Clean Living: Eliminating Near-Duplicates in Lifetime Personal Storage

Zhe Wang
Princeton University

Jim Gemmell
Microsoft Research

September 2005

Technical Report
MSR-TR-2006-30

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Clean Living: Eliminating Near-Duplicates in Lifetime Personal Storage

Zhe Wang

Computer Science Dept, Princeton University

Princeton, NJ, USA

Jim Gemmell

Microsoft Research
Redmond, WA, USA

Abstract

As lifetime personal storage is becoming a reality, we find that it is becoming increasingly difficult to search and navigate the contents one accumulates. One of the most striking issues is the duplicates and near duplicates that clutter search and navigation. We investigated different technique to eliminate the duplicates and near duplicates objects in the MyLifeBits personal storage system. Our results show the effectiveness of near-duplicate detection on personal contents like emails, documents and web pages visited. In one experiment, duplicate and near-duplicate detection reduced the number of documents a user must consider by 21% and the number of web pages by 43%.

Categories and Subject Descriptors

H.3.0 [**Information Storage and Retrieval**]: General H.5.4 [**Information Interfaces and Presentation**]: Hypertext/ Hypermedia – *Architectures, Navigation, User issues*

General Terms

Human Interfaces, Measurement, Experiment.

Keywords

Memex, duplicate, database.

1. Introduction

The MyLifeBits system [1,2,3] is a personal storage system to store and manage a lifetime's worth of *everything* – at least everything that can be digitized. As we accumulate more contents, it becomes increasingly difficult to search and navigate the data. One of the main problems is the duplicate and near duplicates in the presentation. A typical keyword search and/or time range search would usually return hundreds if not thousands of items. One would like to cluster the results together so that the near duplicate contents would be hidden from standard search result.

Determining only exact duplicates would be very simple, but does not work well in our environment for a number of reasons:

- Documents could have small revisions to the original one.
- Web pages (even with same URL) often change slightly every time you visit.
- The same document could be saved into different formats, eg: pdf, doc, etc.
- Emails in a thread tend to be similar but not exact the same.

Our goal is to be able to would reduce the clutter of the user interface. By grouping the duplicates and near duplicates into clusters, all the contents within the same cluster are hidden behind the single representative content.

Near duplicate detection is commonly used by web search engines, who also do not want to overwhelm the user with many results that are deemed "practically the same." Some of the existing techniques for determining near-duplicate web pages include:

- Approximate fingerprint: Create a collection of fingerprints for each document, and compare those fingerprints to detect similar documents.
- Shingling: Reduce the document into a collection of shingles, and then use the shingle or super shingle match to detect near duplicates.
- I-Match: Create a list of words in the document and filter out the words with low IDF value, create a single fingerprint of the content to identify near duplicates.

We used the shingling technique as our basic algorithm to do near duplicate detection and applied them on our document collection. We also tried various filtering technique to see their impact on near duplicate detection. Finally, we used the traditional document edit distance measure to check our clustering result to make sure we do not have false positive on near duplicate clustering.

It is impossible to cluster all documents people think are similar together due to the different ways people think. Even if agreed differences were to be used, many of these would be at a semantic level that we have not addressed. For example, people tend to think of a reply to an email as being similar to the original email, even if they do not share any common content. In this case, application specific heuristics would be needed to further cluster the documents. In our experiment, we considered only document content to determine similarity.

In the remainder of the paper, we outline existing algorithms and explain our approach. Our experimental setup is described, and we present our experimental results. We conclude that through efficient near duplicate

detection algorithm, we can significantly reduce “clutter”. We show that an IDF word filter, while very effective for web document collections [7], is not as effective on the personal document collections.

2. Near-duplicate detection algorithm

2.1 Existing algorithms

The idea of using fingerprint and checksum to identify duplicates has been used in many contexts. Cryptography checksums like md5 or sha1 are widely used in file systems to make sure the files are not changed. In recent peer-to-peer system development, they have also been used to uniquely identify the contents being exchanged. The checksum works well to detect any changes made to the file as a whole, but lacks the flexibility to detect minor modifications.

Udi Manber introduced an algorithm to produce approximate fingerprints to detect similar files within a large file system [4]. The algorithm calculates a Rabin fingerprint [5] value of a sliding window of 50 characters. It selects a fixed subset of fingerprints, and uses them as representatives to compare documents for similarity. This approach reduces the task of detecting similar documents to comparing the collection of fingerprints for each. This is much faster than actually comparing the documents, but in order to find clusters, the algorithm would still need to compare documents with each other and have a run time of $O(n^2)$.

Andrei Broder’s super shingle algorithm [6] uses similar algorithm to convert the document into a set of fingerprints. Rather than depend on the set of fingerprints only, the introduction of min-wise independent permutation algorithm allows the estimation of the document based on the minimum fingerprint. And by creating a *feature* (super fingerprint) using several fingerprints, the algorithm can cluster the near-duplicates within $O(n\log n)$ time. More descriptions of this algorithm will be given in the next section.

The I-Match algorithm proposed by Abdur Chowdhury, et al, [7] does not rely on strict parsing, but instead uses the statistics of the entire document collection to determine which terms to include in the fingerprinting. The words with smaller IDF (Inverse Document Frequency) are filtered out since they often do not add to the semantic content to the document. After filtering, all the terms are sorted (removing duplicates) and only one fingerprint is generated for the document. Two documents would be treated as near duplicates if their fingerprint matches. The overall run time for I-Match algorithm would be $O(n\log n)$ in the worst time when all documents are duplicates to each other and $O(n)$ otherwise.

2.2 Our method

Our requirement is a fast and scalable solution to detect near duplicates in personal life storage. Any algorithm that has overall run time of $O(n^2)$ is not realistic for our real time application. We also need an algorithm that can support efficient dynamic addition of content since we keep adding new documents into the store in real time.

We have chosen to use Broder’s super shingle algorithm to form our baseline algorithm since it can be implemented efficiently in term of dynamic content insertion. We did not use the I-Match algorithm because the addition of just one high IDF word would mean rejection as a near duplicate. While this may be acceptable in a web environment, it is not suitable for personal life storage, where the different versions of same document are very common, and often come about as documents are extended (and possibly extended by adding high IDF words). However, while we did not want high IDF words to carry such weight, we also did not want low IDF words to have undue influence. Therefore, we included an optional step for ignoring low IDF words.

Here is a detailed description of our full algorithm:

1. All documents are processed through Microsoft iFilter[8] interface wherever an iFilter is available for the document type. The output of the iFilter is a text-only document with all formatting and meta tags removed. So, for example, the .doc and .pdf versions of the same document should produce about the same output from their respective iFilters.[1]
2. We reformat the text streams to clean up the spaces between words to create a standard word streams. This step is necessary since the document still contains tabs and extra spaces after iFilter.
3. The text stream is then filtered through our own customized filters. We have designed three kinds of filters: number filter, stop-word filter, low-IDF words filter. We also tried stemming in some experiments.
4. We apply andrei’s algorithm, in which the 14 smallest Rabin fingerprints are collected from each text stream, and combined using the sha-1 algorithm to form one *feature*.
5. Using six different rabin fingerprinting parameters, we create six *features* for each text stream.

---

[1] If the documents have complex formatting, the order of the text may differ. For example, a footnote (like this one) may be placed right after the footnote reference, or at the end of the document; there is no “correct” place for an iFilter to put it. This is another reason that near-duplicate detection is required.

6. We also use the traditional fingerprinting method to create a sha-1 fingerprint for the whole document and use this sha-1 fingerprint to uniquely identify exact copies of the same documents.

7. The database "fingerprints" table is populated with the document fingerprint and six *features*. When a new document is inserted, we would check to see if the new document shares at least two features with any existing document. If so, then we label the new document as the near duplicate of the existing document. The threshold of two features was selected so that we obtain less false positive with reasonable confidence on closeness of the two documents. Please refer to [6] for more details.

8. The database "clusters" table is populated with the document item_id, document fingerprint and cluster_id. Different documents sharing the same cluster_id if they are near duplicates to each other.

9. At the query processing time, we typically have a query result table as a result of some user issued query. By left joining the query result table with the clusters table, we will be able to identify all the duplicates within the result set quickly and present them to user properly by hiding the duplicates in the result set.

3. Experiment setup

3.1 Verification of cluster result

Since the goal of our clustering is to detect near duplicates and remove the clutter from the presentation, we need to be very careful about not generating false positives: we should not cluster two documents together if they are significantly different, because this could effectively hide the clustered documents from user's search. On the other hand, we are not too concerned with a few false negatives, as the impact is a little more clutter to the user, but at they can still find what they are looking for.

In order to verify that we do not have false positive, we use the ***word edit distance*** (also known as the "Levenshtein distance") to determine how much difference there is between a pair of documents in the same cluster. Traditionally, the edit distance between two strings is defined as the minimum number of operations needed to transform one string into the other, where the operations could be character insertion, deletion or substitution. And the typical dynamic algorithm to solve edit distance has the running time of O(*mn*) where *m* and *n* are the lengths of the two strings involved. We extend this definition for string edit distance a little bit by treating the word as individual component. By calculating the "word edit distance", we speed up the calculation and reduce the memory space requirement. The calculation of word edit distances of every pair of documents in our collection (~100,000) would be prohibitively expensive. However, it would only be necessary if we were very concerned with false negative, which, as explained above we are not. In order to discover false positives, it suffices to calculate the edit distances only for pairs of documents within the same cluster, and this is what we have done.

3.2 Clustering method

Once we have a set of *features* to help identify the near duplicates, it is easy to detect whether two documents are near duplicates or not by checking to see if they share more than two *features* or not. This check gives us a way to detect the pair-wise near duplicate relationship.

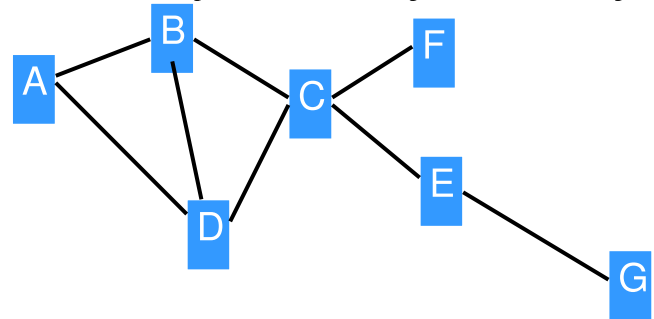

Figure 1 Clustering by full transitivity

Beginning, with pair-wise near duplicates, a number of different approaches make be taken to form the documents into larger clusters. As shown in figure 1, it is possible for the documents to share different features with each other and form a cluster naturally if we add a line for any two pair of near duplicate documents. The cluster is also independent of the order of document insertion. But the problem would arise when the cluster is too big and form long stretch of links between documents (Eg: document A and G are four links away and could potentially be not very similar to each other).

We used the word edit distance to check one set of real documents, and found that it did indeed cause such problem (Fig 2): there are some documents within the same cluster having word edit distance greater than 50%.

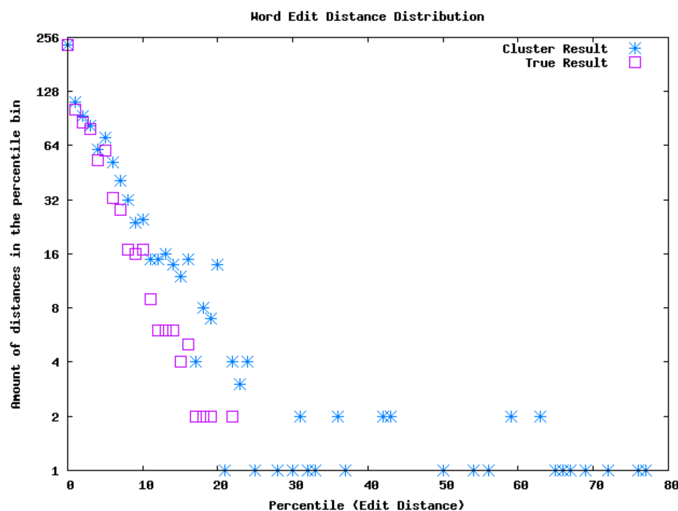

Figure 2 "Word Edit Distance" on fully transitive cluster

Since it is critical to avoid false positive, we decided to use a very simple but conservative method to do clustering: We would only cluster documents together when they are near duplicates to a "cluster representative." For Figure 3, let's assume the document insertion order is from A to G. When document B is inserted, it would be treated as the same cluster as A (A is inserted earlier and will act as the cluster representative), and it would be removed from consideration in the future. In our example, we formed three clusters where document A, C and G would be the cluster representatives.

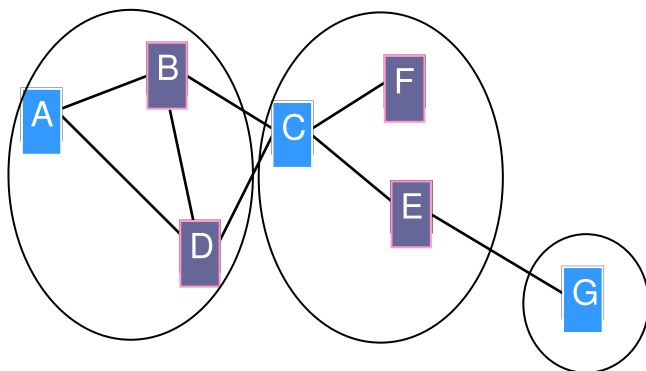

Figure 3 Conservative clustering

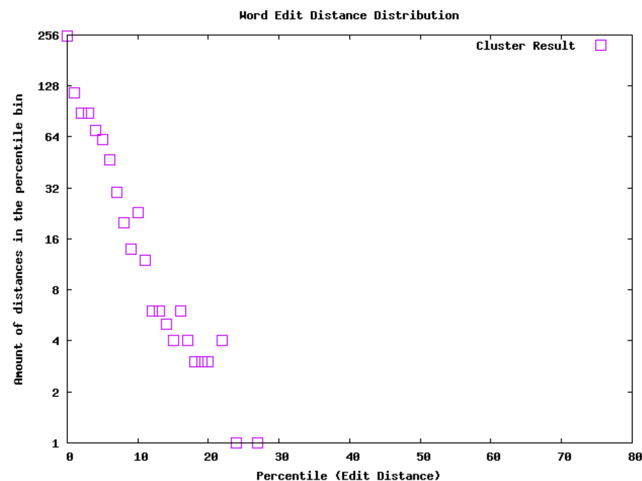

Figure 4 Word Edit Distance on conservative clustering

Figure 4 shows that using conservative clustering, the largest "word edit distance" between two documents within the cluster is smaller than 30%. Table 1 showed that with this simple clustering algorithm, we are still able to capture most of the clusters. It is possible to use more complicated algorithm to form clusters, but for our real time application, we would like to adopt a simple yet effective algorithm. Since these two clustering algorithms are the two extremes in term of aggressiveness of clustering, other clustering algorithm would most likely have total clusters between 3770 and 3886. So we believe the simple approach is good enough for our purpose.

	Fully transitive	Conservative clustering
Total objects	8358	8358
Total unique objects	4756	4756
Total clusters	3886	3770

Table 1 Comparison of two clustering methods

4. Experiment results

We have run our near duplicate detection application on Gordon Bell's document collection [2]. The collection at the time of our experiment contained about 14000 documents, 90000 emails and 39000 visited web pages. Bell has experienced so much frustration with clutter that he has sometimes performed deletion to removed near-duplicates, even though his goal was to keep everything and to perform as little manual maintenance as possible.

Table 2 shows the result from the experiment. Each row corresponding to different document type and last row is the sum of all three types. For the columns:

- Exact duplicates: Exact duplicates found by fingerprinting the whole document.
- No IDF filter: Near duplicates found by near duplicate detection without any IDF filter.

- IDF-4.0 filter: Near duplicates found by filtering each document through the high pass filter where any term with IDF value less than 4.0 is removed.
- IDF-5.0 filter: Near duplicates found by filtering out terms with IDF value less than 5.0.

Type	Exact duplicates	Near-duplicates		
		No IDF filter	IDF 4.0 filter	IDF 5.0 filter
Document	10%	9%	10%	4%
Email	3%	9%	8%	8%
Web page	37%	9%	8%	7%
Total	13%	9%	8%	7%

Table 2: Duplicate and near-duplicate detection on Bell's corpus.

As we can see, in this experiment, exact duplicates account for about 13% of all the documents while near duplicates detected by our system account for about 9% of all the unique documents. It is interesting to see that near duplicates account for similar percentage for all three types, while most of the exact duplicates come from web pages. We exclude the exact duplicates when counting for the near duplicates. If we count both the exact duplicates and near duplicates we find, we can effectively reduce 21% of all documents and 43% of web pages from the viewer to reduce clutter of the user interface.

We found that adding an IDF filter was not very helpful. Only in the case of IDF-4.0 applied to documents did it discover more near-duplicates. In all other cases, it actually reduced near-duplicate detections. We believe the reason is that after IDF filtering, short documents may be reduced to the point of being too small to be fingerprinted by our algorithm.

5. Conclusion

In our experiment with near duplicate detection for life long personal data, we found that the technique used in web page near duplicates detection can be adopted efficiently in personal data as well. Our method of using ifilter to retrieve the real text content and then use feature to detect near duplicates can eliminate about 10% of near duplicate documents in additional to 13% exact duplicates. Adding an IDF filter (inspired by the i-Match algorithm) did not provide any benefit.

In this work, we have considered only text similarity. Web pages duplicates were a major concern for us, and we did not want a new banner ad cause our system to consider the page to be the different. This may not always be desirable, for instance, if one is actually looking for a banner ad. It also may be undesirable for user-created documents. We plan to extend our research into audio and image similarity in the future.

Acknowledgements

Gordon Bell has been a key collaborator on the MyLifeBits effort and he has given us generous support on experimenting with his personal data. Roger Lueder and Jim Gray provided many helpful suggestions.

References

[1] Gemmell, Jim, Bell, Gordon, Lueder, Roger, Drucker, Steven, and Wong, Curtis, MyLifeBits: Fulfilling the Memex Vision, *Proceedings of ACM Multimedia '02*, December 1-6, 2002, Juan-les-Pins, France, pp. 235-238.

[2] Gemmell, Jim, Bell, Gordon and Lueder, Roger, MyLifeBits: a Personal Database for Everything, Communications of the ACM, vol. 49, Issue 1 (Jan 2006), pp. 88-95.

[3] Gemmell, Jim, Williams, Lyndsay, Wood, Ken, Lueder, Roger, and Bell, Gordon, Passive Capture and Ensuing Issues for a Personal Lifetime Store, First ACM Workshop on Continuous Archival and Retrieval of Personal Experiences (CARPE04).

[4] Manber, Udi, Finding Similar Files in a Large File System, *USENIX Winter Technical conference*, January, 1994, CA.

[5] Andrei Z., Broder, Some applications of Rabin's fingerprinting method. In R. Capocelli, A. De Santis and U. Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, Springer Verlag, 1993.

[6] Andrei Z., Broder, Identifying and Filtering Near-Duplicate Documents, COM '00: Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching. pages 1-10, Springer-Verlag, 2000.

[7] Abdur Chowdhury, O.Frieder et al. Collection statistics for fast duplicate document detection." *ACM Transaction on Information Systems* **20**(2): 171-191 2002

[8] Ifilter interface. Microsoft documentation. http://msdn.microsoft.com/library/en-us/indexsrv/html/ixrefint_9sfm.asp