Similarity Based Deduplication with Small Data Chunks*

L. Aronovich[a], R. Asher[b], D. Harnik[b], M. Hirsch[b], S.T. Klein[c], Y. Toaff[b]

[a] *IBM, Toronto, Canada*
aronovic@ca.ibm.com

[b] *IBM – Diligent, Tel Aviv, Israel*
{ronasher,dannyh,hirschm,yairtoaff}@il.ibm.com

[c] *Department of Computer Science, Bar Ilan University, Ramat Gan, Israel*
tomi@cs.biu.ac.il

Abstract

Large backup and restore systems may have a petabyte or more data in their repository. Such systems are often compressed by means of deduplication techniques, that partition the input text into chunks and store recurring chunks only once. One of the approaches is to use hashing methods to store fingerprints for each data chunk, detecting *identical* chunks with very low probability for collisions. As alternative, it has been suggested to use *similarity* instead of identity based searches, which allows the definition of much larger chunks. This implies that the data structure needed to store the fingerprints is much smaller, so that such a system may be more scalable than systems built on the first approach.

This paper deals with an extension of the second approach to systems in which it is still preferred to use small chunks. We describe the design choices made during the development of what we call an approximate hash function, serving as the basic tool of the new suggested deduplication system and report on extensive tests performed on an variety of large input files.

Keywords: Deduplication, similarity, small data chunks, approximate hashing

*This is an extended version of a paper that has been presented at the Prague Stringology Conference (PSC'12) in 2012, and appeared in its Proceedings, 3–17.

1. Introduction and Motivation

Huge amounts of data have to be processed daily and the current trend suggests that these amounts will continue being ever-increasing in the foreseeable future. An efficient way to alleviate the problem is by using *deduplication*: large parts of the available data is copied again and again and forwarded without any change; the idea underlying a deduplication system is to locate repeated data and store only its first occurrence. Subsequent copies are replaced by pointers to the stored occurrence, which significantly reduces the storage requirements if the data is indeed repetitive [3].

Several approaches have been proposed to solve the problem, each concentrating on another aspect of the input characteristics. One of the approaches, based on hashing, can be schematically described as follows [15, 17, 13, 20].

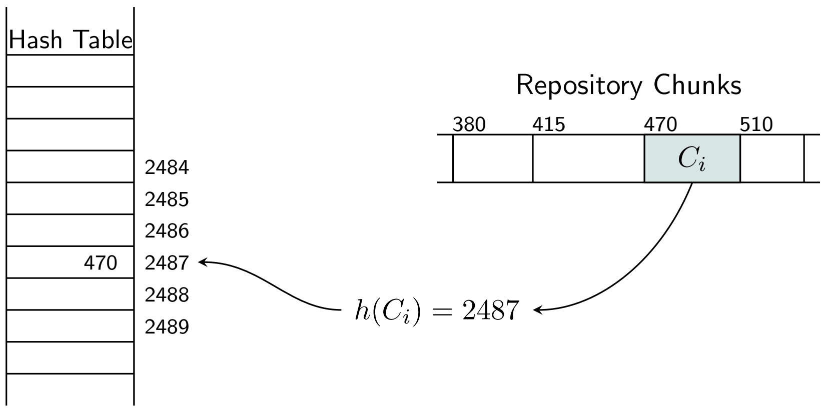

FIGURE 1: Schematical view of a hash based deduplication system.

The available data is partitioned into parts called chunks C_i. These chunks can be of fixed or variable size, and the (average) size of a chunk can be small, say 4–8KB, up to quite large, say, about 16MB. A cryptographically strong hash function h is applied to these chunks, meaning that if $h(C_i) = h(C_j)$, it can be assumed, with very low error probability, that the chunks C_i and C_j are identical. The set S of different hash values, along with pointers to the corresponding chunks, is kept in a data structure D allowing fast access and easy update, typically a hash table or a B-tree. For each new chunk to be treated, its hash value is searched for in D, and if it appears there, one may assume that the given chunk is a duplicate. It is thus not stored again, rather, it is replaced by a pointer to its earlier occurrence. If the hash value is not in D, the given chunk is considered new, so it is stored and its hash value is adjoined to the set S. Figure 1 shows the starting addresses of some consecutive chunks C_i. The hash value of the chunk starting at $ad = 470$ is $h = 2487$, so the address ad is stored at entry h in the table.

The suggested methods mainly differ in the way they define the chunk boundaries, and in the suggested size of the chunks. The chunk size may indeed have a major impact on the performance: if it is too small, the number of different chunks may be so large as to jeopardize the whole approach, because the data structure D might not fit into RAM, so the system might not be scalable. On the other hand, if the chunk size is chosen too large, the probability of getting identical chunks decreases: many instances of chunks might exist, that could have been deduplicated had the chunk size been chosen smaller, but which, for the larger chunk size, have to be kept.

A possible solution to this chunk size dilemma has been suggested in [1] and is implemented in the IBM ProtecTIER Product [9]. The main idea there is to look for similar rather than identical chunks. If such a similar chunk is located, only the difference is recorded, which is generally much smaller than a full chunk. This allows the use of much larger chunks than in identity based systems. The idea of similarity has also been exploited in [2, 18].

For many applications, such as data backups and archiving, data is more fine-grained, and much better deduplication can be performed if one can use significantly smaller chunks. A simple generalization of the ProtecTIER system in which the chunk size would be reduced from 16MB to 8K, that is, by a factor of 2000, without changing anything else in the design, would imply a 2000 fold increase of the size of the index, from 4GB to about 8TB. This cannot be assumed to fit into RAM in the near future. Moreover, keeping the definition of the notion of similarity and reducing the size of the chunks will lead to an increased number of collisions, which may invalidate the approach altogether.

Xia at al. [19] suggest to combine deduplication with delta encoding of similar chunks, assuming that chunks that are adjacent to deduplicated ones tend to be similar. The idea of the current work is to implement the required similarity by what we call an *approximate hash* scheme. This is an extension of the notion of locality-sensitive hashing introduced in [10]. The basic idea is that such an approximate hash function is not sensitive to "small" changes within the chunk, and yet behaves like other hash functions as far as the close to uniform distribution of its values is concerned. As a consequence, one can handle the set of approximate hash values as is usually done in hash applications (using a hash table, or storing the values in a B-Tree), but detect also similar, and not only identical chunks. If a given chunk undergoes a more extended, but still minor, update, its new hash value might be close to the original one, which suggests that in the case of a miss, the values stored in the vicinity of the given element in the hash table should be checked. Such vicinity searches are useless in a regular hash approach.

An approximate hash could be defined by a property that reminds the definition of a continuous function: let A and B be data chunks of fixed size, and let $d(x, y)$ be some distance function to be defined on the set of chunks; a hash function ah will be called an ε-approximate hash if

$$\exists\delta > 0 \quad d(A, B) < \delta \longrightarrow |ah(A) - ah(B)| < \varepsilon.$$

Note the difference with the common continuity definition, in which we would have $\forall\varepsilon\exists\delta$, implying that we can get function values as close as wanted (ε can tend to 0) if we start from close enough arguments. In our case, it would be exaggerated to impose such a property, and we can relax it to find two bounds δ and ε such that if the distance between chunks is bounded by the first, then the distance between the hash values of these chunks is bounded by the second, for reasonably chosen small values of ε.

Actually, even this definition could be too restrictive, and we should allow a small number of exceptions for certain extreme chunks. This leads to a probabilistic version of the above definition: a hash function ah will be called an ε-approximate hash with probability p if

$$\exists\delta > 0 \quad d(A, B) < \delta \longrightarrow Pr\left(|ah(A) - ah(B)| > \varepsilon\right) < 1 - p,$$

where the probability is taken over a uniform selection of the possible chunks A and B.

There are several possibilities to define the distance function d. A simple solution would be the Hamming distance, defined either on bits (number of 1 bits in A XOR B) or on characters (number of differing characters), but this requires the chunks to be of the same length. A more significant, yet more involved, function could be the edit distance: the minimal number of single character insert, delete and substitute operations needed to transform A into B.

The challenge is now to find such a function ah, giving a tradeoff between how well it can be adapted to reflect the approximate nature described above, and how long it takes to evaluate it. We should still bear in mind that one of the most basic requirements of a hash function is that it should not require too much CPU time.

The general algorithm for storing the repository will then be as follows. The number k of bits in the signature will be chosen in advance, and a hash table H with 2^k entries will be used as basic data structure. During the building process, each chunk C will be assigned its approximate hash value $ah(C)$, and the index, or address, of the chunk will be stored at $H[ah(C)]$, the entry in H indexed by the hash value of the chunk. If the location in the table is not free, it is overwritten. This may happen in case the new chunk is identical or very similar to a previously encountered chunk, in which case we prefer to store the address of the more recent chunk for potential later reference; but a collision may also be the result of two completely different chunks hashing to the same value, and then the pointer to the older chunk that has been overwritten will be lost.

In the next section, we describe the details leading to the design of the approximate hash function, and then report on extensive tests in Section 3, showing a noticeable improvement of the suggested method over identity based deduplication with small data chunks.

2. An approximate hashing function

Once it has been decided to base the system on approximate hashes according to the ideas stated above, the problem reduces to devising an appropriate function. This is the main thrust of the present suggestion.

Classical hashing functions have been studied for decades and many good solutions are known [7]. The major challenge in the design of an *approximate* hash function is finding the right balance between the following three competing criteria:

- Uniformity: the function should yield a distribution of values as close as possible to uniform, so as to minimize the number of collisions (false matches);
- Simplicity: the function should be easy and fast to calculate;
- Sensitivity: small changes in the chunk should not, or only slightly, affect the corresponding approximate hash value.

The first two are properties that are common to all hashing functions, the third one, sensitivity, is proper to the approximate version suggested herein. For standard hashing schemes, just the contrary is required: even very small changes in the chunk should lead to extensive changes in the hash value, otherwise the uniformity would be hurt. Some works dealing with similarity rather than identity can be found in [5, 6, 14]. Our approach is different and will be described below.

The value produced by a hash function is, in a certain sense, a summarization of the information contained in the data on which the function has been applied. This is reminiscent of similar functions, like the Fourier Transform with its many applications, or the Discrete Cosine Transform, used in JPEG image compression. Such transforms allow to recode the information of the given data into a different form, which may be more useful for certain applications, for example, being more compressible. Similarly, we would like to recode compactly much of the information contained within a given data chunk under the constraint that this recoding should be immune to small fluctuations.

This lead to the decision of using the *distribution* of the various characters that appear in the data as the basis for the suggested approximate hash. The data will be partitioned into relatively small chunks C of fixed or variable length, with (average) size of about 8–16 K. Each such chunk will be analyzed as to the distribution of the bytes forming it and their frequencies. We define the sequence of different bytes, ordered by their frequency of occurrence in the chunk, as the *c-spectrum* of C, and the corresponding sequence of frequencies as the *f-spectrum* of C. In addition, we consider also the sequence of different byte pairs, ordered by their frequency of occurrence in the chunk, and call it the *p-spectrum* of C. The suggested approximate hash function $ah(C)$ will be a combination of certain elements of these spectra. The reasoning behind the decision of relying on these distributions is that on the one hand, they usually behave like fingerprints, and it will be rare that essentially different chunks will exhibit the same distributions, but on the other hand, small perturbations in the data will often have no, or just a minor, impact on the corresponding spectra. This is the goal we wish to achieve in designing an approximate hash.

The size of the hash values will be fixed in advance, so as to exploit the space of the allocated hash table. For example, one could decide that the table will have about 4 billion entries, which corresponds to a hash value of 32 bits. A much larger hash value using $k > 32$ bits could be prohibitive, since the corresponding hash table would then have 2^k entries. On the other hand, a small value of k limits the number of elements of the spectra that can be chosen to be a part of the definition of the signature. To overcome

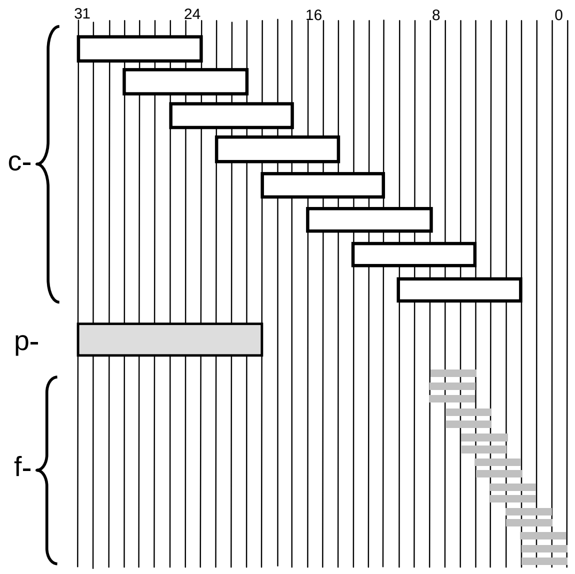

Figure 2: Schematic representation of the building blocks of a signature.

this limitation, the chosen elements of the spectra, and more precisely, only a part of their bits, will be arranged appropriately by shifting them to the desired positions, and all these bit strings will be XORed. By using different indents for the different elements, the final value will not only depend on each of the building blocks, but also on their internal order. Figure 2 is a schematic representation of a possible layout of these elements. The columns correspond to the 32 bit positions, and each rectangle stands for one of the elements, with the upper elements corresponding to the c-spectrum, the lowest elements corresponding to the f-spectrum, and the element in the middle corresponding to the p-spectrum, as detailed below. As can be seen, each bit position of the final signature is influenced by several elements.

We do not claim that the suggested layout is the best possible, not even for the sample data on which it has been tested. Rather, it is brought as an illustration of the ideas leading to its design. The specific values of the various parameters (lengths and shifts) shown in this example have been set empirically by iterating experiments to locally optimize the performance.

2.1. Using elements of the c-spectrum

Let $a_1, a_2, \ldots, a_n$ be the sequence of different bytes in the chunk, or, more precisely, the numerical value of these bytes, ordered by non-increasing frequency in the chunk. Ties are broken by sorting bytes with identical frequency by their numerical value. Let $f_1, f_2, \ldots, f_n$ be, respectively, the corresponding frequencies. The number n of different bytes in the chunk can vary between 1 (for chunks of identical bytes, like all zeroes or blanks) and $|C|$, the size of the chunk. As this size is mostly much larger than the maximum numerical value of a byte, one may assume that $1 \leq n \leq 256$.

A first attempt would be to consider each byte on its own as one of the building blocks of Figure 1, but this might result in a function that is too sensitive to noise. It will often happen that frequencies of certain bytes may be equal or very close. In such a case, a small perturbation might change the order of the bytes and yield a completely different hash value, contrarily to our goal of the approximate hash function being immune to small changes. To circumvent this problem, the a_i will be partitioned into *blocks*, gathering several bytes together and treating them symmetrically. The representation of all the elements in a block will be aligned with the same offset and will be XORed together, so that the internal order within the blocks may be arbitrary, since the XOR operation is commutative.

The sizes of the blocks should not be fixed in advance, but depend on the values themselves. Consider the sizes d_i of the *gaps* between the frequencies, $d_i = f_i - f_{i+1}$, for $i = 1, \ldots, n-1$. The boundaries between the blocks should be chosen according to the largest gaps, however, sorting according to d_i alone would strongly bias the definition of the gaps towards inducing blocks with single elements, since the largest gaps will tend to occur between the largest values. We should therefore normalize the size of the gaps by dividing by an appropriate weight. We chose harmonic weights $\frac{1}{i}$ for $i \geq 1$ according to Zipf's law [21]. The gaps are therefore sorted with respect to

$$\frac{d_i}{\frac{1}{i}} = i \times d_i = i \times (f_i - f_{i+1}),$$

which has the advantage of requiring only integer arithmetic.

For a given parameter ℓ, the $\ell - 1$ gaps with largest weights are chosen and the ℓ sets of consecutive elements delimited by the beginning of the sequence, these $\ell - 1$ gaps, and the end of the sequence, are defined as the blocks. Figure 3 is a schematic representation of an example partition into blocks with $\ell = 8$. The squares represent elements a_i, the arrows stand for weighted gaps $i\ (f_i - f_{i+1})$, and the numbers under the arrows are the indices of the weighted gaps in non-increasing order. In this example, the induced blocks would consist of 3, 1, 3, 2, 4,... bytes, respectively.

The number of elements forming the last block is limited, if necessary, to include at most a predetermined number of bytes, say 10, otherwise the

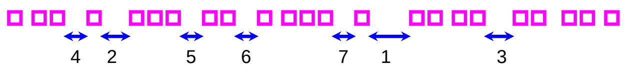

FIGURE 3: Schematic representation of the gaps.

speed of calculation could be hurt, and spurious bytes that appear possibly only once or twice in the chunk would have too strong of an influence. For the same reason, there are also lower bounds on the number of occurrences of a byte to be considered at all (for example, 15) and on the size d_i of a gap (for example, 5). If after these adjustments, the number of blocks in a given chunk is smaller than the selected value of ℓ, a different layout will be chosen that is adapted to the given number of blocks. In any case, one has to prepare layouts also for the possibility of having any number of blocks between 1 and ℓ, since certain extreme chunks may contain only a small number of different bytes.

Each block taken from the c-spectrum will be represented by a string of 8 bits, using the full representation of the corresponding bytes. The strings are depicted in Figure 1 as white rectangles. Each of these rectangles is shifted as indicated in Figure 2, where they are listed in order top down. That is, the first rectangle is shifted by 24 bits to the left (in fact, to get it left justified in the 32-bit layout), the next rectangle is shifted 21 bits, then 18, 15, 12, 9, 6 and 3 bits.

2.2. Using elements of the f-spectrum

The elements of the f-spectrum are incorporated into the signature independently from the partition into blocks of the corresponding bytes. For each frequency value, which can be an integer between 1 and $|C|$, consider first its standard binary representation (say, in 16 bits), and extend this string by m additional zeros to the right, for some predetermined small integer m. Thus for $m = 8$, we assign to each frequency f_i a 24-bit string F_i, for example, if $f_i = 5$, then F_i = 00000000 00000101 00000000. We further define D_i as the substring of F_i of length m bits, starting at the position immediately following the most significant 1-bit, for our case 00000000 000001**01 0**0000000, the bits forming D_i for $m = 3$ appear bold faced. To give another example with a value of more than 8 bits, consider $f_i = 759$; 0000001**0 11**110111 00000000 then displays both F_i and D_i. In the example of Figure 2, the elements included in the layout are the D_i, and the size m of all the elements is chosen as 3 bits. The offsets of these elements are as indicated, this time bottom up, with the largest frequency being depicted as the lowest element in the figure: 0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6 and 6.

The idea behind this choice of bits is to select those with highest variability so as to get a broad spread of values, but to ignore, for the larger frequencies, the lowest bits, which are those most influenced by small fluctuations.

2.3. Using elements of the p-spectrum

Though much of the information of a chunk is already contained in the c- and f-spectrum, we decided to adjoin also elements of the p-spectrum and got empirical evidence that this improved the performance. Further justifications for this decision is given below in Section 4. When the maximal number ℓ of blocks could be used, a single element based on the p-spectrum was sufficient. The corresponding rectangle, depicted in the center of Figure 1, is of length 12 bits and will be placed left-justified in the layout. It is defined as follows: order the pairs by non-increasing frequencies and consider those indexed 5, 6, 7, 8 and 9 in this ordering. The reason for not choosing the most frequent pairs as we did for the individual bytes is that their distribution is much more biased, with the pairs (0,0) and (255,255) appearing as the most frequent in an overwhelming majority of the cases we tested. On the other side, there was already a great variability in the pairs in positions 5 to 9.

For each of the 5 pairs, the following bitstring is constructed. Given the 2 bytes $A = a_7 \cdots a_0$ and $B = b_7 \cdots b_0$, we rotate A cyclically to the left by 3 bits, and B cyclically to the right by 3 bits. The bytes are aligned so that the rightmost 4 bits of A overlap with the leftmost 4 bits of B, and then the strings are XORed. Formally, the 12 resulting bits are now

$$a_4,a_3,a_2,a_1,a_0 \oplus b_2, a_7 \oplus b_1, a_6 \oplus b_0, a_5 \oplus b_7,b_6,b_5,b_4,b_3,$$

where the $\oplus$ operator stands for bitwise XOR. Note that the most and least significant bits of both A and B are in the overlapping part, so if their distribution is biased, they have an additional chance to correct the bias by the additional XOR. This is important for special cases, for example, when the chunk only contains printable text. The representation of all the bytes would then start with the same one to three bits, which could have a negative effect on the uniformity we seek.

2.4. Putting it all together

Finally, all the elements of the layout are XORed, yielding a 32 bit string, representing a number between 0 and $2^{32} - 1$ that will act as the hash value of the given chunk C.

The time complexity of the whole construction process is essentially linear in the size of the chunk. Most of the work is in fact the collection of the statistics; since only a constant number of highest ranked characters, frequencies or pairs is required, there is no need to fully sort the lists and the elements can be obtained by using heaps. The space complexity is constant, as it depends only on the alphabet and the number of elements used for the chosen layout, and not on the sizes of the processed chunks.

The geometry of the layout of the signature has been chosen on purpose as given in Figure 2, with the most frequent bytes being placed left-justified, thereby influencing the most significant (highest) bits, and the lowest elements of the f-spectrum appearing in the area influencing the least significant (lowest) bits. The intention was that in case of small fluctuations in the frequencies, the order of the most frequent characters might remain the same, so only some low order bits would change, yielding just a small difference in the signature values. Minor changes affecting even lower frequencies may go undetected, either because the corresponding frequencies are not among those chosen, or because the change is in the lower order bits that are not recorded in the signature.

3. Experimental Results

We performed a series of tests to assess the usefulness of the approach. A first concern was to verify that the proposed approximate hash indeed spreads its values evenly. Once this has been confirmed, we have to check that this uniformity does not come at the price of sensitivity, as it would for a standard hashing scheme. We thus checked the impact of the signature scheme in some artificial perturbation and clustering tests, described below. Finally, we bring examples of applying the whole deduplication process in comparison with an identity based approach.

As testbed, a subset of an Exchange database (EXC) of about 27GB has been chosen, as well as the entire operating system of one of our computers (OS), a file of about 5GB. The first set of tests was done with chunks of fixed length 8K. These tests were then repeated for variable length sized chunks, the boundary of a chunk being defined by applying a simple Rabin-Karp rolling hash on the d rightmost bytes of the chunk under consideration. If this hash value equals some predefined constant c, the chunk is truncated after these d bytes; otherwise, the following byte is adjoined and the test with the rolling hash is repeated. In the test, $d = 25$, $c = 2718$ and the hash function is $RK(x) = x \bmod P$, where $P = 2^{48} - 257$ is a prime number. To avoid extreme values for the chunk lengths, a lower limit of 2K and an upper limit of 64K has been imposed. The average size of a chunk was around 12K on our test databases.

	# blocks	Average	Excess	St.Dev	Excess	Avg # occ
expected		1/2		$1/\sqrt{12}$		
EXC	3,300,000	0.5050	1.0%	0.2991	3.6%	1.0033
OS	594,969	0.5085	1.7%	0.2858	-2%	1.0996

TABLE 1: Some statistics on the test databases and signatures.

3.1. Uniformity

Table 1 summarizes some statistics about the test databases, the number of 8K blocks, the average signature value (normalized to the [0,1] range), the standard deviation of these normalized values, as well as the deviation from the expected results for a uniform distribution. As can be seen, the values are very close to the expected ones. On the EXC database, the chunk containing only zeros appeared 1756 times, but beside the corresponding signature, all the others appeared mostly only once, some appeared twice, etc. No signature appeared more than 45 times. The last column of Table 1 gives the average number of occurrences for each signature.

For a more precise evaluation, inspecting each individual bit, the graph of Figure 4 shows the probability of getting a 1-bit in each of the 32 positions of the signatures. Note that these probabilities, for all bit positions, are very close to the expected value of 0.5 for a random distribution.

3.2. Perturbation tests

We now turn to observing the properties of the signature when introducing perturbations. Recall that the challenge was to reconcile two contradicting demands: on the one hand, the function is required, similarly to usual hash functions, to spread its values as much as possible, so as to minimize the number of collisions; on the other hand, we want small perturbations to yield only slight differences, if at all, in the corresponding signature values, a property one explicitly prohibits for classical hashing.

To simulate real life changes, the modified bytes did not get a random value, but rather another randomly chosen byte from within the same chunk was copied into the location to be modified. Thus the perturbations were introduced as follows: a random position i between 1 and $|C|$, the size of the chunk, was chosen, and the character from position $|C| - i + 1$ was

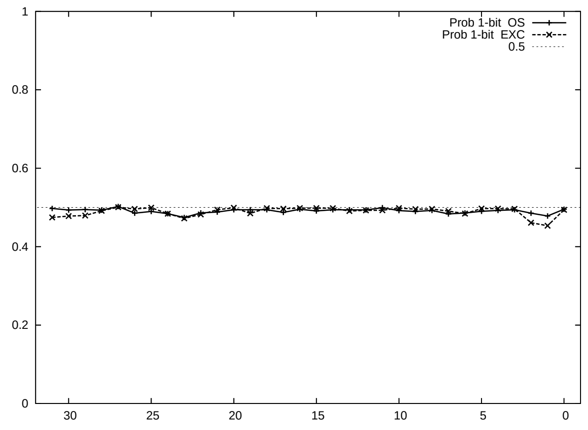

FIGURE 4: Bit distribution on example data.

copied to position i, overwriting the current one. The idea was to change the chunk slightly, but without introducing any new characters that are not already present in the chunk. Obviously, there is a small chance that this "perturbation" is in fact void (when overwriting a character by itself), but the corresponding probability is small enough so as not to bias the overall statistics. The signature function was then applied to the modified chunk and compared to the signature of the original chunk. In many cases, we got the same signature, meaning that changing a single byte in the chunk did not change the function, contrarily to what would be expected from a real hash function.

The above perturbation procedure has then been repeated, and the signature was reevaluated after $2, 3, \ldots, 10, 20, 30, \ldots, 100, 110$ byte changes. The changes were cumulative, that is, each test added one (or 10) more perturbations. Table 2 is a sample of some consecutive lines of the corresponding table.

One could define the distance between two signature values as the absolute value of their difference, reflecting the intention of the design of the signature layout to yield changes in the low order bits of the signature as result of small changes in the chunk. However, in the intended application to a deduplication system, one cannot afford too many search attempts in the vicinity of the hash value. More precisely, suppose a chunk C is given. We would evaluate $ah(C)$ and check whether there is a pointer to a chunk D at the address $H[ah(C)]$ in the hash table. If so and indeed $D = C$, the newly arrived chunk C can be deduplicated by pointing to D. But if $D \neq C$,

signature	1	2	3	4	5	6	7	8	9	10	20	30	40	50	60	70	80	90	100	110
1144762526	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	2	12	12	12	12	12	12	12	12	12
127187251	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	14	14	14	14	14	14	14	14	14	14	14	14
4244827393	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	10	10	13	13	14	13	9	6	5
1818305692	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	14	17	17	15	18	18	18	18	18	20	20
1354737651	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	8	10	10	10	10	6	5	10	12	14
33724058	**0**	**0**	6	6	6	6	6	6	6	6	6	6	2	2	8	8	6	6	6	6
1392679006	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1
59007581	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1	8	7	14	16	12	16	16	16	16
1343544922	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	11	0	7	7	7	11	0	0
1077804921	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
142372494	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	2	2	2	2	2	2	2
1076507414	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	2	2
583838910	**0**	1	2	2	2	2	2	2	2	3	3	9	9	3	0	6	6	6	6	6
2214783602	**0**	**0**	**0**	**0**	**0**	**0**	**0**	10	0	0	9	8	10	8	8	8	8	8	8	8
2217595617	1	1	1	1	1	0	0	1	1	1	0	6	3	2	2	1	1	4	3	3
2198134340	5	5	5	5	5	5	5	5	5	5	6	5	4	4	4	4	4	7	13	10
1073872964	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
3233873385	**0**	1	1	1	1	1	1	1	1	1	2	7	7	7	2	3	1	1	1	1
2155372916	**0**	**0**	8	8	8	8	8	8	8	2	0	2	2	12	12	4	2	2	2	2
4277376398	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	3	3	1	2	4	14	14	12	12	12
4264726240	**0**	**0**	**0**	**0**	**0**	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3
2248374342	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	7	7	7	13	15	16	8	8	8

TABLE 2: Hamming distance with original chunk after $1, 2, \ldots, 110$ perturbations.

the intention was to look for pointers to chunks identical to C at the neighboring locations of the hash table. But each trial is costly, so the number of trials will have to be restricted. It might possibly only be reasonable to check at $H[ah(C)]$, $H[ah(C)+1]$ and $H[ah(C)-1]$. In that case, we can as well restrict ourselves to the *Hamming distance* between signatures, i.e., the number of differing bits in their standard binary representation, rather than their arithmetic difference. These are the values displayed in Table 2.

The first column gives the original signature (as a decimal number) before applying any perturbation, then in the column headed i is the Hamming distance between the original and the new signatures when i perturbations have been applied. Note that this distance is not always an increasing function of the number of perturbations, indicating that there might be quite a few cases in which the signature tends to "correct itself" when there are many changes; however, the overall trend is clearly increasing, as can be seen in the graphs below.

The plot in Figure 5 shows the average number of changed bits as a function of the number of perturbations, for both the Exchange and the OS databases. The average Hamming distance was between 0.3 and 5 to

6. There are slight differences between the databases, but the trend is the same.

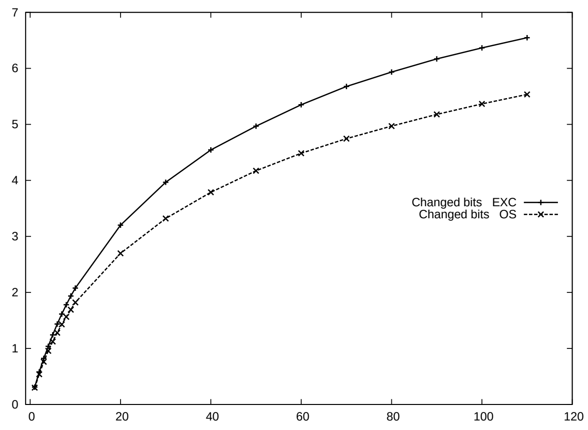

FIGURE 5: Average number of changed bits as function of the number of perturbations for the suggested signature.

Note that the distances, at least for the small number of perturbations, are quite low, and very often even zero, meaning that very small changes often yield the same signature as before. This is in sharp contrast with regular hashing schemes, for which the corresponding graph is expected to be a straight line (that is, independent of the number of changes as long as this number is > 0) at the level of about 16 (that is, about half of the bits are expected to change).

To verify this fact, we devised a control experiment in which a regular hash function (modulo $P = 2^{32} - 17 = 4,294,967,279$), was applied to the same blocks, and then performed the same perturbation tests as for our function, recording the Hamming distance between the original and perturbed signatures. Note that P is a prime number (actually the largest one fitting into our unsigned 32 bit signature). As expected, the number of changed bits was indeed around 16, as can be seen on the plot in Figure 6, more precisely, the average values were in the range from 15.988 to 16.018. Figure 6 also displays again the graph for the OS database, for comparison.

For our function, even if there are more than 100 bytes changed, this implies, at the average, only a change in about 5 to 6 bits. The resulting signature might thus be very different (depending on the position of those 6 changed bits), but the change is clearly not as radical as if a regular hash had be applied. In any case, this is just a noteworthy observation as in the

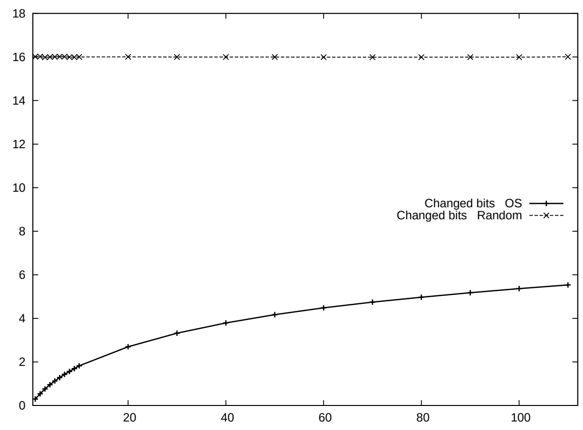

Figure 6: Average number of changed bits as function of the number of perturbations for a real hash function.

intended application, there is no intention to look for similar chunks so far away.

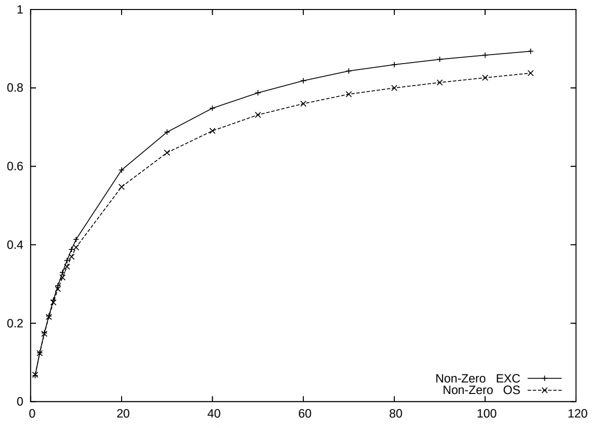

Figure 7: Probability of getting non-zero Hamming distance

To get a feeling on how far one can insert perturbations without yet changing the signature value, consider the plot in Figure 7, giving for each number i of perturbations, the probability of getting a non-zero value in the column headed i of the perturbation table. The plots are again given for both the Exchange and the OS databases. The probability of a non-zero value for a single perturbation is just about 0.06, and we see a clear ascending trend, reaching probability about 0.8 for more than 100 changes. For the control test with the real hashing function, we again got practically always non-zero values, more precisely, the probability for getting a non-zero for each of the columns was between 0.99986 and 1.00000.

3.3. Correlation with edit distance

After testing that the proposed signature indeed has the properties required from an approximate hash function, that is: it gives a uniform spread, yet preserves locality in the sense that similar blocks give similar signatures, we turn now to a less subjective measure of the closeness of two chunks in a test, which in fact checks the transpose of the above implication, that similar signatures also imply similar blocks.

The closeness of two chunks can be measured by the *Levenshtein* distance (LD), also generally known as the edit distance [16]. Given two character strings X and Y, the distance $LD(X, Y)$ is defined as the minimal number of simple operations (delete one character, insert one character or replace one character by another) needed to transform X into Y. The solution of this minimization problem is given by a well-known dynamic programming scheme requiring time and space $O(|X| \cdot |Y|)$.

To study the correlation comparing the edit distance between chunks with the difference of their signatures, the following setup has been used.

1. Generate the list of the signatures.
2. Sort the list so as to facilitate the detection of duplicates.
3. Scanning sequentially the list of differences, if two consecutive entries are found sharing the same signature, the edit distance between the corresponding chunks is recorded. Note that if a signature appears k times, for $k \geq 2$, at blocks with indices $b_1, \ldots, b_k$, this approach produces $k - 1$ edit distances, for the pairs $\{(b_i, b_{i+1}) \mid 1 \leq i < k\}$. A complementing test, trying to generate all possible $\frac{1}{2}k(k-1)$ pairs of the form $\{(b_i, b_j) \mid 1 \leq i < j \leq k\}$, has not been performed, as it would bias the values in case of many identical chunks.
4. The file D_0 of edit distances recorded in point 3. corresponds to pairs of chunks yielding the same signature. Similar files are then generated for the cases where the difference of the consecutive signatures is between 1 and 9 , 10 and 99, and between 100 and 999.
5. For each of these files, a histogram is produced, counting the number of occurrences of each of the possible edit distance values. The upper graph of Figure 8, labeled correlated blocks, displays the cumulative values of this histogram corresponding to file D_0 for the EXC database,

normalized to probabilities, that is, for entry x, the probability for a chunk pair to have the edit distance between its components to be $\leq x$. Three graphs in the lower part of the figure correspond to the files with larger distances, as indicated.

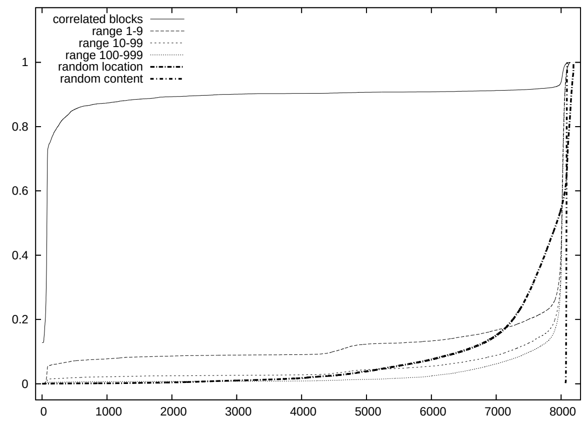

FIGURE 8: Probability of getting a given difference in the signatures as function of edit distance for the EXC database.

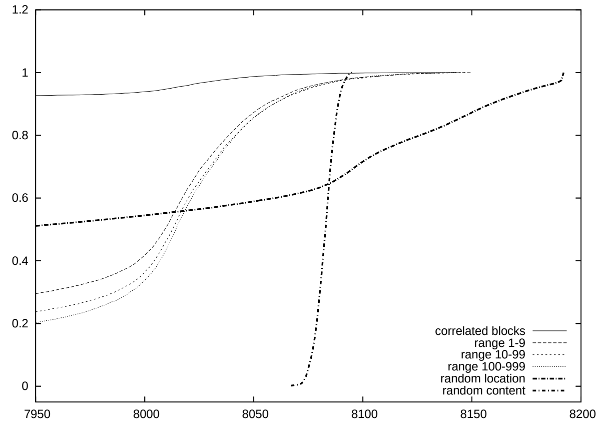

FIGURE 9: Zooming in on Figure 8.

Note that on D_0, 74% of the pairs have an edit distance of less than 100. If the chunks were not correlated, we would expect all the edit distances to be much larger. In order to compare these findings with those expected for non-correlated chunks, we devised the following simulations.

In a first test, 1000 chunks of 8K each were generated, with each character being randomly and independently chosen in the range 0-255, then the edit distance of the 999 pairs of adjacent chunks was calculated. Practically all the values were very close to the maximum 8192. The plot of the corresponding cumulative values, which are 0 up to more than 8000 and then get up to 1 within a very small range, is the almost vertical line in Figure 8 labeled random content.

To extend the comparison also to more realistic chunks, we chose, in a second test, 100,000 pairs of chunks at random positions. The values in this case were also generally very high, though also smaller distances appeared, since there were many quite similar chunks in the database, while among the randomly generated chunks no such similarities could exist. The plot of the corresponding cumulative values is labeled random location. One can see a completely different behavior of the plot, increasing first very slowly, passing the 20% threshold only after 7200. This clearly indicates that the pairs producing the first plot are not randomly selected, that is, that the fact that their distance in terms of the proposed signature function is zero, implied that their edit distance is much smaller than expected for a random choice.

The next step was then to produce similar graphs for the other files, corresponding to the ranges 1–9, 10–99 and 100–999 of the signature distance. The normalized cumulative plots, together with those displayed above, can be seen in the corresponding graphs of Figure 8. The plots for the two random files appear boldfaced at the right side of the figure.

Figure 9 is a closer look at the same graph, zooming in on the range [7950,8192], clearly showing how different from the random data the four plots for the various ranges behave. The plots reveal well how with increasing signature distance, the cumulative curves gradually become less steep and more distant from the curve of equal signature, and at the same time closer to the graph corresponding to the random choice of the blocks.

3.4. Clustering

Another test checking that similar signatures correspond to similar blocks is based on *clustering*. For each of the tested databases, N centroïd chunks have been chosen (we used $N = 11$), so that they were mutually not similar. This is achieved by choosing the chunks in a random sequence, and checking for each new candidate that it is different enough from all the preceding chosen chunks in the sequence. X and Y are said to be different enough if $LD(X, Y) \geq T$, where T is some independently chosen threshold (we used 1000), and LD is the Levenshtein distance mentioned above.

Each of the centroïds is then used to generate a number M of perturbation chunks (we used $M = 10$), which are obtained by either changing a predetermined number K of bytes of the map to a random value, or by copying to each of these K bytes the value of another, randomly chosen, byte value from within the same chunk. The number of perturbations K has been chosen to vary from 2 to 1024, doubling in each step. Finally, the approximate hashing is applied to each of the generated chunks, and the whole set of $N \cdot M$ signatures is then sent to a clustering procedure, which partitions the set of signatures, and thereby the set of corresponding chunks, into subset of similar chunks. The number of hits, that is, correctly assigned correlations between a generated chunk and its generating centroïd, is recorded as a function of the number K of perturbations.

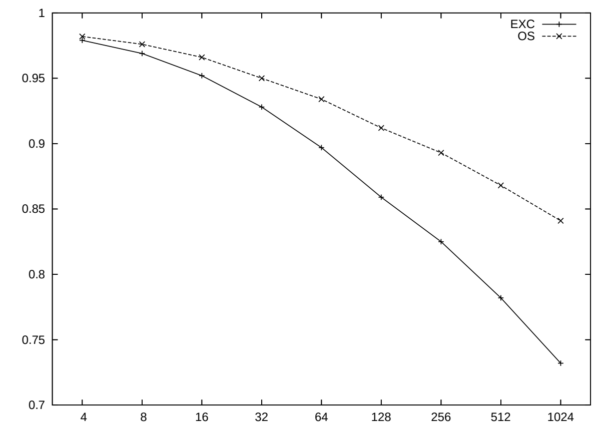

FIGURE 10: Probability of guessing the correct cluster as function of the number of perturbations.

Three different alternatives have been considered to perform the clustering: the hierarchical Tree-method (repeatedly choosing the pair of closest chunks among the set of remaining subsets and dynamically updating the sets), K-means (minimizing the within-cluster sum of squares) [12], and simply checking the distance from every generated chunk to each of the centroïds and choosing that with minimal distance. The results were similar, with the first method consistently giving slightly better performance.

Each experiment was repeated 10 times and the values averaged. The results for our test databases of the hit ratio as function of the number of perturbations are displayed in Figure 10. As can be seen, the success rate is indeed decreasing with increasing K, and for a small number of perturbations, the number of successful assignments may be as large as 95%.

3.5. Comparison of similarity with identity

As has been mentioned earlier, the ultimate aim of these hash based systems is to perform deduplication. One approach is to use standard hashing, even with cryptographically strong hash functions that reduce the probability of false alarms to almost zero, but can thereby detect only identical chunks. The alternative suggested in this paper is the *approximate hash*, which could be able of locating also similar and not necessarily identical data chunks.

It might not be possible to quantify the relative improvement caused by shifting from a system based on identity to one based on similarity: the results will be extremely data dependent, based on the nature of the data and its repetitiveness. It obviously makes no sense to simulate the system's behavior on random data, as is done for many other applications, since truly random data is not compressible. On the other hand, also compressed files cannot be compressed even further, but they may be able to take advantage of deduplication, for example when several copies of such a file appear in the database.

We therefore decided, by lack of what could be agreed on as being "typical" data, to test the performance in tests on publicly available files and report the results just as examples, without claiming that these results are representative. Indeed, on different input data, the figures could be higher or lower, depending on the data at hand.

file name	size (MB)	identity	similarity	gain
`centos-5.3-i386-server`	2816	6.58	7.10	7.3%
`freebsd-6.4-i386`	949	3.88	4.02	3.5%
`fedora-fc6-i386`	265	5.84	6.20	5.8%

Table 3: Comparing identity with similarity based systems.

distance	-5	-4	-3	-2	-1	0	1	2	3	4	5
probability	0.6	2.4	1.3	2.1	2.7	81.7	2.9	2.4	1.3	1.9	0.6

Table 4: Distribution of distances from the approximate hash value.

The files we chose were images of virtual machines obtainable from the web, a sample of which is presented in Table 3. The sizes are given in MB, and the columns headed identity and similarity list the corresponding compression ratios. The compression ratio is defined as the size of the original file divided by the size of the compressed file. For identity, we used the SHA1 secure hash function [8] and second and subsequent copies of identical chunks were removed. For similarity, we used our approximate hashing scheme, and in case a similar chunk has been found, the new chunk was delta-encoded using VCDIFF [4]. For both identity and similarity based algorithms, fixed length chunks (of size 8K) have been used, following a suggestion in [11] that fixed-sized chunks may be more appropriate for VM disk images. The final column lists the relative gain, in percent, of using similarity instead of identity.

Table 4 gives a more specific insight in the distribution of where the matching chunks have been located by our system. We checked first at $H[ah(C)]$, and if this entry did not contain a pointer to C, we also checked $H[ah(C) \pm i]$, for $i = 1, 2, \ldots, 5$. On our example data, in the overwhelming majority of cases among those where the chunk could indeed be deduplicated, the pointer was found at $H[ah(C)]$ itself. But in 18% of the cases, it was found nearby. As could be expected, the probability of locating the chunk decreases with the distance from $ah(C)$, but interestingly, the decrease is not monotonic: the values for ± 4 are larger than for ± 3. Clearly, this is due to the fact that a difference of 4 means that only one bit is different in the signature, while for a difference of 3, there are two differing bits.

4. Deciding the final layout

We chose to start the presentation of the suggested approximate hash function giving all the details of the final layout. Many of these details may seem being the result of quite arbitrary decisions, and indeed, much of our work has been empirical, as usual in many of the works on deduplication. Nevertheless, the final details were the result of an iterative process, and we report in this section some of the decisions that lead to the suggestion above.

In the early stages of the research, we experimented with various possibilities of using only elements of the c-spectrum, to which some elements of the f-spectrum have been added later. Once a satisfactory uniform distribution of the signature values had been reached, the perturbation tests showed that the function was much too sensitive to noise. This triggered several iterations of relaxing the strict uniformity requirements, which gradually also improved the sensitivity, but we were not satisfied with the overall compromise. This lead finally to the extension of the signature to include also representatives of the p-spectrum.

We thus run some tests on a 16 GB sample of the EXC database, about 2.15 million 8K chunks, to learn about the number of different pairs per chunk. The theoretical maximum of 8191 pairs has never been reached. The actual maximal value was 7855, with an average of 279.2. Since we considered using part of the pairs as building blocks for the signature, we could assume that there are enough such pairs available, as only about 0.5% of the chunks of our sample had less than 78 pairs. More interestingly, beside the about 4000 chunks consisting only of a single pair, practically all the others contained more than 50 pairs. One could thus fairly assume that any chunk, which is not all zero or blank, has a well defined ith pair, for low values of i, like 5 to 10.

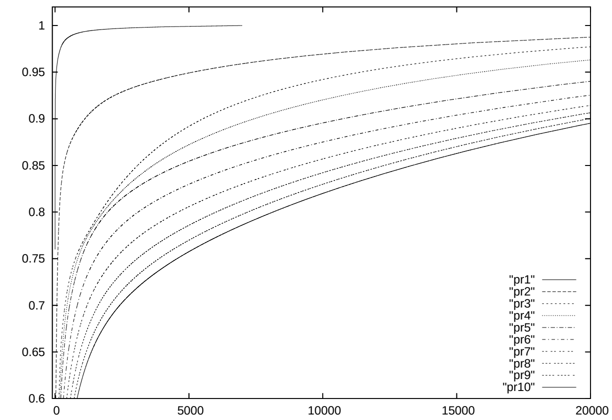

FIGURE 11: Cumulative probability of getting in position j a pair whose index in the global list of pairs is less than the value of the x-axis.

Judging from the variability point of view, it seemed at the first look that choosing the most frequent pair would yield a most variable choice, but a closer look revealed that though there are a maximum number of options for both the pair itself and for its frequency, an overwhelming percentage of these pairs are just the null-pair (0,0), which appeared in 76% of the cases as the most frequent pair! The distribution of the possible pairs are plotted in Figure 11, in which the graph pr1 shows the cumulative probability of getting in the first position a pair whose index, in the global list compiled by all the pairs in the sample sorted by frequency, is less than i, for $1 \leq i \leq 6991$; only about 7000 different pairs appear in the first position, which can be seen in the plot, where the corresponding curve already reaches the limiting value 1 for about $i = 7000$. As mentioned above, even for $i = 1$, the probability is already 0.76. The plot labeled pr$j$ displays the corresponding cumulative probabilities for position j, $1 \leq j \leq 10$, and one immediately sees the much larger range involved. Already for pr2, the probability for $i = 1$ is only 0.09 (meaning that only in 9% of the cases, the pair in second position ($j = 2$) is the pair which is the most frequent ($i = 1$) in the global list), and the limiting value 1 is only reached for the pair indexed 39738, i.e., all pairs in second position have an index smaller than that index in the global list. For pr10, the corresponding two values are 0.02 and 63302. The plots are well separated and tend to get less skewed with increasing position j. We conclude that if we want to include pairs as building blocks for the signature, it should probably not be one of the most frequent ones, but rather one (or more) in positions around 5 or up.

5. Conclusion

We have presented the main ideas leading to the design of a similarity rather than identity based deduplication system working with relatively small data chunks. Similarity has been explored earlier in this context [1], but the performance depended critically on the fact that the chunk size could be chosen large enough, in the MB range, which reduced the size of the required data structures. The current work is a first attempt to adapt this similarity approach also to systems in which a more fine grained resolution is required, with data chunks typically in the KB range.

The tests we performed suggest that the proposed approximate hash function indeed combines quite contradicting properties, like uniformity and sensitivity as required, though this can only be empirically tested on chosen examples, and not quantitatively checked in controlled statistical experiments. The scalability of the system will obviously depend on the amount of duplicate data it contains.

References

[1] Aronovich L., Asher R., Bachmat E., Bitner H., Hirsch M., Klein S.T., The Design of a Similarity Based Deduplication System, *Proc. of the SYSTOR'09 Conference*, Haifa, (2009) 1–14.

[2] Bhagwat D., Eshghi K., Long D.D.E., Lillibridge M., Extreme Binning: Scalable, parallel deduplication for chunk-based file backup. *Proc. Modeling, Analysis, and Simulation On Computer and Telecommunication Systems (MASCOTS)* (2009) 1–9.

[3] Bobbarjung D.R., Jagannathan D., Dubnicki C., Improving duplicate elimination in storage systems, *ACM Transactions on Storage*, **2**(4) (2006) 424–448.

[4] Bentley J.L., Douglas McIlroy M., Data Compression Using Long Common Strings, *Proc. Data Compression Conference*, Snowbird, Utah, (1999) 287–295.

[5] Broder A.Z., Identifying and Filtering Near-Duplicate Documents, *Proc. Combinatorial Pattern Matching Conference, CPM'00*, (2000) 1–10.

[6] Broder A.Z., On the resemblance and containment of documents, *Proc. of Compression and Complexity of Sequences*, IEEE Computer Society, (1997) 21–29.

[7] Cormen T.H., Leiserson C.E., Rivest R.L., *Introduction to Algorithms*, MIT Press, 1990.

[8] Ferguson N., Schneier B., Kohno T., *Cryptography Engineering*, John Wiley & Sons, (2010).

[9] Hirsch M., Bitner H., Aronovich L., Asher R., Bachmat E., Klein S.T., Systems and methods for efficient data searching, storage and reduction, U.S. Patent 7,523,098, Apr. 21, 2009.

[10] Indyk P., Motwani R., Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality, *Proc. of the ACM Symposium on the Theory of Computing STOC'98*, (1998) 604–613.

[11] Jin K., Miller E.L., The effectiveness of deduplication on virtual machine disk images, *Proc. of the SYSTOR'09 Conference*, Haifa, (2009) 7.

[12] Kanungo T., Mount D.M., Netanyahu N.S., Piatko C.D., Silverman R., Wu A.Y., An efficient K-means clustering algorithm: Analysis and implementation, *IEEE Trans. Pattern Analysis and Machine Intelligence* **24** (2002) 881–892.

[13] Lillibridge M., Eshghi K., Bhagwat D., Improving restore speed for backup systems that use inline chunk-based deduplication, *Proc. of the USENIX Conference on File And Storage Technologies (FAST)*, (2013) 183–198.

[14] Manber U., Finding Similar Files in A Large File System, *Proc. of the USENIX Winter 1994 Technical Conference*, (1994) 17-21.

[15] Moulton G. H., Whitehill S. B., Hash file system and method for use in a commonality factoring system, U.S. Pat. No. 6,704,730, issued March 9, 2004.

[16] Navarro G., A guided tour to approximate string matching, *ACM Computing Surveys* **33**(1) (2001) 31–88.

[17] Quinlan S., Dorward S., Venti: A New Approach to Archival Storage, *Proc. of the USENIX Conference on File And Storage Technologies (FAST)*, (2002) 89–101.

[18] Xia W., Jiang H., Feng D., Hua Y., SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput, *Proc. USENIX Annual Technical Conference*, Portland, OR (2011).

[19] Xia W., Jiang H., Feng D., Tian L., Combining Deduplication and Delta Compression to Achieve Low-Overhead Data Reduction on Backup Datasets, *Proc. Data Compression Conference (DCC)*, Snowbird, Utah (2014) 203–212.

[20] Zhu B., Li K., Patterson H., Avoiding the Disk Bottleneck in the Data Domain Deduplication File System, *Proc. of the USENIX Conference on File And Storage Technologies (FAST)*, (2008) 279-292.

[21] Zipf G.K., *The Psycho-Biology of Language*, Boston, Houghton (1935).