The Design of a Similarity Based Deduplication System

Lior Aronovich
IBM Corp.
lioraron@il.ibm.com

Ron Asher
IBM Corp.
ronasher@il.ibm.com

Eitan Bachmat
CS Dept., Ben-Gurion U.
ebachmat@cs.bgu.ac.il

Haim Bitner
Marvell Corp.
haimb@marvell.com

Michael Hirsch
IBM Corp.
hirschm@il.ibm.com

Shmuel T. Klein
CS Dept., Bar-Ilan U.
tomi@cs.biu.ac.il

Abstract

We describe some of the design choices that were made during the development of a fast, scalable, inline, deduplication device. The system's design goals and how they were achieved are presented. This is the firs deduplication device that uses similarity matching. The paper provides the following original research contributions: we show how similarity signatures can serve in a deduplication scheme; a novel type of similarity signatures is presented and its advantages in the context of deduplication requirements are explained. It is also shown how to combine similarity matching schemes with byte by byte comparison or hash based identity schemes.

Categories and Subject Descriptors H.3.1 [*Information Storage and Retrieval*]: Content Analysis and Indexing—Indexing methods; E.5 [*Data*]: Files—Backup/Recovery

General Terms Design, Performance

1. Introduction

Storing large amounts of data efficientl , in terms of both time and space, is of paramount concern in the design of backup and restore systems. Users might wish to periodically (e.g., hourly, daily or weekly) backup data which is stored on their computers as a precaution against possible crashes, corruption or accidental deletion of important data. It commonly occurs that most of the data has not changed since the last backup has been performed, and therefore much of the current data can already be found in the backup repository, with only minor changes. If the data, in the repository, that is similar to the current backup data, can be located efficientl , then there is no need to store the data again, rather, only the changes need be recorded. This process of storing common data once only is known as data deduplication. Data deduplication is much easier to achieve with disk based storage than with tape backup. The technology bridges the price gap between disk based backup and tape based backup, making disk based backup affordable. Disk based backup has several distinctive advantages over tape backup in terms of reducing backup windows and improving restore reliability and speed.

In a backup and restore system with deduplication it is very likely that a new input data stream is similar to data already in the repository, but many different types of changes are possible. The new data may have additions and deletions when compared with previous data. It may also have a different block alignment. Given the potential size of the repository which may have hundreds of terabytes of data, identifying the regions of similarity to the new incoming data is a major challenge. In addition, the similarity matching must be performed quickly in order to maintain high backup bandwidth requirements.

There are many deduplication systems, both scientifi and industrial, which are based on identifying

SYSTOR'09, May 4-6, Haifa, Israel

identical segments via hash functions, see [3, 22, 24, 29], among others. However, these systems, currently, do not scale to support a petabyte of deduplicated data because they require very large index tables for their support. To obtain high performance such index tables must be summarized in cache and this translates into large cache requirements, which may substantially raise the cost of the solution.

In this paper we describe an efficien system for achieving high speed data deduplication in very large repositories, as implemented in the IBM TS7650G ProtecTier. The paper explains our design choices and more generally outlines the design possibilities for a large class of deduplication systems, whose common feature is the identificatio of data similarities.

From the outset the system design requirements included support for a repository with a petabyte of deduplicated data (a usable physical capacity of 1 petabyte), at a deduplication rate of at least 350MB per second, using only 4GB of cache memory and byte by byte comparison for verifying identical data. These design goals were achieved and even exceeded, leading to a highly scalable system.

The main difference between the system described in this paper and segment identity based systems is that this system computes similarity indicators for large chunks of data rather than identity indicators for smaller segments. Similarity indicators were firs introduced in the context of plagiarism detection [5, 25, 26] and identificatio (and elimination) of near duplicate documents [6, 14, 20].

Working with similarities rather than identities allows us to work at a coarser granularity than identity based systems and with smaller hash signatures whose computation is optimized.

We describe new similarity signatures which, for deduplication requirements, improve upon the previous ones described in [5, 6, 14, 20]. The signatures detect similarity even if there are considerably changes to the data, all that is required is that a fair percentage of data blocks remain unchanged. The small size of the index allows us to store it in fast random access memory. The similarity signatures detect similarities with a high probability, make few errors and are computed efficientl .

Once a similarity between an incoming piece of data (a chunk) and a piece of the repository has been identified the system performs a byte by byte comparison between the repository data and the new chunk and stores only the difference. The data from the search stage is used to anchor the comparison, leading to an efficien comparison process.

This allows us to incorporate new data into the repository in deduplicated form, on the fl , without compromising bandwidth and with strong data integrity guarantees.

We will also describe an alternative design that is based on similarities, yet retains the identity hash comparison of identity based systems, rather than the byte by byte comparison.

The paper is organized as follows: In Section 2 we describe segment identity based schemes. Section 3 then describes similarity based systems and in particular our system with an emphasis on the design criteria that have led us to our choice of similarity based signature. In Section 4, we present results from some lab tests, compare the results to those of other systems and present some data from a real production system.

2. Identity based deduplication

Searching for similar data may be considered an extension of the classical problem of pattern matching, in which a repository (text) T of length n is searched for the appearance of a string P of length m. Typically, n is much larger than m. Many publications present search methods which attempt to solve this problem efficientl , that is, faster than the naive approach of testing each location in T to determine if P appears there. By preprocessing the pattern, some algorithms achieve better complexity, for example see [18, 4, 17] All of these algorithms work in time that is of order $O(n+m)$, which means that the search time grows linearly with the size of the repository.

One problem with these algorithms is that they are not scalable beyond some restrictive limit which falls far short of the petabyte range. Another disadvantage of the above algorithms is that they announce only exact matches, and are not easily extended to perform approximate matching.

Instead of preprocessing the pattern, one may preprocess the repository data itself, building a data structure known as a *suffix tree*; this is described in [28, 27]. If preprocessing is done off-line, then the preprocessing time may not be problematic. Subsequent searches can then be performed, using a suffi tree, in time $O(m)$ only (i.e., depending only on the pattern size, not on

the text size), but again, only exact matches can be found. Moreover, the size of the suffi tree may be larger than the repository itself. For backup and restore, it would be desirable to use an algorithm for approximate pattern matching because it will usually be the case that not an exact replica of the input data can be found in the repository, but rather a copy that is strictly speaking different, but nevertheless very similar, according to some define similarity criterion. Approximate pattern matching has been extensively studied, as described in [12, 23] One recent algorithm works in time $O(n\sqrt{k\ \log(k)})$, where n is the size of the repository and k is the number of allowed mismatches between the pattern and the text [1]. We note again that this is at least linear in the size of the repository and not feasible for large k.

Another family of algorithms is based on hashing functions. These are sometimes known as CAS (Content Addressable Storage), as described in [22, 24, 29], among others. The general paradigm is as follows: the repository data is partitioned into segments, and a hash value, also called a fingerprin or a signature, is produced for each segment; all of these hash values are stored in an index. To locate some given input data, called the version, it is also broken into segments and the same hash function (that has been applied to the repository blocks) is applied to each of the version segments. If the hash value of a version segment is found in the index, a match is announced. Since in many cases bytes are added or deleted, segments may shift and thus a match may be missed.

To avoid this issue, it is customary to use variable size segments with content determined boundaries. Typically, a boundary is declared any time a reasonable, byte dependent, hash function has a particular k bit pattern. Assuming that hash values are uniform, this will lead to variable sized segments with an average size of 2^k bits and reasonably small variance. We note that shifting bytes will correspondingly shift the boundaries, preserving the matching.

The advantage of CAS over the previous methods is that the search for similar data is now performed on the index, rather than on the repository text itself. If the index is stored using an appropriate data structure, the search time may be significantl reduced. In particular it can be done in time which is essentially independent of the size of the repository data.

There are, however, disadvantages to this scheme. As before, only exact matches are found, that is, only if a segment of input is identical to a segment of the repository will a match be announced. One of the requirements of a good hash function is that when two segments are different, even only slightly, the corresponding hash values should be completely different, which is required to assure a good distribution of the hash values. But in backup and restore applications, this means that if two segments are only approximately equal, a hashing scheme will not detect their similarity. This creates a dilemma of how to choose the size s of the segments. If a large segment size is chosen, one achieves a smaller index (since the index needs to store n/s hash values) and the probability of a false match is reduced, but at the same time, the probability of find ing a matching segment is reduced, which ultimately reduces the efficien y of deduplication. If, on the other hand, a small segment size is chosen, the overall effi ciency of deduplication may increase, but the increased number of blocks may require an index so large that the index itself presents a storage problem.

Consider the case of a petabyte (2^{50} bytes) of deduplicated data. A typical segment size for identity based systems is 8K which is 2^{13} bytes [29]. This means that the system will contain 2^{37} segments. Using a Bloom filte optimization [2] as in [29], we would need roughly one byte per segment to verify whether it is a copy of a segment in the repository. This means that at a minimum we would need 2^{37} bytes of memory and any additional information about the whereabouts of the identical segment would likely lead to 2^{39}–2^{40} bytes of memory or roughly a terabyte. To make the deduplication efficien the system described in [29] stores this information in memory rather than on disk. Since it is assumed that the hash table is large enough to avoid random collisions, once a match has been found there are no further comparisons and an identity is declared between a new segment and an existing repository segment. The repository data itself is not accessed.

3. Similarity based deduplication

Unlike hash functions which are built to detect exact matching, we base our deduplication scheme on detection of similarities. Looking for similarities means that we can substantially increase the size of the basic units which we call chunks rather than segments. Since chunks are much larger than segments there are fewer

of them and therefore the size of the memory index can be substantially reduced. Working with 16MB chunks, we show that one can store all the relevant information about their similarities in roughly 64 bytes of memory per chunk, leading to 4GB of memory for 1 petabyte of deduplicated data. It is important to note that our calculations assume that the petabyte of data is deduplicated (essentially unique), however, we do not assume that it is compressed. Most deduplication systems, including the one described here, compress data after deduplication. This enlarges the amount of data resident in the system and consequently increases the size of the index. Thus, for example, if 30 petabytes of backup data are reduced in size via deduplication to 3 petabytes and then compression further reduces the size of the data to 1 petabyte, we would need 12GB of memory, rather than 4GB, because compression acts on the data and not on the index.

Another requirement for our deduplication application was the ability to deduplicate inline. By this we mean the ability to deduplicate on the fl , sending to disk only the deduplicated data. The other approach is to firs back-up to disk and then at a later time, deduplicate as a background process. This process requires more disk accesses and longer backup resources which can also slow down foreground applications. For a deduplication application to work inline it must determine very quickly whether an incoming piece of data is likely to be similar to other data which was previously backed up. The metadata, which is required to make such a decision therefore lies in memory. Working with chunks makes this possible since all data can fi in memory.

As noted earlier, the use of similarities, rather than identities, necessitates an added step of comparing the new data to old data from the repository which is similar to it. One of our design goals was to do that using a byte by byte comparison which is safer than relying on hashes to establish identity. To get good performance we integrate the similarity findin process with the byte by byte comparison.

We now describe the system in more detail. As stated earlier, we consider the case of a petabyte of deduplicated data, while ignoring the multiplicative effect of the compression ratio on the size of the index. Also, to simplify matters we will assign specifi values to the different parameters. In addition, we use rather naive, yet fairly accurate, probabilistic arguments. The naivety of the discussion has a negligible effect on the computations and has no bearing on the correctness of comparative statements.

3.1 The similarity signatures

Consider a stream or streams of data which are sent to backup. The data is partitioned into *chunks* of equal size, say 16MB $= 2^{24}$ bytes. Denote by b_i the byte in position i of the chunk, and choose a basic unit of small size, say one block (512 bytes). The i-th unit of a chunk, denoted J_i, consists of $b_i b_{i+1} \cdots b_{i+511}$, the sequence of 512 consecutive bytes starting at byte-position i, for $i = 0, \ldots, 2^{24} - 512$.

Informally, a similarity signature for a chunk consists of a small number of block based signatures, each consisting of a small number of bytes, such that if the chunk shares a considerable number of blocks with another chunk (say, more than some threshold), there is a reasonable probability (threshold dependent), that the two chunks will share one or more common block based signatures. We also want the property that if two chunks are not similar, which means that they have no blocks in common then with high probability the similarity signatures of the chunks will not have common block based signatures.

We compute a similarity signature for each chunk as follows. A hash function is then computed for each block sized unit, which is conveniently done using a sliding window and a rolling hash function. Let $P > 256$ be some prime number. The Karp-Rabin signature [17] of a number n, denoted $S(n)$, is simply $S(n) = n \mod P$, the remainder after dividing n by P.

Given the unit J_i, we can think of the sequence of consecutive bytes $b_i \cdots b_{i+511}$ as a large number with 512 digits written in the base of 2^8, with j_{i+511} being the least significan digit, or equivalently as a number with 4096 binary digits (bits) written in base 2. Either way we get the same number which we still call J_i. Since $P > 256$, the signature of a single byte is itself.

The Karp-Rabin signature can be used as a rolling hash function, as the value of the signature on J_i can be easily computed from the value of J_{i-1}. Indeed, J_i is obtained from J_{i-1} by firs discarding the old most significan byte b_{i-1}, then shifting the common bytes $b_i b_{i+1} \cdots b_{i+510}$ one byte to the left (multiplying by 2^8), finall adding the new byte b_{i+511}. This leads to the formula

$$J_i = (J_{i-1} - b_{i-1} * 2^{4088}) * 2^8 + b_{i+511}.$$

The same formula applies to the signatures, namely we have

$$S(J_i) = S(S(S(J_{i-1}) - S(2^{4088} * b_{i-1})) * 2^8 + b_{i+511}).$$

Assume that P has 55 bits. The formula shows that we can calculate $S(J_i)$ from $S(J_{i-1})$ with high efficien y on a 64 bit machine. Indeed, all the values will have at most 64 bits, multiplication by 2^8 is simply a shift and since $S(2^{4088})$ is a constant, the values of $S(2^{4088} * b_{i-1})$ can be pre-calculated and placed in a lookup table of 256 entries. In addition, if the prime P is chosen randomly one can prove that the Karp-Rabin signature will have nice properties as a hash function, see [17].

Let $k_i = S(J_i)$ be the signature of the i-th unit. Choose the h largest hash values, viewed as integers, among the k_i (for the purposes of this paper, consider the case $h = 4$), and let $i_1, \ldots, i_4$, be the indices of the units whose hash values were the largest, that is, $S(J_{i_1}), S(J_{i_2}), S(J_{i_3})$ and $S(J_{i_1})$ are the 4 largest values of the $2^{24} - 511$ signatures $S(J_i)$ produced for a given chunk.

A good hash function usually spreads its values uniformly over its range, but the maximal values k_i are not uniformly distributed, and will have a tendency to belong to a small sub-interval at the higher end of the range. Therefore, choosing the values k_i as signatures of the chunk would lead to many collisions. The main novel idea of the suggested signature is to use the values $i_1, \ldots, i_4$ only to get the *location* of some special units, and to use as *values* not the signatures at these locations, but to apply some small shift of m bytes to the locations, with $m > 0$, and defin the signatures as the values

$$S(J_{m+i_1}), \ldots, S(J_{m+i_4}).$$

Since another property of hash functions is that even a small change in the argument generally leads to a significan change in the hash value, the hashes at the shifted locations will again be uniformly distributed, as requested. The shift m should be small so that it is unlikely that the index $i_4 + m$ will not be within the chunk range. For the purposes of this paper, we will consider the choice $m = 8$.

The similarity signature values of all the chunks in the repository are held in memory. In addition, the location of the chunk and the specifi index within the chunk which gave rise to the signature are also listed. A given similarity signature may belong to more than one chunk, in which case all its appearances are listed.

When a new chunk is considered, its 4 similarity signatures are calculated. If any of them equals an existing similarity signature, a process of comparing the new chunk to an existing piece of the repository begins, in order to fin identical portions. This process, which will be explained in more detail below, requires to bring data from disk, therefore the similarity assumption should be statistically justified

If no suspected similarities are found, the similarity signatures of the new chunk are added to the index of similarity signatures which is stored in memory. The index contains the similarity signatures themselves, 7 bytes each, along with a pointer to the block which gave rise to the signature, again 7 bytes. The index will therefore contain $14 * 4$ bytes for each chunk of 2^{24} bytes. As there can be at most 2^{26} different chunks in a petabyte, the required cache size will be at most $2^{26} * 56$ which is about 3.5GB, as required. We still have some memory left for other calculations and requirement.

To explain our choice of similarity signatures, consider firs two other possibilities. The firs can be traced to Brin et al. [5], while the second to the work of Broder [6, 7, 8, 9, 10, 11] and independently to Heintze [14], with a similar approach presented in [20]. The contexts of findin similarities in these papers is somewhat simpler since they try to fin similarities between file which are well define objects with specifie content, nonetheless they are useful in our context as well. The third option represents our choice, and we show how it merges the advantages of the two previous choices. Again we consider the case of $h = 4$ signatures.

1. Choose 4 random blocks $J_{i_1}, ..., J_{i_4}$ of 512 bytes within the chunk and let its Karp-Rabin signature be the similarity signature of the chunk.
2. Compute the Karp-Rabin signatures of all the blocks J_i within a chunk, and take the 4 maximal value as the similarity signature.
3. (Our choice) Compute the Karp-Rabin signature of all blocks J_i within the chunk. Assume that $J_{i_1}, ..., J_{i_4}$ produced the maximal values, then take the Karp-Rabin signatures of J_{i_l+8}, $l = 1, ...4$ to be the similarity signatures of the chunk.

The choices will be judged according to the following four criteria:

A – Processing speed. The processing of a chunk should be as efficien as possible.

B – Similarity detection. We would like to maximize the probability that a chunk A which is similar to some chunk B in the repository will be identifie as such.

C – Elimination of false positives. If a chunk is not similar to any other previous piece of the repository, the algorithm should not falsely claim that it is.

D – Resolution. In case there are several previous pieces which are similar to a new chunk A, the algorithm should point to the one which is most similar with high probability.

Consider the firs choice, firs with respect to criterion **B**, namely, similarity detection. Suppose that a new chunk A is given, and that B is a piece of data in the repository, roughly the size of a chunk, that has a portion $0 \leq x \leq 1$ of its blocks in common with A. We consider pieces of data in the repository, rather than chunks, because chunk boundaries may shift and the piece of data which is similar to a new chunk may be composed of portions of more than one old chunk. Since the signature of a completely random block in B has been chosen, there is a probability x that the block is still part of A. In that case, if one computes the Karp-Rabin signatures for all blocks J_i in A, one will fin among them the similarity signature of B with probability x. The amount of space which will be required to save the difference between A and B is approximately a $1-x$ fraction of the size of A.

Assuming 4 signatures, the probability of having a common signature between A and B becomes $1-(1-x)^4$. For example, consider $x = 0.5$. In that case, with probability $15/16$ one will identify the opportunity to write a new data set which is half the size of A. The opportunity is wasted with probability $1/16$. The same computation for $x = 3/4$ shows that the opportunity to write a set which is $1/4$ of the size of A is wasted with probability $1/256$.

Assuming that a new chunk A has a similarity of x with some piece of the repository with probability distribution function $r(x)$, the average amount of data written per new block will be

$$\int_0^1 ((1-x)(1-(1-x)^4)+(1-x)^4)r(x)dx$$

$$= \int_0^1 ((1-x)+(1-x)^4-(1-x)^5)r(x)dx.$$

This should be compared with the ideal deduplication where one always identifie the similar piece in the repository, in which case the average written data has size

$$\int_0^1 (1-x)r(x)dx.$$

Consider for example the uniform distribution given by $r(x) = 1$, the ideal amount of data will then be

$$\int_0^1 (1-x)dx = 1/2,$$

while according to our similarity identificatio scheme one would get

$$\int_0^1 ((1-x)+(1-x)^4-(1-x)^5)dx$$

$$= 1/2 + 1/5 - 1/6 = 1/2 + 1/30 = 16/30.$$

As can be seen, the difference is minuscule in this case. Experience with actual systems shows that $r(x)$ tends to be concentrated in the interval $[0.9, 1]$ in which case the difference is even smaller.

We now consider the fli side of similarity detection, which is criterion **C**, namely false positives or pseudo matches. Assume that A is really a new chunk which is not similar to any piece of the repository. What are the chances that one will mistakenly think that A is similar to an existing piece. If that happens, time will be wasted to bring the pseudo similar piece from the disk.

Examining a chunk A, one computes the 2^{24} Karp-Rabin signatures of its blocks J_i, and asks whether these match any of the roughly 2^{28} similarity signatures in the repository. There are roughly $2^{28} * 2^{24} = 2^{52}$ pairs of possible matches. While matches among these pairs are not really independent events, they have very low correlation and can be assumed to be independent. The total range of possible signatures p has at most 55 bits, so the probability for a single pair to match is roughly 2^{-55}, a number which is dangerously close to the reciprocal of the number of pairs. The probability of a random matching is $1-(1-2^{-55})^{2^{52}}$, which is roughly $1/9$. While this is a rather small number, it is non-negligible.

In addition it requires to take care of another issue: if the new chunk is incorrectly flagge as being similar to older data, then it might happen that the chunk's similarity signatures will not appear in the index, so one will not be able to identify future similarities using them. This can be overcome by insisting on a minimal similarity following the bit by bit comparison, beyond which the chunks are treated as unconnected, but again this comes to show that pseudo similarities are not desirable. The number of pseudo matches can also increase if one chooses a smaller p (to save some cache space) or increases the resolution, say to 8 signatures per chunk.

Consider now the behavior of the second choice with respect to criterion **C**. By choosing the 4 largest Karp-Rabin signatures as the similarity signatures, the search in the new chunk is narrowed from all 2^{24} signatures to just the top 4. Comparing these 4 to the 2^{28} signatures in the repository gives only 2^{30} pairs, which is a lot less than the 2^{52} we had previously. Unfortunately, being maximal values and therefore large by definition means that the signatures are not uniformly distributed over all 2^{55} possible values. Consider for example the largest value. The probability that the largest value is in the range $[P - 2^{31}, P - 1]$ is $(1 - 2^{-24})^{2^{24}} \sim e^{-1} \sim 0.37$, whereas the probability that a uniformly distributed value would fall into that range is only 2^{-24}. The effects of the smaller range and the smaller number of locations of the similarity signatures roughly cancel out, and the probability of a pseudo similarity remains roughly the same. There is a small penalty in terms of the probability of identifying a match (criterion **B**), but we will discuss this issue in the context of the third option.

The third option solves the problem of non uniformity of the similarity signatures, while retaining the good property of the second choice, that the possible locations within the chunk of the similarity signatures are anchored near maximal values. The values of the Karp-Rabin signatures are nearly uniform since the 8 new bytes that enter the calculations, 64 bits in all (more than the 55 bits of the signature) smooth out the non uniformity which was inherent in the large values. We are now in a situation where 4 (nearly) uniformly distributed signatures are compared with 2^{28} signatures that have been uniformly chosen from a 55 bit range, so the chance of a random matching is about $2^{28+2-55} = 2^{-25}$, which is a tiny probability. This also means that if one modify with the numbers slightly, changing the number of similarity signatures, the sizes of the chunks or even reduces the size of the signature to 40 bits to make the index smaller, there will still be very few pseudo similarities. The solution is very robust in this way.

The second and third choices have a distinct advantage in terms of processing speed, criterion **A**, over the firs choice. We note that the firs choice requires a comparison of each of the 2^{24} signatures to the index, whose size is measured in gigabytes. These accesses are therefore not aimed at the fastest cache of the processor. In comparison, only 4 word sized registers are needed to hold the 4 largest signatures and the 2^{24} comparisons are performed against one of them, the one that holds the fourth largest signature. Very rarely (logarithmically many times) additional comparisons are needed with the registers holding the larger signatures. Only 4 searches are then performed against the large secondary cache.

The second and third choices pay a small price in terms of criterion **B**. Before, if the new chunk A had a similarity of x to a repository piece B, then per signature there was a probability x that the similarity signature would remain intact as one of the signatures which were computed for A. In the new situation, not all signatures of J_i are considered when comparing with the similarity signatures of B, rather only 4 specifi ones are compared. For them to survive intact, we need that the nearby maxima still remain maxima of A. Consider the data in B and the data in A which is not in B. The size of the union, in fractions of a chunk, is $1 + (1 - x) = 2 - x$. Let us compute the 4 maximal signatures of the union. Each of the 4 maximal values has a $1/(2 - x)$ chance of coming from B. In addition they have an x chance of surviving in A. If any comes from B and survives in A, it will allow us to detect a possible similarity between A and B. We conclude that the probability of similarity detection is

$$1 - (1 - x/(2 - x))^4.$$

Plugging in $x = 1/2$ for example, leads to $1-(2/3)^4 = 65/81 = 0.8$. This is not as good as the 0.94 that we had before for the firs choice, but still fairly good. In the more realistic situation where $x = 0.9$ or more, the result is essentially the same as with the random signatures, since the extra factor of $2 - x$ is very close to 1. This shows that in terms of similarity detection, the

second and third choice act nearly as well as the first If we are still bothered with the discrepancy between the methods, one can reduce or eliminate the difference in several ways. First one may compute the $h > 4$ maximal values, preserving only the top 4 in the index for new chunks. As h grows, the issue of detection becomes the issue of the survival of any of the 4 top signatures of B in A, and this is precisely the computation for the firs choice. Even mild choices such as $h = 8$ will nearly equate the probabilities. This has essentially no effect on the speed of processing or the probability for false positives.

Another option which is applicable only in the third choice is to take a smaller value of P with 40 bits. This will allow in the same cache size (even smaller) to hold 5 signatures per chunk. The fift signature will improve the similarity detection probability for $x = 1/2$ to 0.88. In fact, for the third choice, both techniques can be combined to get a better detection probability than either of the firs two choices.

We still need to address another issue, criterion **D**, which is related to similarity detection and that is the issue of multiple similarity choices or stated otherwise, an embarrassment of riches. There may be more than one piece of the repository that is detected as being similar to A. Assume that these pieces are $B_1, \ldots, B_k$. Their portion of similarity with A will be $x_1, \ldots, x_k$. We want to be able to fin the B_i, with the largest similarity x_i with high probability. The simplest differentiator is the number of similarity signatures which the chunk B_i shares with A. To simplify the discussion, assume there are 2 candidates B_1 and B_2 and that B_1 is in fact identical with A. Obviously all 4 similarity signatures will be common to B_1 and A. Suppose B_2 is not identical to A, hence $x = x_2 < 1$. For B_2 to match B_1 as a candidate for similarity with A, it will have to share with A all 4 similarity candidates as well. The probability for that to happen is x^4 in the case of the firs choice, and $(x/(2-x))^4$ in the second and third choices. The probability that we will not be fooled is therefore

$$1 - x^4$$

for the firs choice and

$$1 - (x/(2-x))^4$$

for the second and third choices. If $x = 0.75$, we get $1 - 81/256 = 0.7$ in the firs case and $1 - 81/625 = 0.88$ with the second or third choices. We see that the second and third choices provide much better resolution than randomly chosen similarity signatures and are not easily fooled. When $x = 0.9$ the probability of not being fooled is about 0.4 in the firs case and 0.6 for our choice. The penalty in this case is diminished though since 90% similarity will also produce very good space savings. We note that a fift signature per chunk will improve the resolution further.

The issue of resolution between competing similarity candidates also explains why it is better to have several similarity signatures per chunk. If there were just a single signature for say a 4 MB chunk, one would have been more easily fooled by chunks with less similarity. On the other hand using very large chunks with many similarity signatures is also not a good idea since the level of similarity within the chunk will be uneven and different parts of the chunk may be related to different pieces in the repository. The amount of time needed to bring a very large piece of data from the disk will be large, and bringing several pieces for different parts of the chunk may be prohibitive. Therefore we must strike a balance between the number of similarity signatures and the size of the chunk, which explains our choice of a relatively small number of signatures, 4, which still gives good resolution. This resolution can be further improved by considering recent chunks and preferring pieces of the repository which are contiguous with pieces that proved similar to the previous chunks.

We note that it is possible to increase resolution (criterion **D**), using a method of Broder [6], who uses super signatures which are signatures of sets of signatures. His method is inspired by his application which is to identify near identical (rather than simply similar) correspondences between chunks and repository pieces. The method, which can be applied in the context of the second and third choices, requires a penalty in terms of processing speed, since it requires the computation of some permutations on the signatures. It seems that it is not needed for deduplication purposes.

To summarize, the firs choice does well with respect to criterion **B** and to a lesser degree criterion **D**. The second choice does well with respect to **A**, **B** and **D**, while the third choices does well with respect to all criteria and is at least as good and sometimes better than the other two on all accounts. A deduplication system can use either of the three choices but it seems that the third choice is the most effective.

3.2 Post similarity match processing

Once a chunk A has been found to be similar to a specifi piece of data B, a byte by byte comparison is performed which will leave us with the difference. The process of taking the difference, storing and indexing it, is just as crucial to the success and efficien y of the deduplication device as the similarity search. However, since in this paper we chose to highlight similarity based deduplication, this process will be discussed in less detail. There are many known techniques for working out the comparison, some of which are related to our specifi technique. We refer the reader to [3, 15, 19] and their bibliography for more information on this issue. As discussed earlier, some of the methods do not compare bytes but rather compare hashes which are produced from the chunk and the repository piece. We describe how such methods can also work with similarity based matching and what are the tradeoffs involved.

Each of the 4 similarity signatures can match to different pieces in the repository, belonging, say, to backups taken on different days, as may happen when one cuts and pastes parts of documents. The matches which correspond to a specifi piece of the backup stream are identifie by measuring address differences. Two matches are said to be *coherent*, if the address difference between the locations of the blocks in the chunk and the address difference of the matching blocks in the repository are equal (or at least close to each other). The matches are divided into sets of mutually coherent matches. Each mutually coherent set suggests a match between the incoming chunk and a single piece of the repository which is roughly the size of a chunk. The repository piece which has the largest number of coherent matches with the incoming chunk is read from disk for the purpose of comparison. There are several tie breakers in addition to the number of matches. The specifi matches coming from the signatures serve as anchors for the comparison between the chunk and the repository piece.

The matching between the blocks, which occurred at the signatures level, is verifie as an actual identity between the corresponding block. This identity (if it holds) is then extended backwards and forwards as an identity between a portion of the chunk surrounding the block and a portion of the repository piece surrounding the matching block. The procedure is performed around any match in the coherent set which corresponds to the piece. If the forward identity region of one matching block meets the backward identity region of the next matching block, the regions are coalesced. At the end of the procedure we are left with one or more non overlapping regions of the chunk which are identical to a corresponding set of non overlapping regions in the repository piece.

Between the regions there may be islands of non matched data in both the chunk and the repository piece. One can then try to fin fine , partial matches, between portions of the islands in the chunk and the corresponding islands in the data piece. The procedure can now be carried out recursively for each island. Since the signatures of blocks in the chunk have already been computed, one can compute samples of the Karp-Rabin signatures of blocks from the islands of the repository piece and match again, extending each match as previously to identical regions. Most of the calculations are typically simple byte by byte comparisons, which can be aggregated to words. The number of comparisons is essentially given by the size of the chunk, leading to high efficien y. At the end of the process identities between various regions of the chunk and the repository piece are resegmented to produce a decomposition of the chunk into a sequence of segments, some of which are identical to previously existing segments. This sequence is indexed and stored.

After the data which needs to be written to disk is identified a Lempel-Ziv like compression algorithm is applied to the data before it is stored. We refer the reader to [30] for more on the Lempel-Ziv algorithm.

One feature of identity based de-duplication schemes, is that they can avoid byte by byte comparisons if their hashes are considered to be large enough. The main advantage is that instead of reading from disk, the repository piece which is supposed to match the chunk, we can read only the signatures of the data segments. Assuming 8KB segments, the signatures are about 50 times smaller than the segments themselves. If we write on disk an index of hashes and if the hashes corresponding to a specifi chunk happen to be written sequentially in the disk, then reading only the hashes can result in high savings.

Disk accesses are not always the performance bottleneck for deduplication systems, but in those cases where we use a small number of slow disks for our storage, they may become the bottleneck. In our current implementation, we avoid hash based comparisons, since

we (and many customers) believe that it is safer to use byte by byte comparisons. We would like to show, however, that we can implement such ideas within our framework of similarity based signatures.

To see how it can work, consider our choice of Karp-Rabin similarity signatures, but assume that the prime which is used has 200 bits instead of 55. In addition every time that some sequence of 13 specifi bits of the hash has a certain fi ed value, we mark the location within the chunk of this occurrence as a segment boundary. This segmenting procedure goes back to the work of Manber [20]. For each segment, we also compute its Karp-Rabin signature. This requires almost no overhead beyond our previous computations. To see this, consider the process of taking two numbers, M_1 and M_2, each with m bytes, and concatenating their byte sequences to produce one large number M with $2m$ bytes. In terms of Karp-Rabin signatures we have

$$S(M) = S(S(M_1) + S(S(2^{8m}) * S(M_2)))$$

The factor $S(2^{8m})$ can be pre-computed and placed in a table. We see that the signature of the concatenated sequence can be computed using a small number of arithmetic operations on m-byte integers. A segment will be the concatenation of its non-overlapping blocks and a fina partial block remainder. The Karp-Rabin signatures of the blocks are computed anyway, and we will need to compute the Karp-Rabin signature of the remainder, which is at most the size of a single block separately. The expected number of concatenations will be 16, since there are expected to be about that many blocks in a segment, so the overall overhead beyond our basic computation is very small. In terms of criterion **A** (computational efficien y), the main issue is that the computations do not take place within words of a 64 bit processor. Still the process is very efficient The problem of false positives is essentially eliminated by the use of the much larger prime, while similarity detection and resolution remain as before. The size of the similarity signature index will increase from 56 bits per chunk to 128 bits per chunk, which is a doubling of the space, but since the index is so efficien to begin with, such an increase can certainly be tolerated for a petabyte of data. Alternatively we can retain the old 55 bit signature computations to be done in parallel to the new signature computation and preserve the old index for similarity matching.

Either way, when a new chunk enters the system, we identify a similar piece of the repository using the similarity index as described before. Once the piece is identified the (200 bit) hashes of the segments of the piece, rather than the piece itself, are read from disk and compared with the hashes of the chunk. The unmatched data and hashes are then stored to disk. Since we have decoupled the similarity index and the hash comparison, we can decrease the size of the segments to obtain better deduplication ratios without affecting the size of the index. This is an advantage over classical, segment identity based systems.

3.3 System architecture

The IBM TS7650G ProtecTier system, some of whose design features are described in this paper, is a gateway which consists of processors and memory for storing the index and other metadata. In the standard config uration, the gateway (node) has a quadcore 2.9 GHz Xeon processor and 32GB of RAM memory. The gateway runs Diligent's VTL (virtual tape library) and the deduplication application which uses the HyperFactor technology which the paper partially describes.

The gateway is connected on one side to servers which generate the back-up streams and on the other side to an external storage system in which the deduplicated data is stored. Diligent's VTL makes the disk storage system seem to the servers like an automated tape library. The servers issue the backup and restore commands as if the data is stored on tape. Two gateways can be combined to form a clustered failover pair which works on the same repository. In the case of a failure to one of the two gateways, the streams are rerouted to the other gateway which takes over responsibility for all deduplication activity, until the other gateway is repaired or replaced. This option improves the reliability of the system, an important feature in enterprize class solutions. The fact that the gateways deduplicate a common repository improves deduplication ratios in comparison with a two node system each working separately on half the repository, since each node can now detect similarities over a wider repository.

4. Experimental Results

In this section we provide some experimental results, based on data from a large customer installation, and also report on some lab tests which checked the rate

of deduplication that the system is capable of. We also compare the results, with those reported for other systems.

4.1 Comparison of throughput and capacity with other systems

Real customer data cannot always provide evidence on the performance envelope of a product since the user may not be driving the system to its peak capabilities. Consequently, we provide the results of some lab testing which was designed specificall to test the system's performance limits. The tests were conducted by an independent company, Enterprise Strategy Group Inc. (ESG), which was asked to verify the throughput for a system with a single node and a clustered system of two nodes working on the same repository. The full report can be found in [13], we recall here some of the results which provide some relevant information on the system's capabilities.

The system that was tested had a clustered pair of two IBM TS7650G gateways (nodes) based on an IBM x3850 server each with a quad core 2.9 GHz Xeon processor and 32 GB of RAM (a standard configura tion). The nodes were attached to a storage repository consisting of an IBM DS8300 with 256 300 GB 15K RPM FC drives configure with 36 RAID-5 LUNs for 50 TB of backup data (20 7+1, 8 6+1) and four RAID-10 LUNs for meta data (4+4), with a total of 50TB of usable capacity. The nodes received backup streams from a Symantec Veritas NetBackup, Version 6, SP 5, backup application running on three media servers, that were connected to the nodes through a QLogic SANbox 5200 fibr channel switch, having a total of 8, 4Gbps FC. The gateways were also connected, through 8, 4Gbps FC lines, to the storage system.

The test consisted of storing a firs generation of a backup stream followed by more generations, each differing from the previous one by about 3%.

The two node cluster had a sustained throughput of 125 MB per second for a single backup stream, nearly 400 MB/Sec on 4 streams, a little over 1 GB/Sec on 24 streams and nearly 1.5 GB per second on 64 streams. The data reconstruction (restore) rate for the 64 streams was nearly 2 GB/Sec. For a single node with 64 streams the deduplication rate was around 870 MB/Sec, while the reconstruction rate was nearly 1.5 GB/Sec.

It is hard to compare throughput for different systems since they use different hardware, configuration and backup streams. Nonetheless for the sake of completeness we report here the most up to date information, cited in the literature and on the web, regarding competing systems. In [29] the authors describe features of the Data Domain deduplication system. It is reported that the system had a sustained throughput of 110 MB/Sec for a single stream, and 220 MB/Sec for 4 streams. A recent upgrade announcement to Data Domain's 690 series (the more advanced product series) refers to throughput on multiple streams and states that in its largest configuratio for a single repository the system has maximal throughput of 750 MB/Sec. The restore rate for 4 streams cited in [29] starts off at 220 MB/Sec for the firs generation and stabilizes at 140MB/Sec beyond the third generation.

An inline deduplication system which uses similarity based indexing coupled with a hash based data identificatio scheme is presented in [16]. The technology described in that paper was implemented in HP's D2D2500 and D2D4000 deduplication appliances. The paper provides the following information on the performance of the larger D2D4000 appliance. For a single stream the deduplication rate was 90 MB/Sec, for 4 streams it was 120 MB/Sec. The product data sheet states that the maximal deduplication rate for the largest configuratio is at least 150 MB/Sec, presumably for multiple streams. The restore rate for a single stream is reported in [16] to be 50MB/Sec while for 4 streams the rate drops to 35 MB/Sec.

A comparison of the results suggest that the deduplication throughput of a single Protectier node is either favorably comparable or strictly better than that of competing products, despite the fact that it is the only product that performs the costly byte to byte comparison. The two node cluster that works on a single repository, nearly doubles the performance of a single node. As noted already in [29], restore throughput is just as important as deduplication throughput especially on full system restores, since it may directly affect the data outage period. The restore throughput rate is strongly related to the efficien y of the method of storing the deduplicated data. While we postpone a detailed assessment of this design issue to a future paper, we can provide a system comparison in this respect as well. In Protectier with a single node the restore throughput is about 1.6 times faster than the deduplication rate. From the information in [29] we deduce that in the Data Domain product it is about 1.3 times faster, while for mul-

tiple streams in the HP product, restore is about 3 times slower than deduplication, though the accompanying paper [16] is optimistic about the prospects of future improvements. In general, restores are faster than deduplication (when properly engineered) because restores do not require much processing beyond data decompression.

Next, we consider capacity. As noted previously, Protectier was designed from the outset to support a repository whose physical capacity is 1 PB and that is the maximal physical capacity that the product supports. Real production systems rarely require such large physical capacity and in addition such repositories may require tremendous bandwidth. There are however, single system installations with 256 TB of physical storage, so the system capacity limits may soon be tested. The largest physical capacity of a single repository for a Data Domain product is 50 TB, while for an HP D2D4000 system it is 9 TB. Rather optimistically, the accompanying paper, [16] discusses memory requirements for a 100 TB system. It is estimated there that a system which uses a Bloom filte as described in [29] would require about 12 GB of memory (consistent with our estimate of 2^{37} bits per 1 PB for the filter) For their own system, the authors estimate using 3 GB. As we have seen, our methods would require about 0.5 GB. These estimates may explain the current capacity limit for systems which use the methods of [29], but they fail to explain why the HP system is limited to 9 TB, perhaps there are other unknown bottlenecks.

4.2 Results from a customer site

We consider results from a single node which is part of a large, 9 nodes, installation residing at the data center of a major energy manufacturer. The system performs daily backups using multiple streams. The system has been running for a while and we present data which was captured during a six week window in June-July 2008. The data represents a mature and rather stationary state of the system. Figure 1 shows the factoring (deduplication ratio), which is the ratio between the amount of data backed up and the actual amount of physical space occupied by the data. Since the system is mature, we see only little fluctuation during the 2 day period of measurement. The ratio is approximately 40:1, that is, the physical space required to store the deduplicated data is only about 2.5% of the size of the data itself.

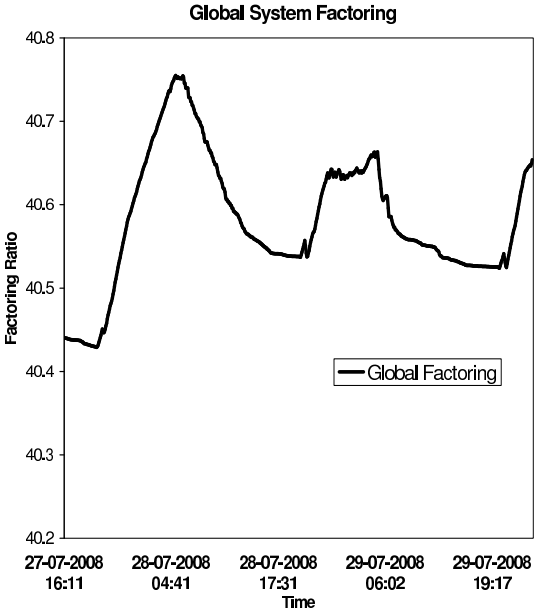

Figure 1. Factoring ratio

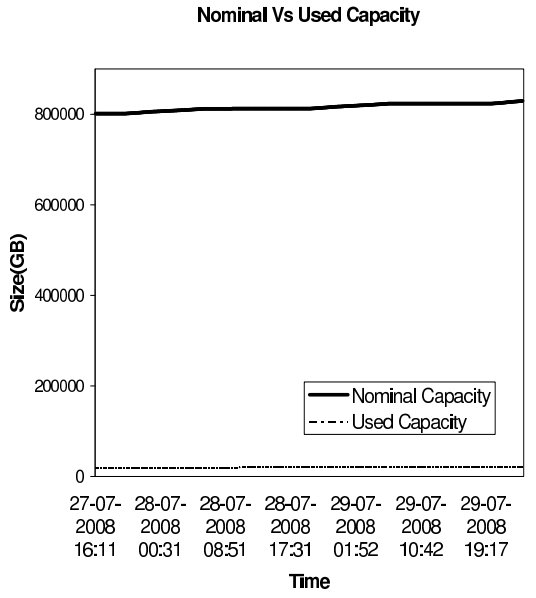

Figure 2. Storage capacity

In Figure 2, we see the same information given in absolute terms which indicate the size of the system. The data to be stored amounts to approximately 800,000GB or 0.8PB (petabytes). The actual (physical) storage space is about 20,000GB or 20TB. Figures 3 and 4 show how much data is backed up every day and at what times. Figure 3 shows the amount of data which was backed up, the amount of matching data from the repository that was found and the amount of

data actually stored which is the difference. Figure 4 tracks the backup process over a shorter 3 day period. As can be seen from Figure 4, the backup process takes place at night, as is expected since online processing which competes with backup over server and storage resources is mostly active during the workday. The peak rates for the node reach nearly 500MB per second and the backup operates with 16 streams. A rate of 500MB per seconds translates into 1.8 TB per hour, which means that the system can backup about 21 TB in a 12 hour backup window.

Over the 45 day period for which the system was tracked, the peak backup size was around 20TB which is close to the calculated backup size for a single node system over a nightly backup window. This shows that efficien y in the deduplication scheme is a very real and important issue and that some customers can really drive the system near its limits. This has also led the customer to a multiple node configuration The actual amount of storage used on the day of peak raw activity was fairly small, on the order of 2TB, however, looking at Figure 3 again, one sees that on June 16, the system used about 9TB of storage space to store around 13–14TB of raw data. Apparently this is a situation where some new data was backed up for the firs time. From some additional data from the site we know that more than 90% of signature matches extended to matching regions whose size was over a megabyte. In almost all other cases the match was very local, with a size of less than 16KB. Matches in the range of 16KB to 1MB, were almost non existent. We note, though, that this is data from a single customer.

5. Conclusions

At the end of [29], the authors make the prediction that *It will be a relatively short time before a large-scale deduplication storage system shows up with 400-800 MB/sec throughput with a modest amount of physical memory*.

In this paper we have shown that a system which achieves and even surpasses the predicted goals, the Protectier system, already existed at the time [29] was written. We have shown that the memory requirements are modest enough to easily support 1 PB of physical capacity, and explained the criteria behind our design, focusing on similarity detection techniques and the issue of decoupling the detection and comparison stages of deduplication, while keeping them synergized.

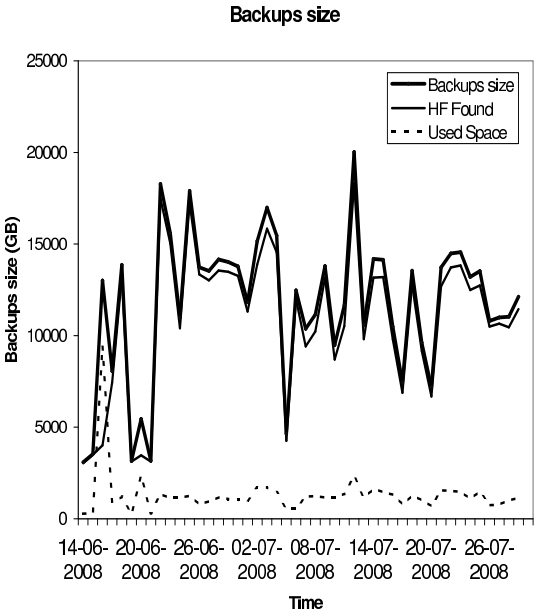

Figure 3. Size of backup on a daily basis

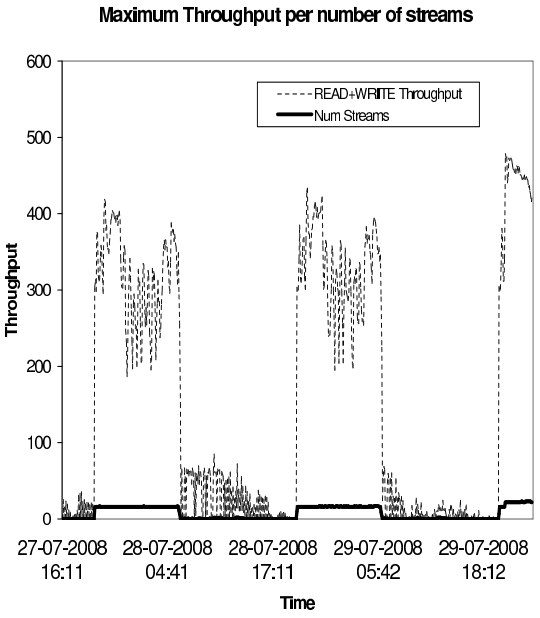

Figure 4. Backup rate over a two day period

Acknowledgments

This work was done while L.A., R.A., H.B. and M.H. were at Diligent Corp. Diligent was purchased by IBM in 2008. Coming up with a vague design for a deduplication device is very different from a finishe product which proves the underlying concepts. We thank all former and current workers of Diligent that allowed this design to materialize.

References

[1] Amir A., Lewenstein M., Porat E., Faster algorithms for string matching with k mismatches, Journal of Algorithms 50(2) (2004) 257-275.

[2] Burton H. Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. Communications of the ACM, 13 (7). 422-426.

[3] D.R. Bobbarjung, D, Jagannathan and C. Dubnicki, Improving duplicate elimination in storage systems, ACM Trans. on Storage, Vol.2(4), 424-448, 2006.

[4] Boyer R. S., Moore J. S., A fast string searching algorithm, Communications of the ACM 20 (1977) 762-772.

[5] S. Brin, J. Davis, H. Garcia-Molina. Copy Detection Mechanisms for Digital Documents (weblink). 1994, also in Proceedings of ACM SIGMOD, 1995.

[6] A.Z. Broder. Identifying and Filtering Near-Duplicate Documents. CPM 2000: 1-10

[7] A.Z. Broder. Some applications of Rabin's fingerprint ing method. In R. Capocelli, A. De Santis, and U. Vaccaro, editors, Sequences II: Methods in Communications, Security, and Computer Science, 143-152. Springer-Verlag, 1993.

[8] A.Z. Broder. On the resemblance and containment of documents. Proceedings of Compression and Complexity of Sequences 1997, 21-29. IEEE Computer Society, 1997.

[9] A.Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-Wise Independent Permutations. Proceedings of the 30th Annual ACM Symposium on Theory of Computing, 327-336, 1998.

[10] A.Z. Broder and U. Feige. Min-Wise versus Linear Independence. Proc. of the Eleventh Annual ACM-SIAM Symp. on Discrete Algorithms, 147- 154, 2000.

[11] A.Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the Web. Proceedings of the Sixth International World Wide Web Conference, pages 391-404, 1997.

[12] Fischer M. J., Paterson M. S., String matching and other products, in Complexity of Computation, R. M. Karp (editor), SIAM-AMS Proc. 7 (1974) 113-125.

[13] B. Garret and C. Bouffard, The Enterprise Strategy Group (ESG) lab validation report for IBM TS7650G ProtecTier, 2008. Available at www.diligent.com.

[14] N. Heintze. Scalable Document Fingerprinting. Proceedings of the Second USENIX Workshop on Electronic Commerce, pages 191-200, 1996.

[15] N. Jain, M. Dahlin, and R. Tewari. TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization. Proceedings of USENIX File And Storage Systems conference (FAST), 2005.

[16] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality, Proceedings of USENIX File And Storage Systems conference (FAST), 2009.

[17] Karp R., Rabin M., Efficien randomized pattern matching algorithms, IBM Journal of Research and Development 31 (1987) 249-260.

[18] Knuth D. E., Morris J. H., Pratt V. R., Fast pattern matching in strings, SIAM Journal on Computing 6 (1977) 323-350.

[19] P. Kulkarni, F. Douglis, J. D. LaVoie, J. M. Tracey: Redundancy Elimination Within Large Collections of Files. Proceedings of USENIX Annual Technical Conference, pages 59-72, 2004.

[20] Udi Manber. Finding Similar Files in A Large File System. Technical Report TR 93-33, Department of Computer Science, University of Arizona, October 1993, also in Proceedings of the USENIX Winter 1994 Technical Conference, pages 17-21. 1994.

[21] Landau G. M., Vishkin U., Fast parallel and serial approximate string matching, Journal of Algorithms 10(2) (1989) 157-169.

[22] Moulton G. H., Whitehill S. B., Hash fil system and method for use in a commonality factoring system, U.S. Pat. No. 6,704,730.

[23] Navarro G., A Guided Tour to Approximate String Matching, ACM Computing Surveys, 33(1) (2001) 31-88.

[24] S. Quinlan and S. Dorward, Venti: A New Approach to Archival Storage. In Proc. of the USENIX Conf. on File And Storage Technologies (FAST), 2002.

[25] N. Shivakumar and H. Garcia-Molina. Building a Scalable and Accurate Copy Detection Mechanism. Proceedings of the 3nd International Conference on Theory and Practice of Digital Libraries, 1996.

[26] N. Shivakumar and H. Garcia-Molina. Finding near-replicas of documents on the web. In Proc. of Workshop on Web Databases (WebDB'98), March 1998.

[27] Ukkonen E., On-line construction of suffi trees, Algorithmica 14(3) (1995) 249-260.

[28] Weiner P., Linear pattern matching algorithm, Proceedings of the 14th IEEE Symposium on Switching and Automata Theory, (1973) 1-11.

[29] B. Zhu, K. Li and H. Patterson, Avoiding the Disk Bottleneck in the Data Domain Deduplication File System, Proc. of FAST 08, the 6th USENIX Conf. on File and Storage Technologies, 279-292.

[30] J. Ziv and A. Lempel. A universal algorithm for sequential data compression, IEEE Trans. Inform. Theory, vol. IT-23, pp. 337-343, May 1977. 282