almost identically, indicating that for this example there is not much advantage of deduplicating the files after partitioning them. On average, the SAP-Space algorithms achieve 22% compression, while random partitioning and chunk-distribution achieve 15.0% and 23.3% compression respectively. These results confirm that our SAP technique achieves much better compression than random partitioning. Further, our technique achieves compression almost similar to the space-optimal access-inefficient chunk-distribution solution.

In Fig. 6, we illustrate through a conceptual plot, the trade-offs in terms of access-efficiency for the different approaches discussed in this section. While the SAP algorithms and



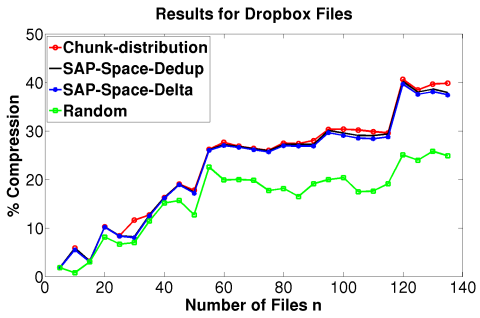


Fig. 4. Space Efficiency: SAP outperforms Random and is close to Chunk-distribution.

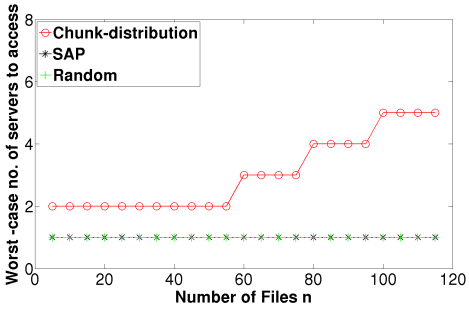


Fig. 5. Illustration of Access-Efficiency

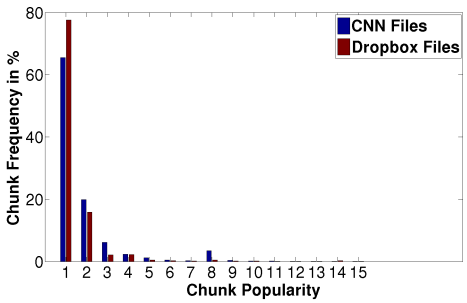


Fig. 6. Chunk frequency decreases very fast with popularity.

random partitioning requires exactly one server to access any file, the chunk-distribution solution could require up to k servers. We plot the graph in Fig. 6 based on our chosen value of k as a function of n, i.e., $max(2, n/20)$. Hence, while the SAP approach achieves nearly the same compression as chunk distribution, it can be much more efficient in terms of file access.

A key intuition for our bounded space guarantees (despite only considering pairs of files to generate the partitions) is based on chunk frequency distribution. Fig. 5 shows the chunk frequency distribution for both our data sets. A point (x, y) in the graph denotes that y % of unique chunks among all the files have a popularity of x, i.e., appear in exactly x files in the data set. As seen in the graph, the majority of chunks appear just in one, two or three files and almost no chunk appears in more than 7 or 8 files for either of the data sets. Hence, the *maximum chunk popularity*, ρ is bound by 7 or 8. This validates corollary 1, which in turn bounds the optimality gap of our SAP algorithms.

VI. Related Work

Deduplication is one of the most prevalent techniques for reducing storage. However, most of the existing systems either consider centralized systems, or require cross-server communication [7], [26], [19]. The deduplication technique, referred to as DEDE in [6], prevent cross-host communication, but uses a shared disk as a centralized reservoir in which file chunks are periodically written to. Similarly, the DDE system [12] avoids network communication costs by using a centralized file system. We, instead focus on a fully distributed system with no shared state (apart from index information on the client). Further, most of the literature focus on the various system and architectural challenges in implementing these solutions. Our focus is to present a provably space-efficient off-line algorithm for the distributed storage problem.

Delta-encoding is a technique widely used to capture the changes between versions of the same file [8], [5], [15]. In many cases it is used along with deduplication to transmit file changes to various servers in an efficient manner. In this paper, we suggest delta-encoding to capture differences between files that are not necessarily versions on one another. Further, we use delta-encoding to partition the files as well as store them (in the `SAP-Space-Delta` and `SAP-Servers-Delta` algorithms). So we suggest it as a potential alternate to deduplication. In [20], [24], the authors construct a graph to capture pair-wise differences and use the minimum branching or a traveling salesman heuristic to find optimal paths through the graph. They compare their solution, empirically against commonly used compression tools like *zip* and *cat*. The authors in [1] use such graph based techniques to compress web-graphs. We use a similar technique, but also present the theoretical framework to compare the space-efficiency of the technique with deduplication and use it for distributed storage.

In [17], [18], the authors make the point that using just the diffs between all files to identify similar files is an expensive operation since it requires $O(n^2)$ comparisons among

the files. Further, they present an approximate differencing technique called *Sif* which can be be used to find similar files in large systems efficiently. We can use such techniques to make our practical implementations far more scalable.

Finally, we note that there are many algorithms for document-clustering such as the agglomerative hierarchical clustering and k-means clustering [22]. These algorithms use a notion of similarity among files based on sophisticated measures such as the cosine measure, aimed more at capturing semantic similarities. This in turn leads to computationally expensive algorithms. We on the other hand, present polynomial time algorithms with similarity based purely on the size of the delta-encoding between the documents.

VII. Conclusion and Future Work

In this paper, we consider the problem of storing a set of files $\mathcal{A}$ that exhibit some similarity among distributed servers. Our main constraint, which we call access efficiency, is to ensure that all information corresponding to any single file can be acquired at a single server even in a distributed setting. To capture this constraint, we define the notion of a (k, C)-distribution of $\mathcal{A}$, which essentially prescribes a way to divide the chunks across all the files so as to ensure access efficiency. We first show that generating a space-optimal, access-efficient (k, C)-distribution is an NP-Hard problem. We present polynomial time similarity aware partitioning (SAP) techniques, to partition the files in $\mathcal{A}$ and store them across the servers such that the solution is access-efficient with a bounded space overhead. Our experimental results on files from *Dropbox* and *CNN* confirm the fact that our technique is much more space-efficient than a scheme in which we randomly partition the files (similarity oblivious) and store them on each server. In the future we wish to consider the additional factors of fault tolerance, load-balance and multiple-server communication to explore new trade-offs.

References

[1] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *Proceedings of the Data Compression Conference*, DCC '01, pages 203–, Washington, DC, USA, 2001. IEEE Computer Society.

[2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, pages 86–, Washington, DC, USA, 1995. IEEE Computer Society.

[3] B. Balasubramanian, T. Lan, and M. Chiang. SAP: Similarity-aware partitioning for efficient cloud storage. Technical report, Princeton University and George Washington University, 2013. Available as http://www.seas.gwu.edu/~tlan/papers/sap.pdf.

[4] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[5] R. C. Burns and D. D. E. Long. Efficient distributed backup with delta compression. In *Proceedings of the fifth workshop on I/O in parallel and distributed systems*, IOPADS '97, pages 27–36, New York, NY, USA, 1997. ACM.

[6] A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in san cluster file systems. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 8–8, Berkeley, CA, USA, 2009. USENIX Association.

[7] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming Space from Duplicate Files in a Serverless Distributed File System. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS '02)*, pages 617–624, Vienna, Austria, July 2002.

[8] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*, pages 113–126. USENIX, 2003.

[9] G. Forman, K. Eshghi, and S. Chiocchetti. Finding similar files in large document repositories. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, KDD '05, pages 394–400, New York, NY, USA, 2005. ACM.

[10] Y. Galambos, J. an Xu. A new method for generating bonferroni-type inequalities by iteration. In *Mathematical Proceedings of the Cambridge Philosophical Society*, 107, pages 601–607, 1998.

[11] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1199–1208, Washington, DC, USA, 2011. IEEE Computer Society.

[12] B. Hong and D. D. E. Long. Duplicate data elimination in a SAN file system. In *In Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST*, pages 301–314, 2004.

[13] G. Jäger and A. Srivastav. Improved approximation algorithms for maximum graph partitioning problems extended abstract. In *Proceedings of the 24th international conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS'04, pages 348–359, Berlin, Heidelberg, 2004. Springer-Verlag.

[14] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, IMC '09, pages 202–208, New York, NY, USA, 2009. ACM.

[15] D. G. Korn and K.-P. Vo. Engineering a differencing and compression data format. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ATEC '02, pages 219–228, Berkeley, CA, USA, 2002. USENIX Association.

[16] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. Technical report, Stanford, CA, USA, 1977.

[17] 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, pages 2–2, Berkeley, CA, USA, 1994. USENIX Association.

[18] U. Manber and S. Wu. Glimpse: a tool to search through entire file systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 4–4, Berkeley, CA, USA, 1994. USENIX Association.

[19] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *Trans. Storage*, 7(4):14:1–14:20, Feb. 2012.

[20] Z. Ouyang, N. D. Memon, T. Suel, and D. Trendafilov. Cluster-based delta compression of a collection of files. In *Proceedings of the 3rd International Conference on Web Information Systems Engineering*, WISE '02, pages 257–268, Washington, DC, USA, 2002. IEEE Computer Society.

[21] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[22] M. Steinbach, G. Karypis, and V. Kumar. A comparison of document clustering techniques, 2000.

[23] W. F. Tichy. RCS - a system for version control. *Softw. Pract. Exper.*, 15(7):637–654, July 1985.

[24] D. Trendafilov, N. Memon, and T. Suel. Compression file collections with a tsp-based approach. Technical report, 2004.

[25] B. Y. Wu, H.-L. Wang, S. Ta Kuan, and K.-M. Chao. On the uniform edge-partition of a tree. *Discrete Appl. Math.*, 155(10):1213–1223, May 2007.

[26] 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*, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association.