Efficient Detection of Large Scale Redundancy in Enterprise File Systems

George Forman, Kave Eshghi, Jaap Suermondt

HP Laboratories
HPL-2008-30R2

Keyword(s):
data mining, min-hashing, set sketches, directory similarity and deduplication, file systems, scalability, storage management.

Abstract:
In order to catch and reduce waste in the exponential demand for disk storage, we have developed a technology based on set sketches that enables enterprise storage managers to efficiently detect approximate duplication of large directory hierarchies, e.g. unnecessary mirroring by uncoordinated employees or departments. Identifying these duplicate or near duplicate hierarchies allows appropriate action to be taken at a high level, e.g. coordinate and consolidate multiple copies in one location.

External Posting Date: December 18, 2008 [Fulltext] Approved for External Publication
Internal Posting Date: December 18, 2008 [Fulltext]

To be published in Operating Systems Review, journal, January 2009, vol.31 (1)

Efficient Detection of Large-Scale Redundancy in Enterprise File Systems

George Forman
Hewlett-Packard Labs
Palo Alto, CA, USA
+1-650-857-1501
ghforman@hpl.hp.com

Kave Eshghi
Hewlett-Packard Labs
Palo Alto, CA, USA
+1-650-857-1501
kave.eshghi@hp.com

Jaap Suermondt
Hewlett-Packard Labs
Palo Alto, CA, USA
+1-650-857-1501
jaap.suermondt@hp.com

ABSTRACT
In order to catch and reduce waste in the exponentially increasing demand for disk storage, we have developed very efficient technology to detect approximate duplication of large directory hierarchies. Such duplication can be caused, for example, by unnecessary mirroring of repositories by uncoordinated employees or departments. Identifying these duplicate or near-duplicate hierarchies allows appropriate action to be taken at a high level. For example, one could coordinate and consolidate multiple copies in one location.

Categories and Subject Descriptors
H.4 [**Information Systems Applications**]: decision support. I.5 [**Pattern Recognition**]: Design Methodology.

General Terms
Algorithms, Measurement, Performance.

Keywords
data mining, min-hashing, set sketches, directory similarity and de-duplication, file systems, scalability, storage management.

1. INTRODUCTION
Driven by issues of cost, risk, and lost productivity, centralized information management has come to the forefront of CIO concerns. Despite continually falling prices and rapidly growing capacity of individual disks, the number one pain point reported by IT storage professionals at large enterprises is the management of storage growth, according to repeated annual surveys [6]. This is partly because the cost of storing and managing files comprises much more than the cost of the media—it includes office and data center real estate, headcount for system management, power, cooling, the network impact of moving data, backup, information lifecycle management (archiving, retention and deletion policies, etc.) and the rapidly growing costs of e-discovery for litigation, combined with substantial legal and financial risk for sloppiness.

In this paper we restrict our attention to file system data, i.e., we exclude databases and backup storage. File system data, because of its unstructured, ad-hoc nature, can be most problematic to manage. This so-called unstructured information (file storage on managed desktops and network attached storage) is the largest and fastest-growing segment of storage, estimated to be at 80% of all enterprise bytes [5].

A good portion of data growth is duplicated data, rather than new file generation by employees. While not all duplication is pure waste (there could be intentional duplication for data protection or caching), much of it is. For example, several employees on a timesharing system may individually download, unpack and build their own private version of the latest Java J2EE JDK (~400 MB), before the overworked system administrator gets around to making a shared copy available to everybody. And when they do get around to it, the other copies do not naturally get garbage collected or even detected. Such wasteful duplication is rampant in many different forms. The greatest duplication occurs when whole directories are duplicated, rather than individual files. In the example above, no individual duplicated file may be particularly large, but the aggregate duplication exceeds the largest individual files. Also note that detecting duplication at the whole-directory level yields much larger "units of discovery" than the largest individually duplicated files, making it much less laborious to review the findings.

To remove this waste, the first step is to have an approximate picture of its extent and of where the duplication occurs from a global perspective. Such assessments are valuable in the IT industry because they help map problem areas and determine the cost-effectiveness of potential solutions—such as shared file systems, content-addressable archive storage, tools for individual users to manage their own storage and backup quotas, etc.

This opens up an interesting and high-value assessment problem for storage administrators and industry consultants: for an enterprise that has petabytes of distributed file storage, can we easily and cheaply identify the degree of large-scale duplication in distributed file systems, and identify the biggest contributors to the duplication?

We have developed a set of algorithms and protocols that perform duplication assessment on very large, distributed file systems with minimal impact on the systems being analyzed, while providing the level of accuracy that is required for decision making. We have also built a user interface that allows effective navigation of the file space, highlighting consolidation opportunities.

It is essential that such a tool should have as little performance impact as possible on the enterprise file systems. To enable assessments in most existing systems without bringing enterprises to their knees, we need to do so in the least intrusive way possible—i.e., without forcing the installation of agents on all assessable systems (a system management nightmare), without creating a centralized infrastructure that indexes all files, without having to read all files, and with minimal bandwidth requirements.

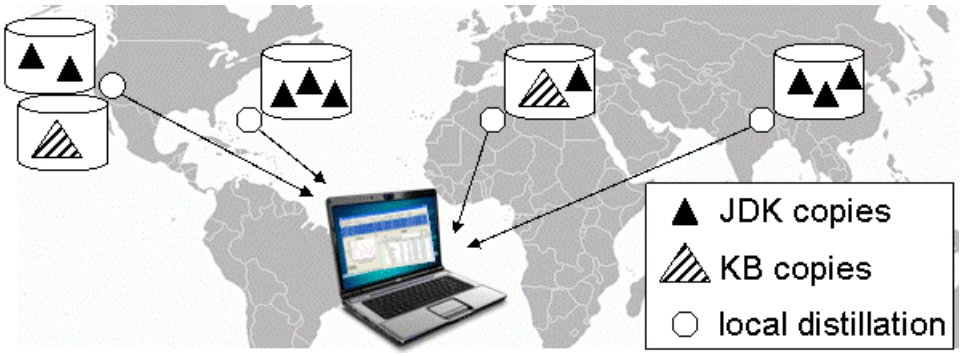

Figure 1. Duplication Assessment in Globally Distributed Enterprise

One approach to the duplicated file problem is to use a Single Instance Storage (SIS) system, as Windows SIS [1] or the research effort Farsite [3]. There are two primary problems with this approach. The first problem is that most single instance file systems are local; they are based on centralized servers that do not scale well to globally distributed computing infrastructures. For example, the Windows Single Instance Storage system does not eliminate duplicates across multiple file systems. The second problem is that the vast majority of existing commercially viable file storage systems are not single instanced. Converting an entire enterprise to single instanced would be a major undertaking. Large enterprises typically have a variety of different operating systems and legacy file systems deployed.

By contrast, our goal is to find duplication easily across multiple, heterogeneous, and potentially geographically distributed file systems, e.g., two separate department servers inadvertently mirroring a large, remote resource. Figure 1 shows an example. Using the duplication assessment method and tool presented in this paper, an IT department can make an argument for deploying the most appropriate duplicate removal solution, which might indeed be a single instance storage system.

From a data mining research perspective, this paper extends similarity detection or de-duplication methods to hierarchical structures, which pose a self-similarity problem among ancestors. Existing literature focuses on detecting near duplicates among many individual files, rather than detecting similarity among hierarchical abstractions (provided by directories). We contribute a method that is highly scalable and is also an *anytime algorithm*, providing results incrementally and designed to discover the best findings first. Although the exposition in this paper addresses file systems, the techniques generalize to other hierarchical structures, and even DAG structures.

The sections of the paper are structured as follows: First, we discuss two inadequate strawmen, and then review the mathematical machinery we leverage. Then we describe the two major phases of our algorithm in Sections 4 and 5. In Section 6 we describe the data mining utility tool we implemented. In Section 7 we describe an experience verifying that its scalability is sufficient to handle real-world enterprise data on the order of 22 terabytes; as this work is the first of its kind, there are no competing methods with which to compare efficiency or accuracy. Finally, we conclude with future work.

2. ILLUSTRATIVE STRAWMEN

The most obvious way of detecting duplicated files is to create a signature for each file in the system (for example, by hashing the contents of the file), and send all of the signatures to a central server where duplicates are detected using standard techniques, e.g. by detecting collisions in an enormous hash table. The problem with this simple approach is that it consumes too much disk, CPU and network resources. For example, in a large scale corporate network that we have studied, there are 350 million files, totaling 22 terabytes of data. Every one of these files would have to be read off the disk, its hash calculated using the local CPU, and the hashes transmitted to the central site for comparison. This would put an unacceptable burden on the corporate computer infrastructure. Moreover, even if this hash data could be computed freely, the personnel labor to then do something about each of millions of such pairs of files would be tremendous, and in most cases would make the return on investment of the endeavor unacceptable. By contrast, instead of dealing with individual files, if one can deal with whole directory hierarchies that have identical or near-identical content, then one has 1000+ fold leverage for the time spent reviewing duplication reports to make actionable decisions.

Our basic observation is that large-scale data duplication in file systems occurs when whole directories are duplicated, and when this happens the filenames and the structure of the duplicated directories remain more or less intact. Thus, rather than looking for duplicated files, we look for large directories that are duplicates or near duplicates of each other.

One alternative approach might seem to be to hash together all filenames that fall beneath each directory, giving a unique signature for each set of files. Two directories with identical contents would have an identical hash, and the collision could easily be detected via a large hash table. This analysis would certainly be lightweight, but if even a single filename does not match within the copied directory, then the hash will not match. We would like an analysis that is both lightweight and robust to slight perturbations.

3. MATHEMATICAL MACHINERY

The trick is to treat directories as sets of files they contain, and to use *set sketches* to efficiently estimate overlap in the full set of files, without having to compare every constituent file. The set sketch algorithm we use is based on the results by Broder, *et al.*

[2]. First we describe the mathematical results that underpin our method, and then we lay out the practical aspects of its application following this section.

Preliminaries:

U: a universe from which all sets of interest are drawn

Permutation of U: a function h is a permutation of U if it is a bijective function from U to U

Min-wise-independent family of permutations: a set of permutations of U that have a special "fairness" property (see [2] for definition)

Jaccard Measure of set similarity: Given two sets S_1 and S_2, the Jaccard measure of their similarity, $\rho(S_1,S_2)$, is defined as follows:

$$\rho(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

Sketch of length n: the sketch of a set is a vector of integers of length n, computed using the algorithm specified below.

Min(S): for the set of integers S, min(S) denotes the smallest element of S.

h(S): for the set of integers $S=\{s_1,s_2,\ldots\}$ and permutation h, h(S) is the set $\{h(s_1),h(s_2),\ldots\}$

Hamming Similarity: for two vectors $V=[v_1,v_2,\ldots,v_n]$ and $V'=[v_1',v_2',\ldots,v_n']$, the Hamming similarity of the two vectors, *sim(V,V')* is the number of positions i where $v_i=v_i'$. In other words, $sim(V,V') = |\{i : v_i = v_i'\}|$

Sketch Algorithm: Let H be a min-wise independent family of permutations over the universe U. Select uniformly, at random, and without replacement, n functions $h_1,h_2,\ldots,h_n$ from H. Then the sketch of a set S is the vector

$$[\min(h_1(S)), \min(h_2(S)), \ldots, \min(h_n(S))].$$

Notice that the functions $h_1,h_2,\ldots,h_n$ are chosen once and for all, and the same functions are used for computing the sketches of all sets. Using the results of Broder, et al. [2], it is easy to show:

Theorem 1 [2]**.** Consider two sets S_1 and S_2. Let V_1 and V_2 be the sketches of the two sets, according to the algorithm above. Then

$$E_{h_1,h_2,..,h_n \in H}[sim(V_1,V_2)] = n\rho(S_1,S_2)$$

So the set sketches can be used for estimating the degree of overlap between the two sets.

An extremely important property of the set sketches used in this paper is *incrementality,* by which we have the following in mind: given the sketches of two sets S and S′, it is very easy to compute the sketch of the set S∪S′ using the following lemma, the proof of which routinely follows from the definition of the sketch algorithm.

Lemma 1 Let $[k_1,k_2,\ldots k_n]$ be the sketch of S, and $[k'_1,k'_2,\ldots k_n']$ the sketch of S′. Then $[\min(k_1,k_1'),\min(k_2,k_2'),\ldots,\min(k_n,k_n')]$ is the sketch of S∪S′.

4. PHASE 1: DIRECTORY SKETCHES

Now we discuss the practical aspects of efficiently computing lightweight sketches for file system directories, and in the following section we discuss the analysis of this metadata.

As mentioned above, we consider directories to be the set of their constituent files. We need to answer the following three questions:

1. How do we represent a file as an integer?
2. How do we compute min-wise-independent hashes?
3. How do we compute the directory sketches, given that directories are hierarchical structures?

We answer each of these questions in turn, and follow with an additional opportunity for optimization.

4.1 File Representation

As stated previously, we wish to estimate the overlap between directories, where a directory is considered to be a set of files. To fit in the framework described in the previous section, we need to map files to integers in a finite universe, such that identical file copies are mapped to the same integer, and different files are mapped to different integers. We could use a strong hash function such as SHA1 or MD5 on the file contents, which would almost guarantee the one-to-one mapping of files to integers. But doing so would impose an unacceptable burden on the local system, since the contents of all files would need to be read from the storage systems and the CPU burdened to compute the hashes. Instead, we compute the MD5 hash of the string formed by concatenating the size of the file and its name (but excluding the file path, which would certainly differ between two copies). By excluding the actual file contents from the hash, we eliminate the need to read the vast majority of each disk. Of course, two files with the same name and same number of bytes may actually hold different contents. While this does allow for some false positive matches, in practice, a few false positives at the file level do not adversely impact the accuracy of our results, since we are only interested in large scale directory duplication that involves many other files.

Our detection relies on copies preserving their original filenames, which is typical, though not always the case. Any empirical study to verify how typical this behavior is would be specific to the data sample studied. In our anecdotal experience, when a large directory is duplicated, directory names are renamed more often than the individual files, e.g., appending a software version indicator to the top-level directory.

4.2 Min-wise-Independent Permutations

In general, true min-wise-independent permutation functions are very expensive to compute on large universes. The MD5 hashing algorithm is expensive, and computing n separate hashes with varied salt appended to the content would multiply the workload. We used an approximation that works well in practice when the elements of the set are already strong hashes. We describe this technique for the case when the elements of the universe are outputs of the MD5 hash function. Recall that MD5 hashes are 16 bytes. In this scheme, the permutation function *h* is represented by a random permutation of the set $I_{16}=\{1,2,3,\ldots,16\}$. Here is the algorithm:

somewhere/U/
somewhere/U/V/
somewhere/U/V/W/
…
elsewhere/U′/
elsewhere/U′/V/
elsewhere/U′/V/W/

Figure 2. Example: directory U′ is a copy of U.

- Choose, uniformly and at random, a permutation p of I_{16}
- Represent 16-byte integers as a vector of bytes $[b_1,b_2,\ldots,b_{16}]$
- Then $h([b_1,b_2,\ldots,b_{16}])=[b_{p(1)},b_{p(2)},\ldots,b_{p(16)}]$

This algorithm is extremely fast to compute and in practice satisfies the requirements of min-wise independence.

Using all 128 bits for each hash would be overkill for our application, since we can tolerate an occasional hash collision. Hence, we select only the first eight bytes of each permutation, yielding a 64-bit long integer for each of the n hashes.

4.3 Computing Directory Sketches

For the purposes of duplication assessment, we consider a directory to comprise the set of files accessible beneath it, irrespective of subdirectory structure. In reality, directories are hierarchical, each containing a number of files and sub-directories. In computing the sketch of a directory, it is essential that we re-use the effort spent computing the sketches of the sub-directories. Here Lemma 1 comes to our rescue: a bottom-up scan of a directory structure allows the sketch of all of the directories in that structure to be computed while every node in the structure (a file or sub-directory) is only touched once.

Hence, we perform a depth-first, post-order traversal of the file system, processing each directory after all of its contents. As each file is encountered, we compute the MD5 hash of its file name and size, and then update its parent directory sketch. When each directory is completed, we output its final sketch and then integrate it into its parent's directory sketch, leveraging the incrementality property. In addition, we also record the total number of files beneath each directory, as well as the cumulative content size of all of its files.

Ideally, the directory sketches are computed as locally as possible to each file system in order to eliminate network load and minimize the latency for the many round-trip communications that are needed to list an entire file system. For example, each file server could run several sketch-generation threads in parallel to distill the contents of its locally mounted file systems, perhaps spanning multiple disk drives. Afterwards, these distillations, which are much smaller than even a file listing, can be easily brought together to a central place for joint analysis. Figure 1 depicts this mechanism

It is important that n is the same for all distillations that are analyzed together. We set n=16 as a constant. The computation and distillation size scales linearly with n. Greater values lead to finer granularity in estimating set overlap, but often we find the most interesting pairs of directories are nearly perfect duplicates, which have all sketch hashes in common regardless of n. That is, when looking for highly similar directory pairs, we find little incentive to use larger n.

5. PHASE 2: INTER-SKETCH ANALYSIS

Given two specific directories of interest, we can now efficiently estimate the level of similarity in their sets of files by comparing their n hashes. But we don't know which directories to compare from the outset, and comparing each directory to every other directory is certainly infeasible.

In the first subsection, we describe a pruning technique that can optionally be used to eliminate most of the sketch metadata, allowing the fraction that remains to be analyzed entirely in RAM. This analysis is the subject of the second subsection.

5.1 Optional Pruning of Unique Hashes

Most directories are not duplicates, and if we can determine that their sketch hashes will uncover no useful duplicates, then we can eliminate the majority of the hashes from consideration, greatly improving scalability. We can detect a lack of duplicates for a directory only after we have gathered the various file system distillations in a central location. When we learn of a new distillation summary, we may discover that a directory previously considered unique now has a duplicate. It would not be sufficient to identify those hashes that have only a single occurrence in the distillations, since a subdirectory U/V/W/ and its ancestors U/ and U/V/ often have hashes in common, especially when U/ has few direct child files and U/V/W/ contains most of its files. Refer to Figure 2.

The proper procedure for determining the hashes of interest is as follows: we make a single pass through each distillation, which was generated naturally in post-order. We output each sketch hash to a temporary file when it fails to propagate to its parent's sketch, or else when it propagates all the way to the root. For example, if a particular hash X first appears in the sketch of a subdirectory U/V/W/ and it propagates up to the sketch of U/V/ and also U/, then these three occurrences stem from only a single origin and X is output only once in the temporary file—when it fails to propagate to the parent of U/ or else when it reaches the root /. On the other hand, if that same hash X later occurs in a sketch of a different branch U′/V/W/ of the listing, perhaps in a separate distillation, then it would be output a second time.[1]

Note that the procedure to construct the temporary file emulates the original computation to obtain the initial sketches. But the workload is much less because at most n sketch hashes are considered per directory, rather than all of the files. If instead one were to output the unique hashes during the initial computation, then additional files of roughly the same size as the distillations would need to be managed and moved across the network to the central location, which we deem an inferior design. Furthermore, each distillation can be processed in parallel, and their temporary output merged arbitrarily.

The temporary file of hashes is then sorted via a high-quality, external-memory merge-sort, such as provided by the UNIX or DOS `sort` utility program. Any sketch hash that occurs more

[1] This algorithm also works correctly for the case where a directory U/V/ has a copy in a sibling directory U/V′/: the hash X propagates from U/V/ to its parent U/, but the same hash X in U/V′/ fails to propagate to its parent U/ because X is not strictly less than the previous X installed by the sketch of U/. So, X is correctly output again, marking the multiple originations.

Table 1. Overall Algorithm

1. For each distributed file system in parallel:
    - Perform a depth-first post-order traversal: [See 4.3]
        - Case file: (if size > 0)
            - Compute MD5 digest of basename & size [4.1]
            - For each of n 8-byte permutations, select the min-hash for the parent directory's sketch [4.2]
        - Case directory: [4.3]
            - Output its sketch (if cumulative size > 10 MB)
            - Merge with parent's sketch [Lemma 1]
2. Gather the distributed distillations
3. Optional: determine the sketch hashes that originate in more than one directory, greatly pruning the sketch data [5.1]
4. For each directory D, sorted largest (& shallowest) first: [5.2]
    - Add it to the directory↔hash bi-partite graph
    - Determine how many of D's sketch hashes are common with previously registered directories for those hashes [Thm.1]
    - For each such directory P: (except if P is an ancestor of D)
        - Output D↔P unless their similarity is less than or equal to a previously output pair that covers D↔P

than once in the sorted output represents a file that appears in multiple subdirectories, and may contribute to discovering substantially similar directories. By far, most hashes originate from a single location and can be disregarded henceforth. Any directory whose n sketch hashes are completely eliminated is also omitted from further processing. Although this step is optional, it has the advantage of greatly improving the scalability of the remaining analysis—using memory and computation only for the relatively few directories that do have some degree of duplication.

5.2 Incremental Indexing

To discover the interesting pairs of duplicates, we could build a reverse index of the hashes pointing back to the directories from which they come, i.e., a bi-partite graph of directories and their hashes. It could then be analyzed to discover each pair of directories and their percentage of sketch hashes in common (as in [4]). But this approach does not scale well for hierarchical directory information, since there can be a very large number of directories with some degree of similarity—for example, ancestor chains are self-similar—and many subordinate directories may be discovered first, overwhelming the discovery process. For example, in Figure 3, we see that directory A/, which contains subdirectories 1…m, has been copied to another location A′/. Ideally, we would like to discover the pair A/ = A′/ first, and avoid the overhead of discovering the many subordinate pairs of duplicate subdirectories.

We accomplish this by sorting all of the directories by their cumulative content size, largest first (favoring shallower depth in case of ties, which are surprisingly common), and then building up the directory-hash graph incrementally. In the example above, A/ may be registered first, with no duplicate directories discovered at this time, and later when A′/ is registered, its sketch

```
somewhere/A/
somewhere/A/1/
somewhere/A/2/
        …
somewhere/A/m/
        …
elsewhere/A′/
elsewhere/A′/1/
elsewhere/A′/2/
        …
elsewhere/A′/m/
```

Figure 3. Example: directory A′ is a copy of A.

hashes will connect it with A/, and the pair will be discovered before their subdirectories are linked in. At this point, one may consider the option of not processing any subdirectories of A/ or A′/. But this would be overly hasty. It may be that the subdirectory A/1/ is relatively large and has a duplicate copy in some other location D/1/. If we were to omit registering A/1/, then this duplication could not be detected. We have encountered exactly this scenario in analyzing real data.

Instead, we do eventually register A/1/, but when later registering A′/1/ and thereby discovering its mate A/1/, we omit this duplication pair from the pool of interesting findings because their mutual parents A/ and A′/ already cover this finding. Thus, the ordering of the directories from largest to smallest leads to discovering the largest (highest) pairs of similarity first, and also serves to filter the pairs reported later on in the discovery process. If the number of similar pairs of directories is overwhelmingly large, at least we have an *anytime* algorithm that finds the largest such pairs first. Our graphical user interface described in the next section takes advantage of this, allowing the user to examine initial findings while the analysis continues in the background to post additional findings.

There are two additional complications. First, many directories naturally share self-similar hashes with their parents and higher ancestors. When each such pairing is discovered, the ancestor relation is tested to omit it from the pool of interesting findings. Second, it may be that two directories U/ and U′/ share some level of similarity, say, 14 of 16 sketch hashes in common, and their subdirectories U/V/ and U′/V/ share a greater level of similarity, e.g., 16 hashes in common. The filtering as described so far would filter away the latter pair because it is covered by the prior pair. The additional detail is that we specialize the filter for each level of similarity. That is, once a pair is discovered with s hashes in common, it will be used to filter any pairs it covers with ≤ s hashes in common, but not pairs with greater similarity. Table 1 summarizes the overall algorithm.

6. OUR DATA MINING UTILITY TOOL

While the algorithm described above is at the heart of our software, any actual application must address concerns and practical features that are useful in real-world deployment.

We built the tool in Java for portability, and made it available (for HP-internal use) via WebStart, which lets one install and run the application directly from a web page. This is intended to make it easy for administrators on different systems to run the distillations on each file server locally. The user interface enables one to scan

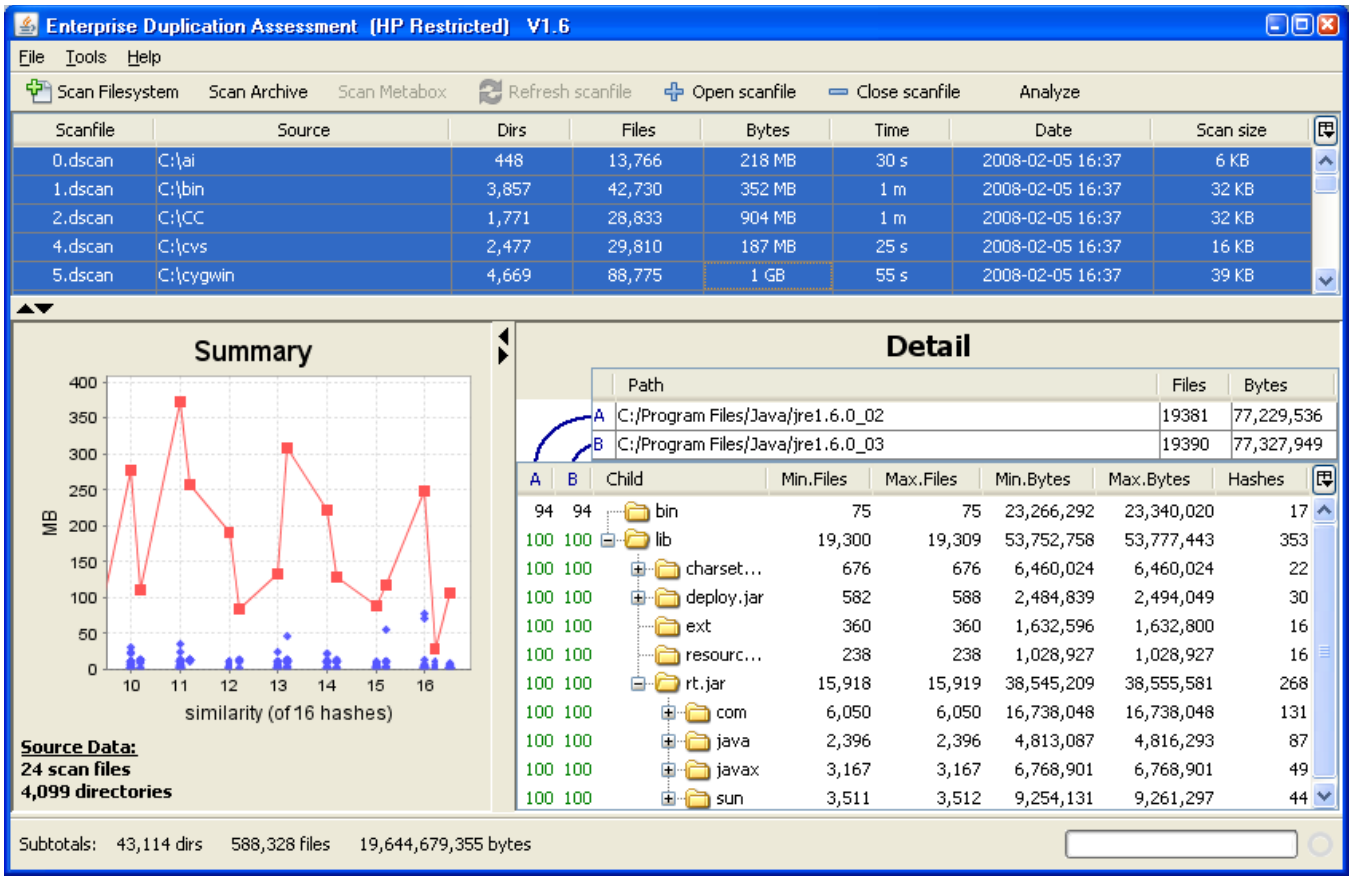

Figure 4. Screen shot of our utility tool.

one or more directory hierarchies in parallel threads, which can improve throughput, especially if multiple disk drives are involved. Alternately, it can list the contents of a compressed archive of a file system, or read a simple file listing thereof.

A common type of duplication is to have a compressed project archive in addition to an unpacked version, perhaps with a few differences or the addition of some *.o compilation objects. This happens, for example, when software is downloaded from the Internet, stored permanently, and also unpacked. In order to catch this common use-case, there is an option when scanning a file system to also list the contents of compressed archives that it encounters. Archive files are treated as pseudo-directories and expose the directory structure inside the archive; this requires no other special-case processing in the algorithm. Finally, for information security, there is an option to encrypt all recorded directory names.

Each "*.dscan" distillation file is stored in compressed format; each line lists a directory path, the cumulative number of files and bytes it represents, and the n=16 hashes of its sketch. Despite the random hashes, gzip compression saves ~25%. As a practical option, we omit listing small directories with less than a *minimum directory size*—10 MB of file content is the default. Furthermore, when building the sketches, we ignore files of size 0, which are surprisingly common in some file systems. This can greatly reduce the directory-hash graph connectivity, and thereby improve scalability without missing meaningful data duplication.

After one has generated these *.dscan distillation files, perhaps on various remote servers, one gathers them together on a single computer for analysis. Figure 4 shows an example screen shot of our tool. The upper window pane shows a table of the various *.dscan files that are currently included in the project, totaling 19 GB in 588K files in 43,114 directories (see status bar). Observe that the last column of the table shows the size of these distillation files. A few kilobytes represent hundreds of megabytes of file content (~ 1:25,000 ratio); this ratio depends on the nature of the data, as well as the minimum directory size.

By activating Tools→Analyze or the toolbar button, all of these files are jointly pruned of hashes that originate in only a single directory, and the remaining directory sketches are loaded into memory. The analysis immediately opens the lower panel, and as the anytime algorithm generates findings incrementally, they are posted to the summary graph in the lower left pane. Each blue dot indicates a pair of similar directories. The x-axis represents the degree of similarity, and the y-axis represents the number of megabytes that might be saved if the smaller of the two directories were deleted; in some cases, this reflects the size of the compressed archive copy. An additional red curve gives a rough estimate of the cumulative savings if all such pairs were eliminated at each level of similarity.

The similarity on the x-axis is primarily the number of sketch hashes in common. But because we have additional information about the directories, we also know when a pair of similar

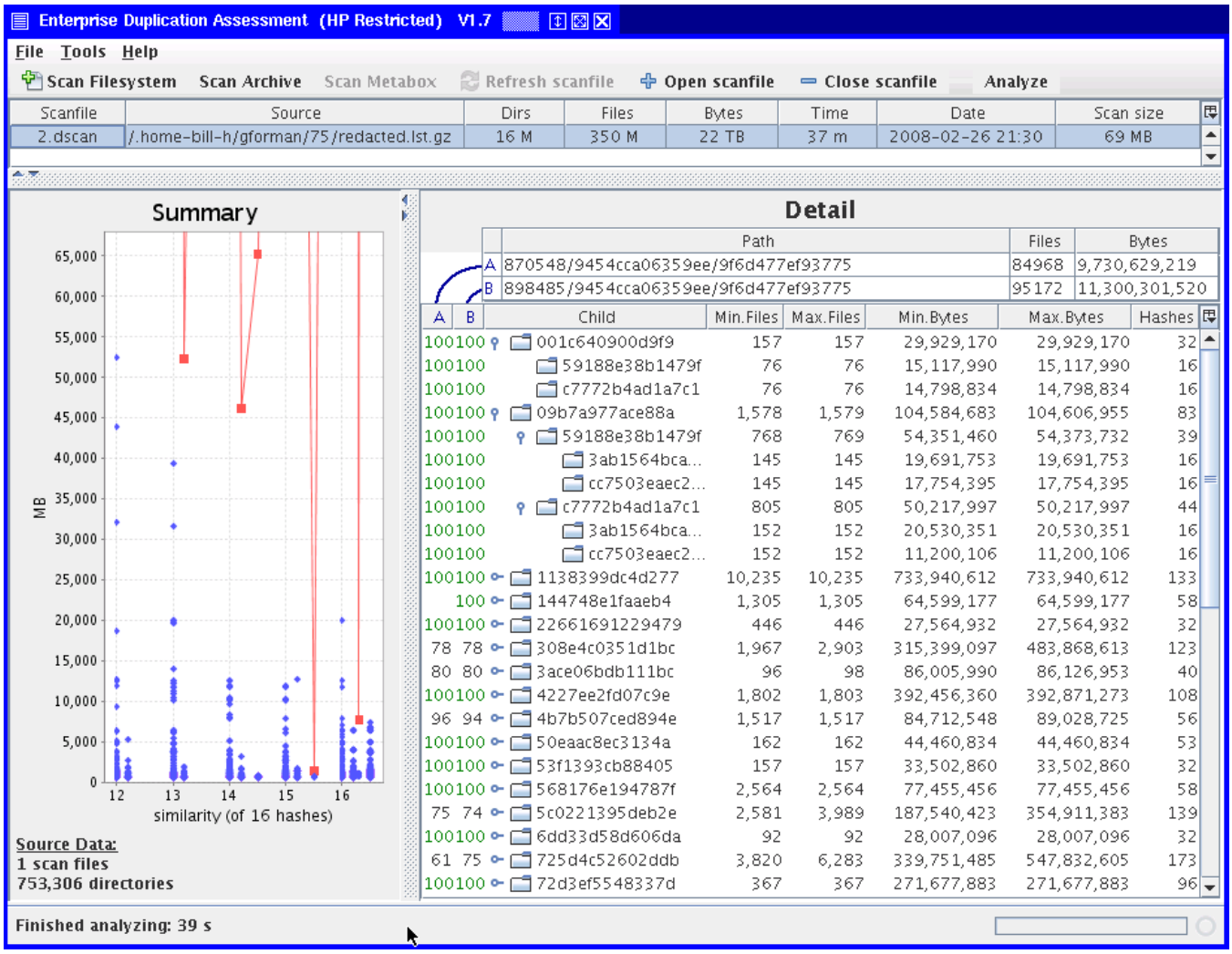

Figure 5. Sample finding from analyzing 22 TB of files.

directories happens to contain the same number of files. To visually distinguish such pairs as being more similar, we add 0.2 to the similarity score. Likewise, by the cumulative contents counter, we also know when a pair of similar directories contains the identical number of content bytes. When this happens, we add an additional 0.3 to the similarity score. Thus, perfect duplicates obtain a score of 16.5, and appear furthermost to the right. Although *ad hoc*, it proves useful in practice to distinguish at a glance these different degrees of similarity. One could extend this idea to further analyze the similarity in structure of their subdirectories and their sketches, or go back to the original file system itself and verify whether all files are present.

The user can zoom a portion of the graph by dragging the mouse. If the user clicks near a blue dot, the pair of similar directories is shown in the detail pane in the lower right, along with a drill-down view of their mutual subdirectory structure in the bottom right tree-table. For each subdirectory name in common, we determine the union set of sketch hashes we have available under the two directories. The first two columns in the lower tree-table show the percentage of hashes covered by the A directory and respectively the B directory—often 100% even if the exact number of files or content bytes does not match identically.

The user may right-click on a directory to open it in a file explorer, or to compare the two directories via an external directory comparison tool, such as the open-source WinMerge utility, which performs comparisons on both file contents and (recursive) directory listings.

7. AN EXAMPLE LARGE ANALYSIS

We had available a large NFS trace listing the distributed file systems of a large enterprise customer. The filenames and path names were encrypted for security, and completely inscrutable to us. The file listing contained 350 million files in 16 million directories, totaling 22 terabytes of cumulative file data. Compressed, the text file listing totals 3.9 GB.

Our tool distilled this listing into a *.dscan file in 37 minutes. For comparison, simply uncompressing the whole file listing and passing it to the 'wc' word counter takes 6 minutes, and listing directly from the file systems would take much longer. Our resulting *.dscan file includes 753,306 directories of >10 MB, and

its compressed format consumes just 69 MB, i.e. 1:319,000 compared to the 22 TB file content or 1:56 compared to the compressed file listing. We ignored 4.5 million files (1%) having zero size (prior to this enhancement, the analysis phase became swamped with a highly interconnected graph).

Loading and analyzing the *.dscan file took less than one minute. The system stops analyzing after it finds the largest 9000 similar directory pairs over a specified minimum similarity; we generally look for just a few large opportunities for consolidation. Figure 5 shows a sample screen shot; the directory names have all been encrypted to hexadecimal strings for customer privacy. The summary graph has been zoomed in to show several pairs exceeding 20 GB of duplication. Whether these are essential for data backup or caching is unclear to us because of the filename encryption, and would require further investigation with the customer, who is unavailable to us. By clicking on a dot, the lower right panel drills down on a particular pair having roughly 10 GB of duplication. The file paths are inscrutable to us for customer privacy, but in practice an administrator can often determine what the nature of the duplication is or at least can find out which file owners to blame. The point of this large customer analysis here is not what was found in particular or how much total duplication was found globally, but rather to validate that the design of the analysis software enables one to mine the data for large-scale directory duplication easily and efficiently.

Typical findings in our file systems include multiple versions of large software packages that have been both downloaded and installed (e.g. the Java JDK), directories of photos or music that have been duplicated as a form of manual backup or tagging (e.g., copying an album to a "favorites" directory), daily versions of large virus definitions that are automatically downloaded by virus checkers, and mirrored copies of repositories. Some copies are made simply to rename the navigation path for user convenience or to attempt to ensure longevity of one's access to shared files. Depending on the cost and benefit to an organization, the duplication may be deemed worthwhile; the aggregate cost of many uncoordinated duplicates, however, may not.

8. CONCLUSION & FUTURE WORK

All in all, we find our algorithm and implementation highly scalable. The remaining bottleneck has to do with user attention, and ultimately social incentives may be needed to get people to actually consolidate, once the best opportunities have been found. This is an eternal struggle between the IT storage managers and the employee users.

There are two main avenues for deploying this technology: the lightweight consulting assessment scenario, which has a large discovery component and in which privacy and invasiveness considerations dominate, and the continuous monitoring scenario, which is more performance-focused. The latter scenario suggests future work to remember prior decisions made on large directory pairs, e.g., "this is an important mirror of that," or "only this copy is sanctioned; other copies that come into being should have their owners notified." Finally, supervised and unsupervised data mining technologies may also be brought to bear in order to quicken the repeated review process.

9. ACKNOWLEDGMENTS

Our thanks to Eric Anderson and Brad Morrey of HP Labs for obtaining and sharing the large, encrypted NFS traces. We also wish to thank the anonymous customer enterprise for graciously sharing their data for file systems research such as this. Additional thanks to Janet Wiener and Ken Braly for improving this manuscript.

10. REFERENCES

[1] Bolosky, W. J., Corbin, S., Goebel, D., and Douceur, J. R. 2000. Single instance storage in Windows® 2000. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4* (Seattle, Washington, Aug. 3-4, 2000). USENIX Association, Berkeley, CA, 2-2.

[2] Broder, A. Z., Charikar, M., Frieze, A. M., and Mitzenmacher, M. 2000. Min-wise-independent permutations. *Journal of Computer and System Sciences.* 60, 3 (Jun. 2000), 630-659.

[3] Douceur, J., Adya, A., Bolosky, W., Simon, D., Theimer, M. 2002. Reclaiming Space from Duplicate Files in a Serverless Distributed File System. In *the 22nd IEEE International Conference on Distributed Computing Systems* (ICDCS′02).

[4] Forman, G., Eshghi, K., and Chiocchetti, S. 2005. Finding similar files in large document repositories. In *the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining* (Chicago, Illinois, USA, August 21 - 24, 2005). KDD ′05. ACM, New York, NY, 394-400.

[5] Gantz, J. F. *et al*. 2007. The Expanding Digital Universe: A Forecast of Worldwide Information Growth Through 2010. IDC White Paper, Framingham, MA. June 22, 2007. www.idc.com

[6] Simpson, D., and Hatcher, J. TIP survey reveals storage trends. InfoStor Europe, Dec. 2006. www.infostor.com