SimiHash: Hash-based Similarity Detection

Technical Report UCSC-SOE-11-07

Caitlin Sadowski
University of California, Santa Cruz
supertri@cs.ucsc.edu

Greg Levin
University of California, Santa Cruz
glevin@cs.ucsc.edu

February 22, 2011

1 Abstract

Most hash functions are used to separate and obscure data, so that similar data hashes to very different keys. We propose to use hash functions for the opposite purpose: to detect similarities between data.

Detecting similar files and classifying documents is a well-studied problem, but typically involves complex heuristics or $O(n^2)$ pair-wise comparisons. Using a hash function that hashes similar files to similar values, file similarity can be determined simply by comparing pre-sorted hash key values. The challenge is to find a similarity hash that minimizes false positives.

We have implemented a family of similarity hash functions with this intent. We have further enhanced their performance by storing the auxiliary data used to compute our hash keys. This data is used as a second filter after a hash key comparison indicates that two files are potentially similar. We use these tests to explore the notion of similarity.

2 Introduction

As storage capacities become larger it is increasingly difficult to organize and manage growing file systems. Identical copies or older versions of files often become separated and scattered across a directory structure. Consolidating or removing multiple versions of a file is desirable. However, deduplication technologies do not extend well to the case where files are not identical. Techniques for identifying similar files could also be useful for classification purposes and as an aid to search.

A standard technique in similarity detection is to map features of a file into some high-dimensional space, and then use distance within this space as a measure of similarity. Unfortunately, this typically involves computing the distance between all pairs of files, which leads to $O(n^2)$ similarity detection algorithms. If these file-to-vector mappings were reduced to a one-dimensional space, then the data points could be sorted in $O(n \log n)$ time, greatly increasing detection speed.

Our goal is to create a *similarity hash function*. Typically, hash functions are designed to minimize collisions (where two different inputs map to the same key value). With cryptographic hash functions, collisions should be nearly impossible, and nearly identical data should hash to very different keys. Our similarity hash function has the opposite intent: very similar files should map to very similar, or even the same, hash keys, and distance between keys should be some measure of the difference between files. Of course, "file size" is a sort of hash function on files which satisfies these requirements. However, while similar files are expected to have similar sizes, there is no expectation that two files which are close in size have similar content. It is necessary to minimize the number of such false positives in similarity hashes in order to implement a practical system. That said, it is not at all clear how to condense information from a file into a more useful one-dimensional key.

While `SimiHash` does produce integer-valued hash keys, we rely on auxiliary data to refine our similarity tests. Our key values are based on counting the occurrences of certain binary strings within a file, and combining these counts. The key values are roughly

proportional to file size, which causes false positives. However, the auxiliary data provides an easy and efficient means of refining our similarity detection. A refined version of our keys based on file extension gives a much wider spread of key values, and alleviates some of the aforementioned problems.

3 Semantics of Similarity

"Similarity" is a vague word, and can have numerous meanings in the context of computer files. We take the view that in order for two files to be similar they must share content. There are different ways to define that sharing, or what is meant by "content." Take a text file encoded in RTF format as an example. Content could refer to the entire file, just the text portion of the file (not including RTF header information), or the semantic meaning of the text portion of the file (irrespective of the actual text).

Many previous attempts at file similarity detection have focused on detecting similarity on the *text* [1, 2] level. We decided to instead use *binary* similarity as our metric. Two files are similar if only a small percentage of their raw bit patterns are different. This definition overlooks some files which we intuitively recognize as similar. For example, adding a line to source code file might shift all line numbers within the compiled code. The two source files would be detected as similar under our metric; the compiled results would not. We decided on binary similarity because we did not want to focus on one particular file type (e.g. text documents) or structure.

Another issue we do not explore in this paper is that of semantic similarity. For example, two pieces of text might use different words to convey the same thing. Or, two MP3s of the same song sampled differently may result in completely different binary content. We focus on syntactic, not semantic, similarity. In the words of Udi Manber, "we make no effort to *understand* the contents of the files." [3]

Broder [4] first made clear the distinction between *resemblance* (when two files resemble each other) and *containment* (when one file is contained inside of another). As an example of a containment relationship, take the case where one file consists of repeated copies of another smaller file. `SimiHash` is focused on resemblance detection. Two files with a significant size disparity (as in the example above) are implicitly different; containment relationships between files do not necessarily make two files similar under our metric.

In order for files to be similar under our estimation, they must contain a large number of common pieces. Another dividing point of similarity identification techniques is the granularity and coverage of these pieces. `SimiHash` operates at a very fine granularity, specifically byte or word level. We do not attempt complete coverage; we only care about the portions of the file which match our set of bit patterns.

Given some similarity metric, there needs to be a threshold which defines when files count as similar. We are focused on files which are very closely related, ideally within 1-2% of each other.

Another tradeoff for similarity detectors is whether they are meant to be used for organization or search; the focus could be either on retrieving a set of files similar to a given file (relative), or retrieving *all* pairs of similar files (absolute). `SimiHash` does both.

4 Implementation

Our hash key is based on counting the occurrences of certain binary strings within a file. The keys, along with specific intermediate data, are stored in a relational database (Figure 1). A separate program then queries the database for keys with similar values, and outputs the results. `SimiHash` was written in C++, and developed simultaneously for the Windows and Macintosh platforms.

4.1 Computing Hash Keys

Since our notion of similarity is based on binary similarity, `SimiHash` counts the occurrences of various specific binary strings within a file being processed. We preselect a set of strings, called *tags*, to search for. We only use a subset of all possible strings of a given length in the set of tags, as summing matches over all strings would blur file differences and essentially reflect file size. Tag size is important, since

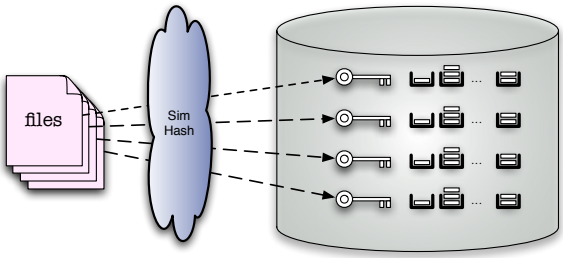

Figure 1: `SimiHash` processes files on a per-directory basis and stores the hash keys and sum table values in a relational database.

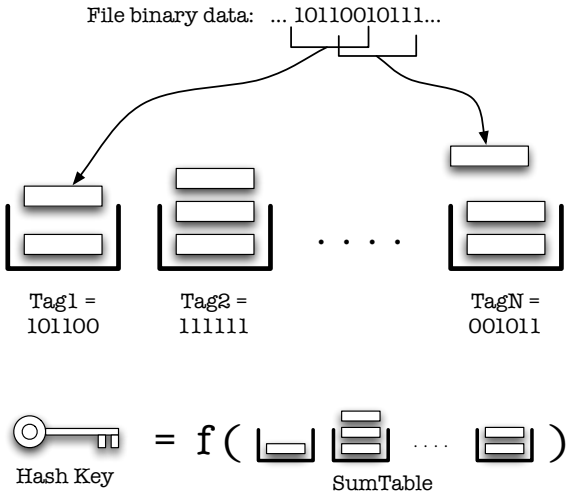

Figure 2: `SimiHash` produces two levels of file similarity data: tag counts make up the sum table entries, which are then combined to form a hash key.

shorter strings do not represent meaningful patterns and longer strings may not occur with meaningful frequency in smaller files. We chose to use 16 8-bit tags, although we experimented with several different tag sets; 8 bits strike a balance between opposing concerns.

The `SimiHash` program opens each file in a directory and scans through it linearly looking for matches with each of our tags (Figure 2). We advance the detection window one bit at a time, rather than one byte, since bit patterns across consecutive bytes encode some information about byte order. This also gives us a larger and more varied key space, which allows more refined similarity detection. When a match is found, a skip counter is set and then decremented in the following bits. This prevents overlaps of matches on the same tag (for example, `0x00` will only be detected twice, and not 9 times, when two consecutive zero bytes are scanned). A count of matches, or *hits*, is kept for each tag, and these are stored in a *sum table*. The hash key is then computed as a function of the sum table entries. We restrict our attention to linear combinations of the sums, as giving an order of magnitude more weight to one tag over another would be basically equivalent to not using the other tag at all. We experimented with various weighting schemes for our linear combinations.

Figures 3 shows key values plotted against file sizes. This figure shows the key spaces for keys with identical weights (Uniform) and heavily imbalanced weights (Skew), and plots the line $y = x$ for reference. Note that both key values are roughly proportional to file size, although the Skew Key has a wider variance.

For each file, the file name, path, and size, along with the `SimiHash` key and all entries in the sum table, are stored in a MySQL database. Our implementation has the capacity to compute and store multiple keys per field, so that different sum table weightings can be compared side-by-side or used as additional filters.

4.2 Extensions

We also implemented a variation of the key function which accounts for file extensions. It is not unreasonable to claim that two files are inherently different if they have different extensions; our extension-aware hashing scheme assigns very different values to any two files with different extensions. To this end, the extension-aware `SimiHash` computes a simple hash of the first three characters of a file's extension, with a value between 0 and 1. The distribution of these values is fairly uniform across the space of possible extensions. We then multiply this extension value by `MAX_INT`, and add to our `SimiHash` key value. Since

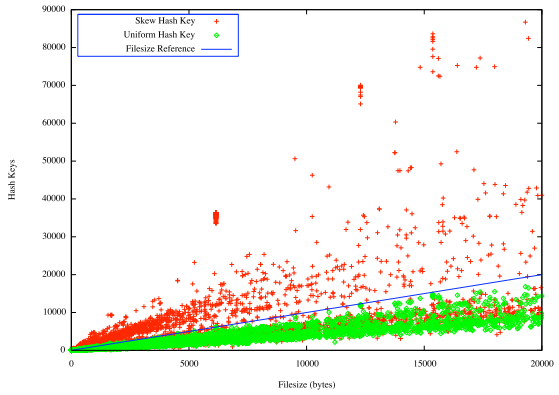

Figure 3: Visualization of Key Spaces for Uniform and Skew Keys

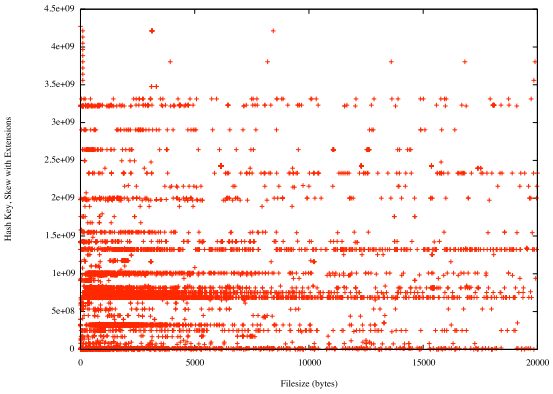

Figure 4: Visualization of Skew Key Space with File Extension Modification

we only care about the *distance* between keys, and not their actual values, this mapping will not affect the relation between files with the same extension but will tend to widely separate files of different extensions. This has the effect of more evenly distributing our key values across the space of 32-bit integers, and making cross-extension key matches very unlikely.

Figure 4 shows the Skew Key with file extension modification. Keys span the full range of 32-bit values, with horizontal stripes representing different file extensions. It is clear from the picture that key hits between different file types would be highly unusual. For further discussion of these keys, see Section 5.

4.3 Finding Similarities

Once the database has been populated, we use a second program called `SimiFind` to look for similar keys (Figure 5). `SimiFind` performs a SQL query on each key to find all other key values within a certain threshold, one file at a time. `SimiFind` has a single tolerance level across all files; this tolerance level is multiplied by the size of the target file when creating the threshold since we expect key values to increase proportionally to file size. The threshold-matches are the first pass from the similarity filter. `SimiFind` then discards any potential matches where the file size differs too much from the target file. Next, `SimiFind` computes the distance between the sum tables for each potential match and the target file, which is just the sum of the absolute values of the differences between their entries. If this distance is within a certain tolerance (proportional to the key tolerance and file sizes), then we report the two files as similar. Further, if the sum table distance is zero, the two files are optionally compared directly to determine if they are, in fact, identical files.

Essentially, two files are deemed similar if they each contain a very similar number of our selected bitwise tags. This method has several strengths and drawbacks. Because the ordering of the tag matches within a file is not accounted for, rearranging the contents of a file will, up to a point, have little impact on key values and sum tables. Similarly, adding or removing small pieces of a file will have only a small effect on the hash key. Consequently, small changes to a file shouldn't throw off our similarity measure. Further, the results of our calculations are relatively easy to understand and reason about. Because "similarity" is based on the numerical distance between values, we can easily change the tolerance level for key and sum table distance matches. Of course, increasing tolerance values both widens the range of similar files found and increases the number of false positives, so a good balance between these must be found.

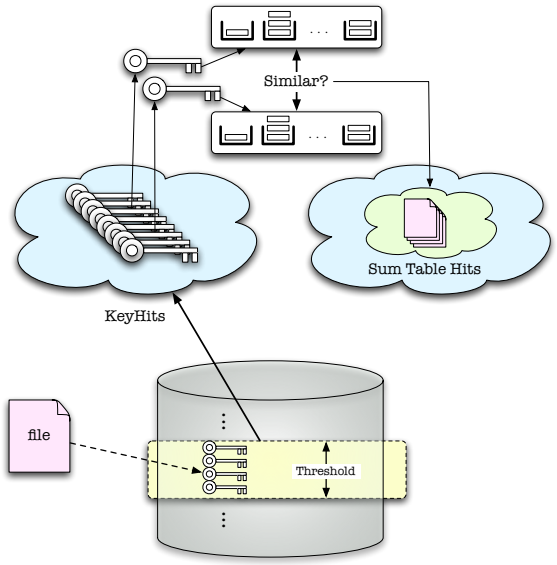

Figure 5: `SimiFind` identifies all files which have key values within a certain tolerance of a particular file, then performs pairwise comparisons among the sum table entries to return a filtered selection of similar files.

As the order of strings within files is not measured, very different files can be detected as similar if they happen to share too many bit patterns. For example, the law of large numbers makes false positives within the key space more likely for large files with effectively random data. Since key similarity comparisons are based on $O(\log n)$ searches through a sorted list, an excess of false key-similar positives means a larger number of pair-wise comparisons that must be performed on the sum table.

5 Results

We explored how different selections of tags and tag weights affected our output. We also investigated factoring in other data, such as the file extension, when computing hash keys. We ran tests on a variety of data sets, both artificially generated and found in the wild. There are still many settings and improvements to explore in the future.

We found that an unbalanced weighting scheme where only a few tags were used to compute a key worked best on a realistic file set, although a more uniform weighting scheme performed better on contrived data sets. We also identified several problems with our method which highlight areas for future work.

5.1 Choosing Tags and Keys

As previously discussed, we settled on 16 8-bit tags to apply our similarity measure. In trying to select tags, we noticed that `0x00` was often the most significant single differentiator of file structure. This is not surprising, as some files are padded with large sections of zeros, while data rich files and text files contain few or no `0x00` bytes. Other than `0x00`, an essentially random selection of bytes values seem to perform generally better than contrived or structured tag sets. We included the ASCII representations of 'e' and 't', as these appeared to be strong indicators of text-based data, and also non-ASCII-range bytes, which would be less prevalent in text files.

One measure of key performance is the ratio of sum table hits to key hits. That is, what fraction of key hits are validated as actually similar according to the sum table? The higher this ratio, the lower the presumed false positive rate of the key. We use this measure to compare different modifications.

Figure 6 shows results of comparing four keys on two distinct file sets. The Uniform Key has all 16 tags weighted equally, while the Skew Key applies uneven weights to only 4 of the tags, with the `0x00` tag getting the largest weight (other weighting schemes were tested, but were found to regularly lie between these two extremes). The "Similar Files" set contains a sequence of 4K files, each of which differs in only a few bytes from its predecessor. Consequently, "nearby" files should all be considered similar. The "Different Files" set contains a sample of assorted files from a real file system. From this set, files with apparent similarities were removed, and all files were truncated to 4K to make results more easily comparable. That we observe a smaller rate of false positives on a set of similar files is not surprising, since there are simply fewer dissimilar files. For the somewhat more realis-

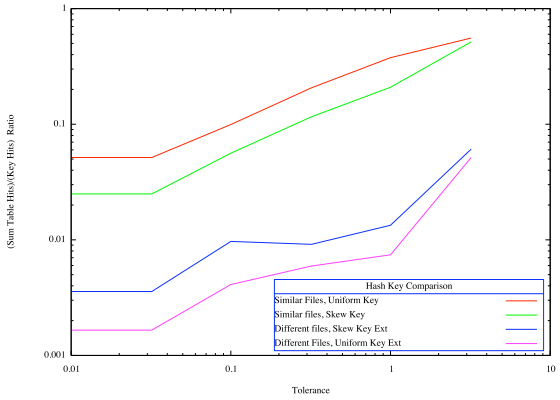

Figure 6: Comparing effectiveness of different key strategies on sets of predominantly similar and predominately different files.

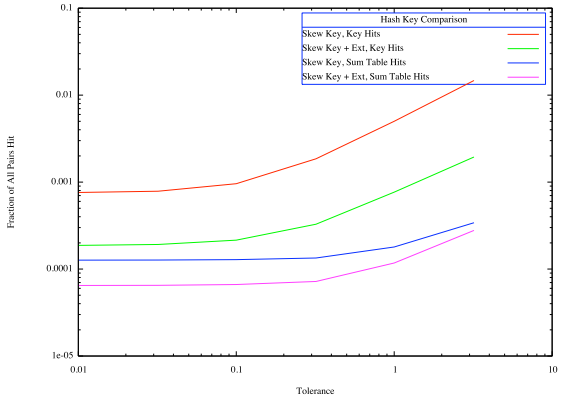

Figure 7: Number of key and sum table hits with and without the file extension key modification. Results are scaled by the total number of file pairs in the file set.

tic set of different files, the Skew Key shows better performance[1]. Running tests on several other data sets confirmed this observation and so the Skew Key was adopted as the standard key for future tests.

We also investigated various relative weightings of tags in the calculation of the hash key. We tried equal and unequal weightings of all tags, as well as giving zero weights to (i.e. ignoring) a large fraction of our tags. On the whole, this last scheme performed best on real file sets.

The next trial demonstrated the value of the file extension key modification. Figure 7 shows key hits and sum table hits for the Skew Key, with and without the file extension modification. The trial was run on 10,364 files from a random selection of directories on an existing Macintosh file system. To normalize the results, values were divided by $\binom{n}{2}$, the total number of pairs of files across the test set. Even at a rather high tolerance, the Skew Key with extensions only reports a match for about one in 5000 pairs, or about two hits per file. In other words, only a small number of auxiliary sum table comparisons are needed on average. While a file set of 10000 is small compared to an entire file system, the small percentage of matches is an encouraging sign that our methods have some hope of scaling well to larger file sets.

5.2 Problems

There are a number of problems with our method, although it is not clear if any generalized similarity measure could address all of these. One problem we encountered was certain file formats with very large headers. Postscript is an example of this. We took 200 randomly generated 4K ASCII files (which should have no more similarity than would be predicted by the averaging effects of the law of large numbers), and then converted them to both PDF and Postscript. The results of applying the Skew Key to these three sets is shown in Figure 8. In all three formats, there is a critical tolerance region, where we quickly jump from "none similar" to "all similar." While the shape of the curves is nearly identical, this threshold appears for Postscript files at a tolerance value nearly 100 times more sensitive than their raw ASCII coun-

[1]Note that these are ratios, not absolute counts of similarity hits. While some effort was made to remove similar files from the "Different Files" set, the remaining hits that *were* found were all between files with matching extensions, even without the file extension enhancement in place. Further, matches appeared to occur on genuinely similar files, with similarity decreasing as tolerance increased.

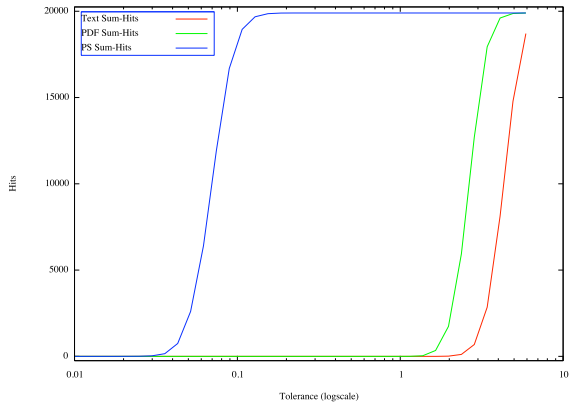

Figure 8: Sum table hits for 200 randomly generated ASCII files, with data converted to PDF and Postscript formats. Large file headers in Postscript cause the same content to be identified as similar at much lower tolerance levels.

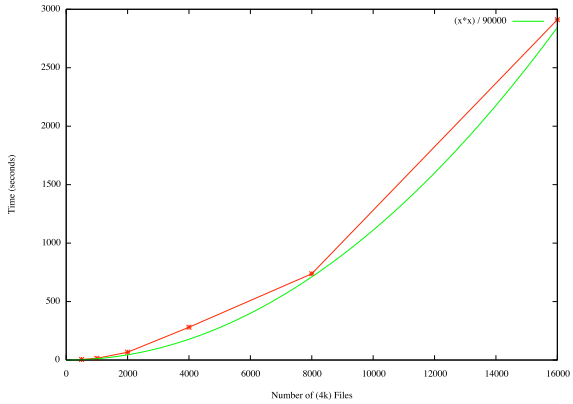

Figure 9: Run time versus file count for `SimiFind` on sets of similar files, with a quadratic curve plotted for reference.

terparts. This is due to common headers and file padding in the more complex file formats. For example, 4K of raw text translates into a 12K PDF, and an enormous 252K Postscript file. Obviously, the majority of the file is structure and not content, and so two Postscript files with entirely different data will still show a very high similarity as a percentage of their size. This makes detecting differences in data problematic for such file formats. A potential solution would be to manually set the tolerance levels for different file formats. A very low tolerance setting for Postscript extensions would rule out similarity that amounts to only a small percentage of the file size.

Another problem we haven't addressed here is scale. So far, we have only used a maximum test set of 10,000 files. As similarity detection would be useful in file systems of potentially millions of files, it is not immediately obvious how well our results scale. In terms of space usage, only a few hundred bytes were necessary to store the `SimiHash` information associated with a particular file. In our implementation, most of this was taken up with the file path string for increased readability. In a more integrated implementation, that text could be replaced with the file inode address or other numerical representation (such as a hash of the path), reducing the space usage to approximately 100 bytes per file.

We ran our `SimFind` process on sets of similar files that ranged in count from 500 to 8000, and timed the results, shown in Figure 9. A quadratic curve is plotted for reference, and our time growth appears to be $O(n^2)$. This is exactly the sort of prohibitive growth that a hash key method has potential to avoid, although we must keep in mind that these are sets of very similar files, where numerous key hits are expected. It should only take $O(\log n)$ time to determine the *number* of key matches against a file, but if the number of matches is proportional to the total number of files, then we will need to perform $O(n)$ sum table comparisons for each file. $O(n^2)$ growth may be unavoidable. Within a typical file system, it may be the case that the number of files that actually generate key hits against a single file is a tiny fraction of the overall file count; this fraction may be small enough to make the constant on the $O(n^2)$ of a manageable size, so that our method would be still be efficient in practice. In any case, we did not expect key hits by themselves to provide a reliable measure of fine-grained similarity. Instead, we expected they would provide a strong filter which greatly reduces the number of pair-wise comparisons subsequently needed.

A final difficulty is the relation between key and sum table hits. Sum table hits represent a much more accurate measure of similarity. Ideally, if sum table comparisons would label two files as similar, then those two files should also first pass the coarser key similarity test. In practice this ideal may not be desirable. Consider the Uniform Key, where all tags are given equal weights. If two files have a sum table distance of 20, then the differences in their keys would lie somewhere between 0 and 20. If our sum table tolerance is 20, then our key hit tolerance should be 20 as well to allow all possible sum table hits to be checked[2].

In general, it may be that a key tolerance of 15 only eliminates 10% of our sum table hits, but reduces the number of false positives by 40%. This phenomenon is visible in Figure 7, where the improved key with extensions admits fewer sum table hits. This trade-off between performance and accuracy may be acceptable or even desirable. Exploring the interaction of these two tolerances remains a topic for future work; their actual values could be set based on the needs of a given implementation environment.

6 Related Work

The problem of identifying file similarity is not a new one, although no one seems to have discovered a consistently good general solution. Much of the research on similarity detection has focused on very specific applications and file types. This includes:

- technical documentation [1]
- software systems [5]
- plagiarism detection [2, 6]
- music [7]
- web pages [8, 9]

For example, Manku et al. applied a so-called `simhash`[3] algorithm [10] to detect similar web documents in a large repository [9]. Their strategy works at the document level and counts instances of hashes instead of bit patterns.

There has also been a body of research focusing on redundancy elimination or deduplication. In most cases, the main goal of redundancy elimination is a reduction in either bandwidth or storage. Redundancy elimination can focus on eliminating multiple copies of the same file, or else preventing repeats of specific blocks shared between files. The standard way to identify duplicate blocks is by hashing each block. Venti [11] is an archival storage system which stores only one copy of every block. As files are modified, new copies of modified blocks are written to disk, without changing references to unmodified blocks. Shared or unmodified blocks are identified by comparing hashes of the blocks within a file before writing to disk. LBFS [12] exemplifies a similar idea but is focused on bandwidth reduction; when a file is changed, only modified blocks are sent over the network. Redundancy elimination at the block (or chunk) level provides a coarse-grained method of file similarity; files which share enough identical blocks are similar. Forman et al[1] took this approach when identifying similar documents in a repository. A file is represented by a collection of hashes of content-based chunks. If the collections of a set of files share a large overlap, then the files are considered similar.

A natural question when classifying blocks is how to identify block boundaries. The options for this include fixed-size chunking (for example, file system blocks), fixed-size chunking over a sliding window (rsync [13]), or some form of dynamic content-based chunking [12]. Content-defined chunking consistently outperforms fixed-sized chunking at identifying redundancies, but involves larger time and space overheads [14].

Instead of coalescing repeated blocks, delta-encoding works at a finer granularity. Essentially, it uses the difference (or delta) between two files

[2]Only in an extreme case is this high a key tolerance necessary. We compared sum table hits to key hits in a data set of similar files made by successively appending bytes to a starter file. In this worst case scenario, the sum hit to key hit ratio approaches one.

[3]Our algorithm was originally called `SimHash`, and was developed concurrently with the previous use of this name [9]. However, we changed it to `SimiHash` in this technical report to avoid confusion.

to represent the second one. This is only effective when the two files resemble each other closely. Different versions in a version control system provide a good example. DERD [15] investigates dynamically identifying similar files (or web pages) and then using delta-encoding to shrink the total footprint of similar pairs. The goal of REBL [16] is heavy redundancy elimination with a combination of previous techniques. Identical and similar (content-based) chunks are identified. Identical copies of chunks are removed, and similar chunks are delta-encoded. The remaining chunks are compressed; this is essentially redundancy elimination within one file.

Udi Manber [3] developed a technique for finding what he calls *approximate fingerprints* of a file. His approach involves the concept of a set of *anchors*, or small patterns of characters. In each file, a checksum of the characters around occurrences of each anchor is computed. This set of checksums forms the fingerprints for the file, and is compared against fingerprints for other files when searching for similar files. Our sum table counts are in a similar vein. An important distinction between the two ideas is that we also form one hash key for each file. This serves as a low-cost, first-pass filter which greatly reduces the number of pair-wise comparisons needed.

7 Future Work

There are many ways to extend and build on the ideas and methods presented here, some of which were discussed in earlier sections. Our hash function could almost certainly be improved upon. Even the file extension modified hash appears to cover only a small portion of our 32-bit key space. A hash function which covered this space more uniformly while still encoding similarity could provide a much lower rate of false positive key hits. We could also combine multiple hash keys in a progressive filtering scheme to remove more false positives before getting to the level of sum table comparisons. Even the sum table pair-wise comparison scheme could be used as an intermediate filter for more complex pair-wise comparison schemes. Since computing sum table differences is fast and efficient, it might provide an efficiency boost to other methods. Alternately, we could combine our similarity measurements with others.

There are a few other metrics besides binary similarity which would extend to the diversity of file types available in a system. One specific example is metadata about files. We did not use file metadata (author, creation date, etc.) for file similarity detection because we wanted our code to be cross-platform compatible. We also did not want to get caught up with issues of metadata reliability, and so limited our focus to the actual contents of the file. In future, one could imagine adding heuristics for metadata similarity.

Multiple similarity metrics for different file types could be combined to have a cohesive file similarity measure for the entire system. Extending `SimiHash` to provide varying built-in tolerance levels for different extensions may alleviate this problem somewhat, but there will still be some file types for which binary similarity does not work (e.g. music files). As an alternative, `SimiHash` could be made more effective by running certain files through a filter before computing their hash values. For example, many different document formats are often essentially just text (the aforementioned Postscript, PDF, and RTF, as well as MS Word documents, HTML files, etc). We could extract the text content from such files and hash only the raw ASCII.

Existing similarity detection methods based on similar file chunks could be extended and enhanced with similarity hashes. Using the LBFS [12] scheme for dynamically partitioning files into chunks, we could restrict our attention to a similarity hash on file chunks. One of the main drawbacks of our approach is the strong correlation between our keys and file sizes. With a small upper bound on chunk sizes and data *quantity* not playing such a role, we could implement more diverse schemes for similarity hashing.

8 Conclusions

We developed a similarity hash function, called `SimiHash`, which stores a set of hash keys and auxiliary data per file, to be used in determining file sim-

ilarity. We define similarity on a binary level, and experimented with variations. We discovered some fundamental limitations to a general-purpose similarity hash function for files, and directions for future work.

9 Acknowledgments

Thanks to Ian Pye for help with getting MySQL up and running, and teaching us Python for all our scripting needs. Thanks to Darrell Long for inspiring us to work on this project.

10 Code

Source code for `SimiHash` is available under GPL v2 at: `http://code.google.com/p/simhash/`.

References

[1] G. Forman, K. Eshghi, and S. Chiocchetti. Finding similar files in large document repositories. *International Conference on Knowledge Discovery in Data Mining (KDD)*, 2005.

[2] T.C. Hoad and J. Zobel. Methods for identifying versioned and plagiarized documents. *Journal of the American Society for Information Science and Technology*, 54(3):203–215, 2003.

[3] U. Manber. Finding similar files in a large file system. *USENIX Annual Technical Conference (ATEC)*, 1994.

[4] A. Broder. On the resemblance and containment of documents. *Conference on the Compression and Complexity of Sequences*, 1997.

[5] T. Yamamoto, M. Matsusita, T. Kamiya, and K. Inoue. Measuring Similarity of Large Software Systems Based on Source Code Correspondence. *International Conference on Product Focused Software Process Improvement (PROFES)*, 2005.

[6] Y. Bernstein and J. Zobel. A scalable system for identifying co-derivative documents. *Lecture notes in computer science*, 3246:1–11, 2004.

[7] Matt Welsh, Nikita Borisov, Jason Hill, Robert von Behren, and Alec Woo. Querying large collections of music for similarity. Technical Report UCB/CSD-00-1096, EECS Department, University of California, Berkeley, 2000.

[8] D. Buttler. A Short Survey of Document Structure Similarity Algorithms. *International Conference on Internet Computing (ICOMP)*, 2004.

[9] G.S. Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. In *International conference on World Wide Web (WWW)*, 2007.

[10] M.S. Charikar. Similarity estimation techniques from rounding algorithms. In *Symposium on Theory of Computing (STOC)*, 2002.

[11] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. *Conference on File and Storage Technologies (FAST)*, 2002.

[12] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Symposium on Operating Systems Principles (SOSP)*, 2001.

[13] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Department of Computer Science, The Australian National University, 1996.

[14] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. *SENIX Annual Technical Conference (ATEC)*, 2004.

[15] F. Douglis and A. Iyengar. Application-specific delta encoding via resemblance detection, 2003.

[16] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *Usenix Annual Technical Conference (ATEC)*, Berkeley, CA, USA, 2004.