Finding Similar Files in Large Document Repositories

George Forman, Kave Eshghi, Stephane Chiocchetti
Intelligent Enterprise Technologies Laboratory
HP Laboratories Palo Alto

HPL-2005-42(R.1)
June 15, 2005*

E-mail: george.forman@hp.com, kave.eshghi@hp.com, stephane.chiocchetti@hp.com

content management, document management, near duplicate detection, similarity, scalability

Hewlett-Packard has many millions of technical support documents in a variety of collections. As part of content management, such collections are periodically merged and groomed. In the process, it becomes important to identify and weed out support documents that are largely duplicates of newer versions. Doing so improves the quality of the collection, eliminates chaff from search results, and improves customer satisfaction.

The technical challenge is that through workflow and human processes, the knowledge of which documents are related is often lost. We required a method that could identify similar documents based on their content alone, without relying on metadata, which may be corrupt or missing.

We present an approach for finding similar files that scales up to large document repositories. It is based on chunking the byte stream to find unique signatures that may be shared in multiple files. An analysis of the file-chunk graph yields clusters of related files. An optional bipartite graph partitioning algorithm can be applied to greatly increase scalability.

* Internal Accession Date Only
To be published in and presented at the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'05), 21-25 August 2005, Chicago, IL, USA
Approved for External Publication

Finding Similar Files in Large Document Repositories

George Forman
Hewlett-Packard Labs
1501 Page Mill Rd. MS 1143
Palo Alto, CA 94304 USA
george.forman@hp.com

Kave Eshghi
Hewlett-Packard Labs
1501 Page Mill Rd. MS 1143
Palo Alto, CA 94304 USA
kave.eshghi@hp.com

Stephane Chiocchetti
Hewlett-Packard France
1 ave Du Canada, MS U215
91947 Courtaboeuf, France
stephane.chiocchetti@hp.com

ABSTRACT

Hewlett-Packard has many millions of technical support documents in a variety of collections. As part of content management, such collections are periodically merged and groomed. In the process, it becomes important to identify and weed out support documents that are largely duplicates of newer versions. Doing so improves the quality of the collection, eliminates chaff from search results, and improves customer satisfaction.

The technical challenge is that through workflow and human processes, the knowledge of which documents are related is often lost. We required a method that could identify similar documents based on their content alone, without relying on metadata, which may be corrupt or missing.

We present an approach for finding similar files that scales up to large document repositories. It is based on chunking the byte stream to find unique signatures that may be shared in multiple files. An analysis of the file-chunk graph yields clusters of related files. An optional bipartite graph partitioning algorithm can be applied to greatly increase scalability.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing; I.5 [**Computing Methodologies**]: Pattern Recognition

General Terms

Algorithms, Documentation, Management, Performance

Keywords

content management, document management, near duplicate detection, similarity, scalability

KDD'05, August 21–24, 2005, Chicago, Illinois, USA.

1. INTRODUCTION

A critical factor in the success of any large technology business is having a high-quality repository of technical support knowledge. It is essential to have this knowledge recorded in documents and not just in the heads of the technical support engineers. In electronic form, it can be searched and shared much more effectively by the people who need it, internally and externally. At Hewlett-Packard (HP) we have millions of technical support documents covering many different products, solutions, and phases of support. Building such a body of knowledge comes both from having a culture of sharing, and also from integrating external and acquired content.

But adding in new documents and merging collections can lead to a variety of problems that degrade the overall quality of the repository, among them that content may be duplicated. For example, if subsidiaries are acquired or companies merge that have previously integrated third-party content, then many duplicates arise. Sometimes authors prefer to copy rather than link to content by reference, to avoid the possibility of a dead pointer later. Furthermore, duplication may emerge as new documents are created that copy portions of documents in separate collections that are later merged. Metadata about which documents are related can easily become corrupt or may be missing from the outset.

Rather than simply allowing duplicates to proliferate with growth, HP has used an analysis process we developed to identify and help weed out duplication. This and other data mining efforts at HP are used to manage the content and thereby improve customer satisfaction.

If our content were to consist of edited news articles, then a reasonable approach might be to identify whole paragraphs that appear in multiple articles. However, the domain of technical support documents does not easily lend itself to breaking into discrete paragraphs, and so our solution relies on chunking technology to break up the documents into paragraph-like pieces in a semantically unmotivated, but *consistent* manner. By detecting collisions among the hash signatures of these chunks, we efficiently determine which files are related in a large repository.

In Section 2 we describe the methods we use. In Section 3 we discuss our implementation, performance, and a few of our business-relevant results, including the value in what we *didn't* find. In Section 4 we discuss enhancements to address practical issues that surfaced. In Section 5 we highlight related work, and then we conclude with future work in Section 6.

2. METHOD

There are three essential steps in the process of finding similar files in a file collection. The first step is to break up each file into a sequence of chunks using a content-based chunking algorithm. The second step is to compute the hash of each chunk. Thus each file is now represented by a list of hashes. The third step is to find those files that share chunk hashes, reporting only those whose intersection is above some threshold.

Below we describe these steps in more detail, including key embellishments and valuable implementation notes from our experience.

2.1 Hashing Background

We use the 'compare by hash' method to compare chunks occurring in different files [5]. Hash algorithms map long sequences of bytes, in our case the chunks, to short, fixed size sequences in such a way that it is almost impossible to find two chunks that have the same hash. We use the MD5 algorithm, which generates 128-bit hashes. Using the hash for comparison rather than the chunk itself has two advantages: (1) comparison time is shorter, (2) hashes, being short and fixed size, lend themselves to efficient data structures for lookup and comparison.

2.2 Chunking

Content-based chunking, as introduced in [7], is a way of breaking a file into a sequence of chunks so that chunk boundaries are determined by the local contents of the file. This is in contrast to using fixed size chunks, where chunk boundaries are determined by the distance from the beginning of the file; inserting a single byte at the beginning would change every chunk.

The Basic Sliding Window Algorithm [7] is the prototypical content-based chunking algorithm. This algorithm works as follows: There are a pair of pre-determined integers D and r, $r < D$. A fixed width sliding window of width W is moved across the file, and at every position k in the file, the fingerprint, F_k, of the contents of this window is computed. A special, highly efficient fingerprint algorithm, e.g. Rabin's fingerprint algorithm [8], is used for this purpose. k is a chunk boundary if $F_k \bmod D = r$ (see Figure 1).

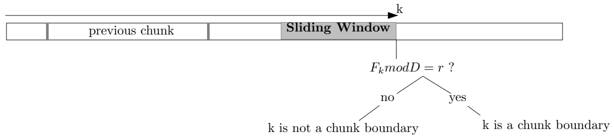


Figure 1: The Sliding Window Algorithm

2.3 Chunking and File Similarity

The rationale for using content-based chunking algorithms to break the files is the following property of these algorithms: When two sequences R and R' share share a contiguous sub-sequence larger than the average chunk size for the algorithm, there is a good probability that there would be at least one shared chunk falling within the shared sequences (Figure 2). Thus, to detect shorter shared sequences, shorter chunks are needed, implying more of them.

We use a chunking algorithm, TTTD (details in [3]), that performs better than the basic sliding window algorithm in the following sense: for the same average chunk size, there is a higher probability that a shared sub-sequence includes a whole chunk. The use of this algorithm improves the accuracy (recall) of our file detection algorithm.

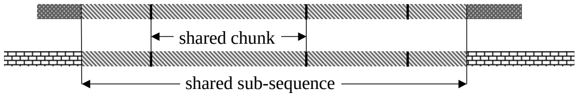


Figure 2: Shared Chunks in Shared Sub-Sequences

Notice that, as shown in Figure 2, the total length of the shared chunks only provides a *lower bound* on the length of the shared sub-sequence. This is sufficient for detecting pairs of similar files, and a post-processing comparison of pairs of specific files may be used to determine the exact amount and quality of the similarity.

Figure 3 shows an example pair of similar files discovered in an actual analysis of the HP OpenView *Engineering Documents and Notes* repository. The three minor differences are circled. The file on the right is an updated version, with the missing word 'you' filled in. Sometimes authors update a support document, but by mistake or limited authorization, the previous version is not removed, and in many cases the association between the files is lost.

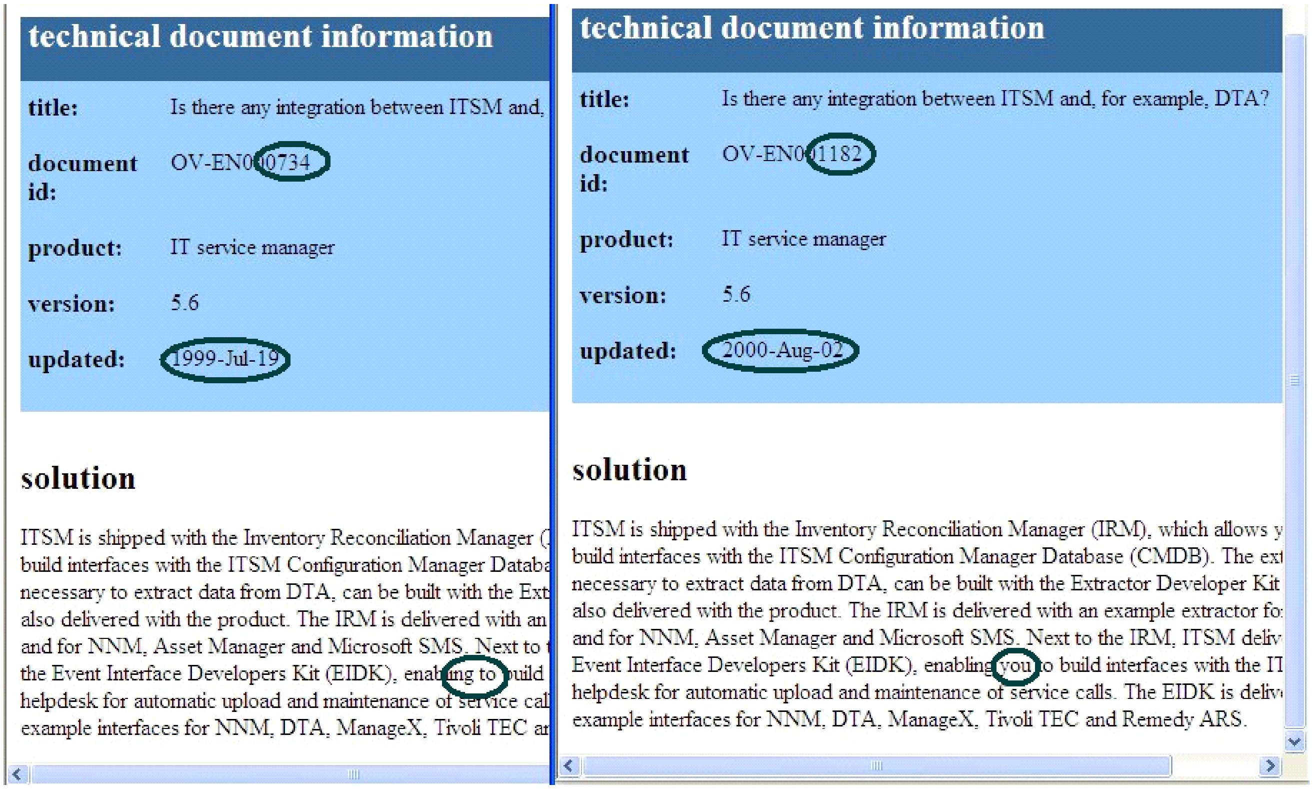
For purposes of illustration, we ran our chunking algorithm on the visible text of each of these files, and reveal the individual chunks in Figure 4. Since the first difference occurs early in the file, their first chunks do not match, depicted by the different colors and different chunk boundaries. Starting near the end of the first line of the solution there are three chunks in common, plus two more in common at the end. (Note that if the chunks were instead based on paragraph boundaries, this similarity would not have been detected.) Based on these commonalities, the analysis determines the lower bound for the bytes in common. In the actual analysis, 58% of these two files were determined identical based on their chunks computed from the raw HTML source files.

Next we enumerate the steps of the file similarity algorithm:

File Similarity Algorithm

1. For each file in the repository, break its content into chunks and for each chunk, record its byte length and its hash code. The size of this metadata amounts to a few percent of the original content, depending primarily on the hash length and the average chunk size parameters.

```
./OpenView/EngineeringDocs/file832.html
298  9E123814C58254D237F9E19B5D9C4E5A
184  33F3C217EBDFC714C8996D2559484D6F
142  DD275200C54CBF7262809BBD4D134F215
103  804C3E31FB559E2A8289A9015624C392
152  B0E77C988953A1E0DDA5D2FBF262D07B
...
```



technical document information

title: Is there any integration between ITSM and,

document id: OV-EN000734

product: IT service manager

version: 5.6

updated: 1999-Jul-19

solution

ITSM is shipped with the Inventory Reconciliation Manager (
build interfaces with the ITSM Configuration Manager Databa
necessary to extract data from DTA, can be built with the Ext
also delivered with the product. The IRM is delivered with an
and for NNM, Asset Manager and Microsoft SMS. Next to t
the Event Interface Developers Kit (EIDK), enabling to build
helpdesk for automatic upload and maintenance of service call
example interfaces for NNM, DTA, ManageX, Tivoli TEC ar

technical document information

title: Is there any integration between ITSM and, for example, DTA?

document id: OV-EN001182

product: IT service manager

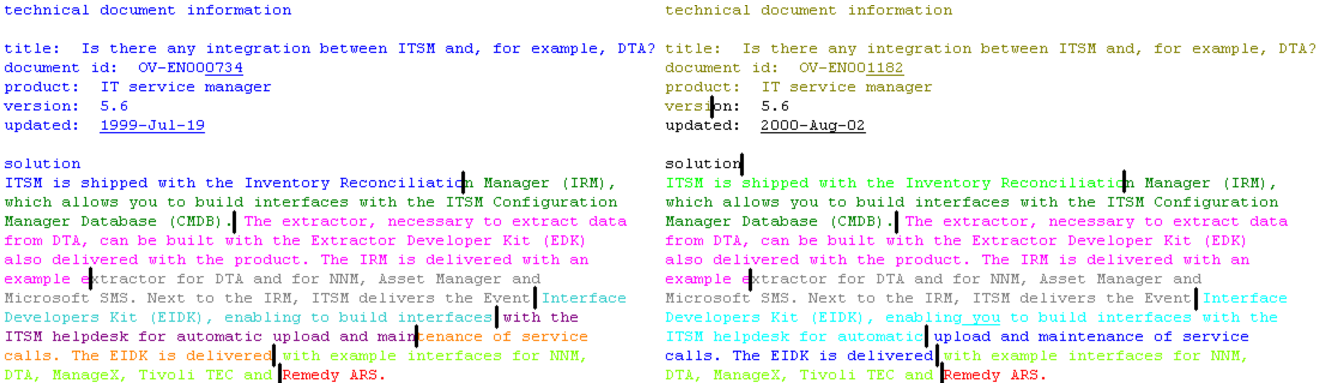
version: 5.6

updated: 2000-Aug-02

solution

ITSM is shipped with the Inventory Reconciliation Manager (IRM), which allows y
build interfaces with the ITSM Configuration Manager Database (CMDB). The ext
necessary to extract data from DTA, can be built with the Extractor Developer Kit
also delivered with the product. The IRM is delivered with an example extractor fo
and for NNM, Asset Manager and Microsoft SMS. Next to the IRM, ITSM deliv
Event Interface Developers Kit (EIDK), enabling you to build interfaces with the IT
helpdesk for automatic upload and maintenance of service calls. The EIDK is deliv
example interfaces for NNM, DTA, ManageX, Tivoli TEC and Remedy ARS.

Figure 3: Comparison of two technical support documents that are near duplicates.



```
technical document information

title:  Is there any integration between ITSM and, for example, DTA?
document id:  OV-EN000734
product:  IT service manager
version:  5.6
updated:  1999-Jul-19

solution
ITSM is shipped with the Inventory Reconciliation Manager (IRM),
which allows you to build interfaces with the ITSM Configuration
Manager Database (CMDB). The extractor, necessary to extract data
from DTA, can be built with the Extractor Developer Kit (EDK)
also delivered with the product. The IRM is delivered with an
example extractor for DTA and for NNM, Asset Manager and
Microsoft SMS. Next to the IRM, ITSM delivers the Event Interface
Developers Kit (EIDK), enabling to build interfaces with the
ITSM helpdesk for automatic upload and maintenance of service
calls. The EIDK is delivered with example interfaces for NNM,
DTA, ManageX, Tivoli TEC and Remedy ARS.
```

```
technical document information

title:  Is there any integration between ITSM and, for example, DTA?
document id:  OV-EN001182
product:  IT service manager
version:  5.6
updated:  2000-Aug-02

solution
ITSM is shipped with the Inventory Reconciliation Manager (IRM),
which allows you to build interfaces with the ITSM Configuration
Manager Database (CMDB). The extractor, necessary to extract data
from DTA, can be built with the Extractor Developer Kit (EDK)
also delivered with the product. The IRM is delivered with an
example extractor for DTA and for NNM, Asset Manager and
Microsoft SMS. Next to the IRM, ITSM delivers the Event Interface
Developers Kit (EIDK), enabling you to build interfaces with the
ITSM helpdesk for automatic upload and maintenance of service
calls. The EIDK is delivered with example interfaces for NNM,
DTA, ManageX, Tivoli TEC and Remedy ARS.
```

Figure 4: Illustration of example chunks for the two texts in Figure 3.

It is important that the bit-length of the hash code be sufficiently long to avoid having many accidental hash collisions among truly different chunks. Given a b-bit hash code, one can expect about a 50% chance of having one or more accidental collisions if there are $2^{\frac{b}{2}}$ chunks from a random population [5]. For our 128-bit MD5 hash, this exceeds $10^1$9 chunks, which is overkill for our application. Even a 64-bit hash is sufficient for ∼4 billion chunks, and a few accidental collisions are likely acceptable for the inexact nature of this data mining work.

2. *(Optional step for scalability)* Prune and partition the above metadata into independent sub-problems, each small enough to fit in memory. The details of this step are described in Section 2.5. The remaining steps are applied to each partition of the problem.

3. Load the file-chunk metadata into memory, constructing a bipartite graph with an edge between a file vertex and a chunk vertex iff the chunk occurs in the file.[1] The file nodes are annotated with their file length, and the chunk nodes are annotated with their chunk length.

4. Construct a separate file-file similarity graph as follows. For each file A:

   (a) Look up the chunks AC that occur in file A.

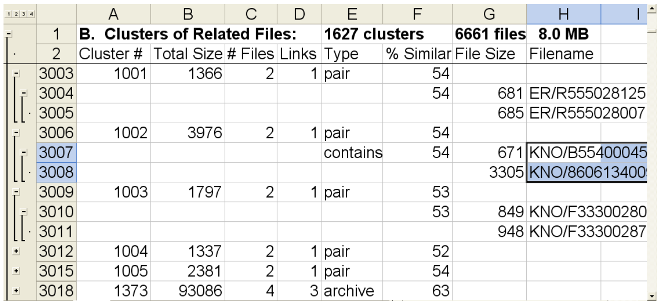
   (b) For each chunk in AC, look up the files it appears in, accumulating the set of other files BS that share any chunks with file A. (As an optimization due to symmetry, we exclude files that have previously been considered as file A in step 4.)

   (c) For each file B in set BS, determine its chunks in common with file A,[2] and add A-B to the file-file similarity graph if the total chunk bytes in common exceeds some threshold, or percentage of file length.

5. Output the file-file similarity pairs as desired. In our implementation, we use the well-known *union-find algorithm* to determine clusters of interconnected files. We then output the clusters sorted by size, and with each, the list of file-file similarities found, the total number of files involved, the average similarity of the links, etc. See the screen shot in Figure 5.

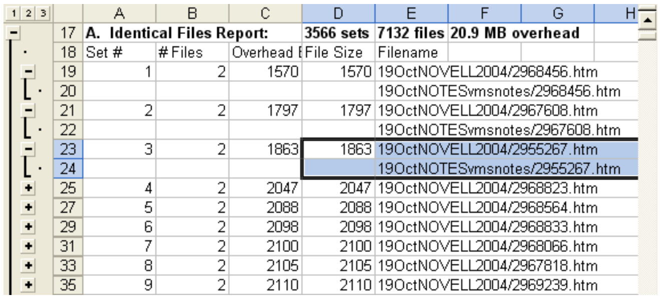
2.4 Handling Identical Files

A common special-case of similar files is having multiple files with identical content. Although this can be handled sufficiently by the process just described, the output is more easily understood if this sort of duplication is recognized and handled separately. For example, it can be a much quicker decision to retain only a single copy of each document, installing forwarding symbolic links where needed, or not merging any content that is already represented in the master repository.



	A	B	C	D	E	F	G	H	I
1	**B. Clusters of Related Files:**				**1627 clusters**		**6661 files**	**8.0 MB**	
2	Cluster #	Total Size	# Files	Links	Type	% Similar	File Size	Filename	
3003	1001	1366	2	1	pair	54			
3004						54	681	ER/R555028125	
3005							685	ER/R555028007	
3006	1002	3976	2	1	pair	54			
3007					contains	54	671	KNO/B55400045	
3008							3305	KNO/860613400	
3009	1003	1797	2	1	pair	53			
3010						53	849	KNO/F33300280	
3011							948	KNO/F33300287	
3012	1004	1337	2	1	pair	52			
3015	1005	2381	2	1	pair	54			
3018	1373	93086	4	3	archive	63			

Figure 5: Clusters of files with some duplication.



	A	B	C	D	E	F	G	H
17	**A. Identical Files Report:**			**3566 sets**	**7132 files**	**20.9 MB overhead**		
18	Set #	# Files	Overhead I	File Size	Filename			
19	1	2	1570	1570	19OctNOVELL2004/2968456.htm			
20					19OctNOTESvmsnotes/2968456.htm			
21	2	2	1797	1797	19OctNOVELL2004/2967608.htm			
22					19OctNOTESvmsnotes/2967608.htm			
23	3	2	1863	1863	19OctNOVELL2004/2955267.htm			
24					19OctNOTESvmsnotes/2955267.htm			
25	4	2	2047	2047	19OctNOVELL2004/2968823.htm			
27	5	2	2088	2088	19OctNOVELL2004/2968564.htm			
29	6	2	2098	2098	19OctNOVELL2004/2968833.htm			
31	7	2	2100	2100	19OctNOVELL2004/2968066.htm			
33	8	2	2105	2105	19OctNOVELL2004/2967818.htm			
35	9	2	2110	2110	19OctNOVELL2004/2969239.htm			

Figure 6: Sets of filenames with identical content.

This may be provided for using the same metadata with a small enhancement to the basic algorithm. While loading the file-chunk data, we compute a hash over all the chunk hashes, yielding a single hash that uniquely identifies files by their content alone. During loading in step 3, we maintain a hash table that references file nodes according to their unique content hashes. If a file has already been loaded, we note the duplicate filename and avoid duplicating the chunk data in memory. In the output, multiple filenames referring to identical content are listed out separately in a report. (Figure 6 shows a screen shot of an example report.) This significantly reduces the volume of output to examine: each copy of a file would otherwise multiply the number of similar links found for it. This is illustrated in Figure 7, which shows two clusters of similar files in which each file had a duplicate copy in another directory. With special handling for identical files, we reduce the number of links in this graph from 21 to 4.

2.5 Bipartite Partitioning for Scaling Up

The algorithm as described can easily process file sets with tens of thousands of files comprising hundreds of megabytes. However, to process larger repositories, we need a partitioning method to break the problem into pieces small enough to fit in physical memory, or else virtual memory thrashing will occur because of the essentially random access patterns while examining the bipartite graph. Towards this end, we present the partitioning algorithm we use, which is suited especially for the large but sparse bipartite graphs typically generated by these analyses.

[1] Although it is rare, should the same chunk occur more than once in a single file, we record multiple forward edges for that file-chunk pair.

[2] When determining the chunks in common between files A and B, we sped up the inner loop of the analysis ∼2.5x by using a *merge-sort* style comparison rather than a hash table to count collisions.

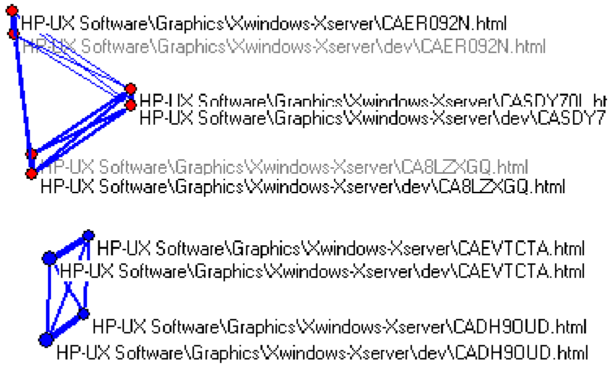

HP-UX Software\Graphics\Xwindows-Xserver\CAER092N.html
HP-UX Software\Graphics\Xwindows-Xserver\dev\CAER092N.html
HP-UX Software\Graphics\Xwindows-Xserver\CASDY70L.ht
HP-UX Software\Graphics\Xwindows-Xserver\dev\CASDY7
HP-UX Software\Graphics\Xwindows-Xserver\CA8LZXGQ.html
HP-UX Software\Graphics\Xwindows-Xserver\dev\CA8LZXGQ.html
HP-UX Software\Graphics\Xwindows-Xserver\CAEVTCTA.html
HP-UX Software\Graphics\Xwindows-Xserver\dev\CAEVTCTA.html
HP-UX Software\Graphics\Xwindows-Xserver\CADH9OUD.html
HP-UX Software\Graphics\Xwindows-Xserver\dev\CADH9OUD.html

Figure 7: Without special handling for identical files, the number of similar file links multiplies.

1. Have the file-chunk metadata stored in a file format that contains the chunk hash code *and the filename on each line*. In practice, it is much more efficient to refer to the filename and its long directory path via a short index number into a separate table of filenames stored in a database, for example:

   ```
   298  9E123814C58254D237F9E19B5D9C4E5A  file832
   184  33F3C217EBDFC714C8996D2559484D6F  file832
   142  DD275200C54CBF7262809BD4D134F215  file832
   103  804C3E31FB559E2A8289A9015624C392  file832
   152  B0E77C988953A1E0DDA5D2FBF262D07B  file832
   ...
   ```

2. Perform a *disk-based sort* on this data, as is commonly available in UNIX 'sort' implementations. This is a well-developed field of computer science, and can be performed in parallel, etc., to perform an efficient sort with minimal I/O. At the end of this step, all the files that refer to a given chunk will be in adjacent lines in the file. (Alternately, one could employ a disk-based recursive hash-bucket partitioning, which yields the same desired post-condition property, but without having to achieve a global ordering. While this might be more efficient, it is not as ubiquitously available as UNIX 'sort'.)

3. Perform the highly efficient *union-find algorithm* on a graph where each file is represented by a vertex. For each chunk encountered in the sorted input, all the different files it appears in are unioned into the same connected component. At the end of this process, the connected components are identified by the union-find data structure. Each of these represents a subset of files which may be processed separately.

   Because the file-chunk graph is bipartite and the chunks are sorted, we do not need to represent the many chunk vertices to partition the overall graph. Since there are often two orders of magnitude more chunks than files, this step is highly scalable. Our implementation scales to 15 million files within 1GB RAM.

4. In practice, a majority of the connected components contain only a single file, and these can be immediately discarded as not being similar to any other files, requiring no further processing.

5. (Optional) A standard *bin-packing algorithm* can be used to gather together multiple connected-components that can fit in memory all at once. If there are many small connected-components, then this reduces the number of separate files generated in the following step, and the number of times the remaining steps in the master algorithm must be run. While this is optional, we found this greatly reduces process set-up and tear down overhead.

6. Linearly scan the file-chunk metadata (either the sorted file or the original unsorted version) and output its lines into distinct sub-problem files by which partition the file has been assigned by the connected-components or optional bin-packing. Each of these files serves as an independent input to the remainder of the master algorithm.

Even for problems containing tens of thousands of files that do not require partitioning, the heavy degree of pruning achieved in step 4 alone greatly speeds up the file-chunk graph analysis, so we generally include this optional partitioning process.

We have run into a situation a couple times where the graph could not be partitioned because of one or a few chunks that represent a ubiquitous header in all the files, such as the common practice of placing a copyright notice at the top of all source code files. To combat this problem, we ignore in step 3 chunks whose degree exceeds some large threshold, such as half the total number of files.

2.6 Complexity Analysis

The chunking of the files is linear in the total size N of the content. The bipartite partioning, implemented via sorting as we have, takes O(C log C), where C is the number of chunks in the repository, including duplicates. Since C is linear in N, the partitioning step takes O(N log N). Finally, the analysis of the file-chunk graph in step 4 of the main algorithm is O(C log C). For this analysis, we assume that as the total number of files grows, the average number of files that are related to a given file does not grow substantially.

3. RESULTS

We implemented the chunking algorithm in C++ (~1200 lines of code). We used Perl to implement the similarity analysis algorithm (~500 LOC), the bipartite partitioning algorithm (~250 LOC), and a shared union-find module (~300 LOC). We leveraged the HP-UX 'sort' utility, which includes the ability to sort files larger than memory. We use ASCII file formats for the intermediate data, and generate our hierarchical reports in Microsoft Excel format, which allows for hierarchical sorting and filtering of the clusters. Clearly the implementation could be made more efficient if need be, but run-time cycles and disk space were far cheaper than programming effort and calendar time.

3.1 Performance

The performance on a given repository ranges widely depending on the average chunk size, which is a controllable parameter. In practice, we often start with a large chunk size, e.g. 1000 bytes, to get a quick characterization of a new repository. This analysis will completely miss pairs of files that are similar due to shorter common sequences. We then re-run the analysis with a smaller chunk size, e.g. 100 bytes, to get a more fine-grained comparison. Note that we can independently set a minimum threshold of, say, 1000 bytes in common between two files for reporting purposes. Thus, there is a tradeoff between computational effort and the likelihood of missing a positive match.

A complete performance characterization is beyond the scope of this paper and depends largely on the characteristics of the dataset. Here we present an example run to give some feel for its performance. We consider a particular repository of 52,125 technical support documents in 347 folders, comprising 327 MB of HTML content. These measurements were performed on an HP Kayak XW with a 3 GHz Intel processor and 1 GB RAM: The chunking process ran in just 4 MB RAM at ∼80% CPU utilization, indicating the intense file input bandwidth needed, and especially the file system overhead for opening over fifty thousand individual files. With the target chunk size set to 5000 bytes, the chunking process took 25 minutes and generated 88,510 chunks (targeting 100 byte chunks, it took 39 minutes and generated 3.8 million chunks). The remaining similarity analysis took 14 seconds at 100% CPU utilization and used just 32 MB RAM. If we first run the bipartite partitioning algorithm, taking 4 seconds and fitting in memory for this problem, then the analysis phase takes less than 2 seconds—an overall improvement due to pruning.) Again, these numbers are only suggestive, and vary substantially depending on the dataset. The main point is that it is quite tractable.

3.2 Business Results

Due to the business nature, we can divulge only limited information about the business impact. Our implementation has been used successfully over a range of projects, and on over a dozen different technical support repositories. The run-time for the whole algorithm including the partitioning phase on different projects ranges from 20 minutes to three days on a single server.

In a project referred to earlier, we found ∼1000 pairs of nearly duplicate documents in one of the HP OpenView repositories containing over 40,000 documents. Domain experts used this information to perform cleaning on the content. This cleaning effort would be unthinkable without this data mining technology.

In another project, what we did not find proved valuable. We analyzed a series of repositories that were under consideration for being integrated into a larger HP collection. Based on not finding any substantial number of similar files, the business decided to proceed with the migration, without having to carefully prune duplicates.

In another project, the algorithm found a large number of pairs of identical files (see Figure 6). It turned out that two different subdirectories contained the same set of files. On first blush this may sound easy to detect, but to better appreciate the problem, consider having to migrate unfamiliar content in hundreds of nested subdirectories under time pressure.

4. DISCUSSION

The great majority of the computation is in determining the chunks of the source files, which may be done in a parallel or distributed fashion. Furthermore, separate repositories that do not share storage or administrative domains can still collaborate in efficient duplicate detection, so long as they use the same chunking method. Only the file-chunk metadata need be shared, which typically amounts to only 1–3% of the total file volume in our system. This is a great benefit if the repositories are geographically distributed in a worldwide company like HP, or belong to different companies that pool their technical support knowledge.

In practical use, the output can often include clusters of files that are similar simply because they contain a large template in common. For example, there was a collection of HTML files generated by a common process that inserts a long JavaScript pre-amble. All of these appeared in a single large clique. Given the many different processes for capturing and processing technical documentation, there can be many such templates. A similar problem can arise with large boilerplate texts regarding licensing or copyright terms, which may be present in many of the files. Such frequent 'false alarms' are a serious impediment to practical use of the output, and can mask true duplication of content in different templates.

In order to disband such clusters, the analyst can mark them as being due to template similarity. The software can then record that the chunks in common represent a template, e.g., of the 185 documents in the cluster, there may be 17 chunks that are shared in all of them. These chunks are to be ignored on future runs, effectively reducing the file sizes to their non-template payload. Occasionally this process must be repeated more than once, but generally a single iteration is sufficient. Note that the files do not need to be re-chunked for each iteration.

5. RELATED WORK

Broadly related work in determining file similarity includes pair-wise file comparison, such as UNIX 'diff', and identical file detection, such as the Windows NTFS Single Instance Store technology. These are related, but are not designed for finding only partly similar files in a large repository.

Brin et al. [1] propose to have a large indexed database of existing documents, and offer a way to detect whether a given new document contains material that already exists in the database. Note that their method is a 1-vs-N documents method, whereas ours is all-to-all. If one were to use their 1-vs-N method repeatedly, and add a new document to the database after each test, the execution time of this method would be untenable. By contrast, ours is designed to scale up to very large document collections by processing the whole batch at once. Also, note that using their method in this way would yield an algorithm that is sensitive to the order that documents are added to the database, whereas ours is not sensitive. Finally, their paper only describes mathematical abstractions, which if implemented directly, yield performance that does not scale beyond the memory capacity of one machine; whereas our method is explicitly designed for scalability beyond memory capacity.

Furthermore, in their method chunk boundaries are based on the hash of 'text units,' which can be paragraphs, sentences, etc. When the input file is not prose text (e.g., technical documentation with file listings or program output) their notion of a unit does not apply in a natural fashion. Our approach, using the enhanced TTTD chunking algorithm that uses a sliding window to determine the chunk boundary, leads to chunk sizes of any desired average length and can be applied to text as well as non-text data. It also overcomes the problems of the simple sliding window algorithm with respect to repetitive text (e.g. long strings of the same character).

In the AltaVista shingle method by Broder et al. [2], instead of chunking up the files and storing the hashes of the chunks, the shingles of each file are stored and used for comparison. A shingle is the fingerprint of a fixed size window in the file whose fingerprint mod a pre-determined number is another pre-determined number. To find the shingles, a fixed size window is slid across the file and the fingerprint is calculated at each position (in a manner similar to that used for finding chunk boundaries). Each fingerprint whose value satisfies the mod criterion is included among the shingles. The many shingles generated for a file are then sorted and the largest 10 (a parameter) are recorded for the file (this reduces storage and later computation, but it reduces the probability of detecting a match). The intuition behind the shingle method is that two files which share a shingle probably share the window of text that the shingle is a fingerprint of. The more such shared shingles, the more similar the two files are. Shingle comparison is coarser in terms of accuracy than chunk comparison. For example, if the chunks of two files are identical and in the same order, then the two files are identical; if the shingles of the two files are identical and in the same order, the two files may still be different in the areas that are outside the boundaries of the shared shingles. Also, using chunks it is possible to compute the size of the shared areas by adding the sizes of shared chunks. With shingles this is not possible, since a) shingles may overlap and b) shingles do not cover the areas between shingles. Thus, to achieve the same accuracy as our method, one would need a lot more shingle hashes.

Finally, we consider the *sif* system by Manber [6], which is also a N-vs-N document comparison method. Like AltaVista shingles, his 'fingerprint' hashes only consider a few bytes at the boundary of a chunk, whereas we compute the hash code of the entire chunk. Thus, his system is also blind to any differences in the file between these fingerprints. Also, because our system keeps track of the length of each chunk, when a shared chunk is found, we can state something about the lower bound of how many bytes are held in common. In his system, each common fingerprint is only a probabilistic clue that the files contain some similarity. His fingerprint algorithm suffered drawbacks that TTTD does not have, as described previously.

Manber stated, "It turns out that providing a good way to view the output is one of the major difficulties." We agree, and have partly addressed this by setting aside identical files in a separate report, by clustering the results and by providing a means for sorting and filtering these clusters interactively after the analysis. Moreover, we have addressed the issue of getting past template similarity to get to more meaningful results.

6. CONCLUSIONS

Large corporations, such as Hewlett-Packard, have a variety of content repositories for capturing knowledge. In the process of managing these through incremental growth, acquisitions, mergers, and integration efforts, it is inevitable that some duplication of content will ensue. To counter this natural entropy, HP employs a process that mines the repository for partially duplicated material, helping to maintain the quality control of the content. To identify pieces that may have been duplicated, the solution relies on chunking technology rather than paragraph boundary detection, because of the nature of technical support documentation.

Though the overall process is adequately efficient with computer resources, we find that in practical use the bottleneck is in human attention to consider the many results. Towards this end, we provide special handling for exact duplicates, and a way to reduce a frequent source of false alarms—template similarity. Future work may profitably focus on further reducing false alarms and missed detections, and making the human review process as productive as possible, rather than conserving computer resources.

This technology may also prove useful for other business purposes, such as regulatory compliance. For example, it could be used to help enforce the purging of data that has reached its end-of-life and must be deleted, despite having been copied into other documents without linking metadata.

7. REFERENCES

[1] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 398–409, San Jose, CA, 1995.

[2] A.Z. Broder, S.C. Glassman, M.S. Manasse, G. Zweig. Syntactic Clustering of the Web. *Computer Networks and ISDN Systems*, 29(8-13):1157–1166, 1997.

[3] K. Eshghi and H.K. Tang . A Framework for Analyzing and Improving Content-Based Chunking Algorithms. Hewlett-Packard Labs Technical Report TR 2005-30

[4] R.A. Finkel, A. Zaslavsky, K. Monostori, and H. Schmidt. Signature extraction for overlap detection in documents. In *Proceedings of the 25th Australasian Conference on Computer Science*, v4, pages 59–64, Melbourne, Australia, 2002.

[5] V. Henson and R. Henderson. Guidelines for Using Compare-by-Hash. Forthcoming, 2005. http://infohost.nmt.edu/˜val/review/hash2.html

[6] U. Manber. Finding similar files in a large file system. In *Proceedings of the Winter 1994 USENIX Technical Conference*, San Francisco, CA, January 1994.

[7] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (SOSP '01), pages 174–187, Banff, Canada, October 2001.

[8] M.O. Rabin. Fingerprinting by Random Polynomials. Tech. Rep. TR-15-81, Center for Research in Computing Technology, Harvard Univ., Cambridge, Mass., 1981.