Scalability in Recursively Stored Delta Compressed Collections of Files

Angelos Molfetas[1] **Anthony Wirth**[1] **Justin Zobel**[1]

[1] Department of Computing and Information Systems
The University of Melbourne
Email: {angelos.molfetas, awirth, jzobel}@unimelb.edu.au

Abstract

The archiving and maintenance of vast quantities of data is a key challenge for the current use of information technology. When storing large repositories, possibly mirrored at multiple sites, an archiving system aims to reduce both storage and transmission costs. *Delta compression* is a key component of many archiving and backup systems. A file may be stored succinctly as a sequence of references to other files in the collection, establishing a dependency relationship between files. On the one hand, exploiting large dependency chains provides excellent compression. On the other hand, if a file is stored compactly, so that it depends on hundreds of other files, then retrieving it from the archive may be very time and resource consuming.

This paper assesses the scalability of delta compression of typical data collections. We use experiments to model and examine the dependency relationship, and quantify the cost of full use of dependencies. We propose strategies to reduce dependencies and yet retain highly effective compression.

Keywords: Differential compression, repository compression, scalability, delta files.

1 Introduction

The proliferation of Web services and cloud computing necessitates centrally managed and distributed stored repositories of large collections of files. Examples of such large data collections are versioned textual content (e.g. in the case of wiki sites, such as Wikipedia[1]), emails (in the case of Web-based email systems, such as gmail[2]), and source code (in the case of Web-enabled repository sites, such as Github[3]and Bitbucket[4]).

Repositories of source code and executables are mirrored between countries; collections, such as genetic databases, are distributed to scientific institutions; companies duplicate data so that the people at each site have a local copy; applications such as search engines maintain their indexes at multiple regional data centres.

In enabling these Web-based applications, the repositories should be replicated, or partially replicated, in an efficient and scalable manner. Furthermore, these large datasets often have large amounts of redundancy, allowing for efficient transport and storage of data. The aim of scalable and efficient transport and storage raises numerous relevant questions. For example, given a repository and a query file, can we find (a specified number of) files in the repository that are the closest to the query, assuming some definition of file similarity? Or, given two repositories, can we estimate the communication required to compress the information required to synchronize the repositories? In Web scale systems, these types of questions are very difficult to answer in a manner that scales with the increasing quantities of data being produced.

The cost of sending, say, an executable to a machine over the Internet can be greatly reduced if it is possible to first identify large parts of the executable that have already been delivered as components of other programs.

Closely related to *differential compression* is *data deduplication*. A simple example of *deduplication* is the removal of an identified duplicate file from a collection, replacing it with a hard link, with the same name, pointing to the inode for the original file. Duplicate files are usually identified through some hashing technique.

Companies offering mass storage often deduplicating their data. For example, the online file hosting service `Dropbox` identifies identical files or pieces of files, and stores a single copy[5] (presumably before initiating replication for redundancy). Though it is similar to *deduplication*, in contrast, *differential compression* may in a single file object interleave references to duplicate data with non-duplicate data.

Many organizations keep a record of a large-scale Web crawl for search and mining. Plausibly, several organizations may share the results of several crawls (Suel et al. 2004). The task of our algorithms, therefore, would be to ensure that the updates that each crawl finds are transmitted to the other sites. The Stanford WebBase project is an example of this approach (Hirai et al. 2000).

[1]`wikipedia.org`
[2]`mail.google.com`
[3]`github.com`
[4]`bitbucket.org`
[5]`blog.dropbox.com/2011/07/changes-to-our-policies`

Differential Compression *Differential compression* is a method for concisely encoding files by taking advantage of their similarity to other files (Ajtai et al. 2002). It is a particularly effective, and a commonly applied, technique for the transfer of versioned data, for example by synchronization tools such as `rsync`[6] (Tridgell & Mackerras 1996). *Differential compression* has been used in a cache-based technique for optimized Web transfers distribution of content to Web clients (Chan & Woo 1999), for IP-level network requests (Spring & Wetherall 2000), and for Web traffic (Mogul et al. 1997).

Simple *two-file differential compression* involves encoding a *target file* with respect to a *reference file*. The *reference* is typically (but not necessarily) an older version of the *target file*. This process looks for matching strings between the *target* and *reference files* and produces a *delta file*.

When decoded in the presence of a *reference file*, the *delta file* spawns the *target file*. This *delta file* is composed of two types of instructions: *copies* that specify that a matching strings be copied from the reference file, and *adds* that specify verbatim a string (that was not found in the *reference file*) and should be spliced into the reconstructed *target file*. A standard approach to construct delta files is the Bentley & McIlroy (1999) hash-table dictionary scheme for finding long common strings.

In addition to encoding a single file against another file, *differential compression* can also be used to compress a collection of documents. This differs from the common *combine and compress* approach, which merges all files into a single file, with `tar` or similar, and then compresses the resultant file with a compression utility, such as `gzip`[7] or `7zip`[8]. When compressing a collection using *differential compression*, the key objective is to exploit repeating occurrences in different files while at the same time being able to access them atomically, without having to decompress the whole collection (Peel et al. 2011).

Though differential compression can make the most of limited storage space, it also poses scalability problems. Principally, a *delta file* could refer to several *reference files*, which themselves are stored as *deltas*, in turn linking to other *reference files*, and so forth. Therefore, to restore a single file to its original *target* form, the system might need to resolve a large number of dependencies, and thus access many many files.

Related Work In the context that nearby files may be unrelated, Bhagwat et al. (2009) exploit file similarity, rather than locality, to build a scalable file-backup system. Here, chunks of size 4 to 8 kB are cryptographically hashed to detect inter-file duplication. Echoing some of the considerations in this paper, Min et al. (2011) consider how to divide an index of fingerprints, and which replacement policy to invoke, for the backup of multimedia files.

PRESIDIO is storage framework for immutable archives (You et al. 2011). It incorporates multiple different compression techniques, including chunking and delta compression, and develops alternatives to standard clustering techniques. You et al. explore the impact of *delta chain length*, one of the central issues in scalability, and a focus of this paper. They show that long delta chains, in which *delta files* refer to other *delta files*, provide massive increases in compression effectiveness. However, as we describe in detail in Section 3, resolving long delta chains burdens disk and CPU heavily. Different from those in this paper, You et al. present some strategies for reducing delta chain length.

[6] `rsync.samba.org/ftp/rsync/rsync.html`
[7] `gzip.org`
[8] `7-zip.org`

Outline This paper continues as follows. Section 2 presents a taxonomy of the different ways that *differential compression* can be applied. Section 3 discusses scalability problems of some *differential compression* schemes and some known solutions. Experiments demonstrating the problems with a simple *differential compression* approach follow in Section 4. Resolutions to these, as well as further experiments, appear in Section 5. The paper concludes in Section 6.

2 Types of Differential Compression

Differential compression schemes can be classified into four broad categories: *two-file*; *serial*; *collection-based, single dependency*; and *collection-based, multiple dependencies*.

Two-File This scheme is the simplest, involving only a single *delta file*, which encodes a *target file* with respect to a single *reference file*. The *target file* can be recreated from the *delta file* in the presence of the *reference file* (Figure 1). This scheme is a good choice when one wants to transfer a large *target file* to a remote location containing a copy of the *reference file* that is very similar to the *target file*; in this situation, one can send a much smaller *delta file* incorporating the differences between the two files.

Examples of tools of this kind are `xdelta`[9], `vdelta` (MacDonald 2000), and `zdelta` (Trendafilov et al. 2002).

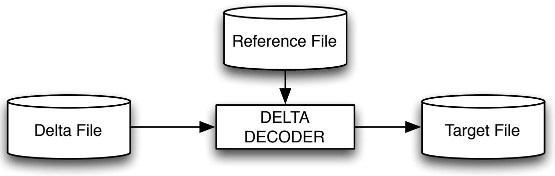


Figure 1: Two-file differential compression.

Serial Here, *delta files* are encoded sequentially; most of the *reference files* are themselves stored as *delta files*, differencing from some previous version. Thus, the process of recovering a *target* could involve decoding multiple *reference files* (Figure 2). *Serial differential compression* can be implemented by repeated application of a *two-file* tool. For example, `xdelta` could recreate a *target file* from a *delta file*, then the

[9] `xdelta.org`

recreated target file becomes the *reference file* for the next *delta file*.

Some types of Version Control System (VCS), such as SCCS (Rochkind 1975), implement a *serial differential compression* scheme.

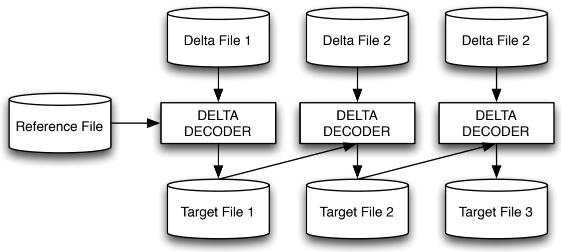


Figure 2: Serial differential compression.

Collection-Based, Single Dependency *Collection-based* differential compression is applicable to collections comprised of files that are related or have similar content, but are not sequences of versions. When a file is added, it is encoded with respect to another file in the collection, which in turn may be a *delta file* encoded with respect to some other file in the collection. This approach is very similar to the serial approach described in Section 2, but differs in that branching between *delta files* can occur (Figure 3).

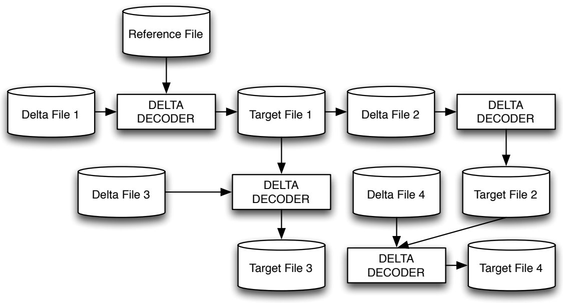


Figure 3: Collection-based differential compression, single dependencies.

Nevertheless, each file depends on a *single reference*. With single dependencies in collection-based *differential compression*, one can (again) apply a *two-file* tool repeatedly to build a compressed collection.

The key challenge with type of compression is to identify sufficiently similar pairs of files to differentially compress. Manber (1994) presented a method for finding similar files in a large file system, and suggested that this method could be used for data compression. Following this work, Douglis & Iyengar (2003) showed how to apply hash (or fingerprint) techniques to detect resemblance amongst a collection of files of disparate kinds in order to *differentially compress* them. Ouyang et al. (2002) clustered files by similarity to optimize branching, which are then compressed using the *two-file* `zdelta` utility.

Collection-Based, Multiple Dependencies Finally, a file repository can also be stored as a collection of *delta files*, each of which has multiple dependencies (Figure 4).

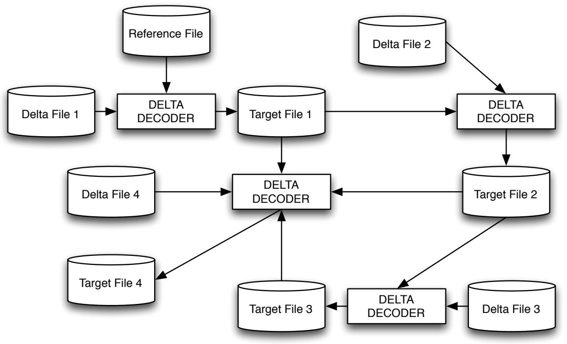


Figure 4: Collection-based differential compression, many dependencies.

In contrast to encoding with single dependencies, a *two-file* delta coding tool is not sufficient for this task. Moreover, from multiple dependencies arise significant scalability concerns. These are the central focus of this paper.

3 Scalability Issues

In *serial differential compression*, when encoding a new *delta file*, there is no need to decide which existing *delta file* should be chosen as a *reference*. As a new version is serially added to the collection, the previous version *delta* is simply selected as the *reference*.

The problem with this technique is that decoding a *delta file* requires decoding all ancestor *delta files*. In the context of a VCS that employs this scheme, all revisions of a file must be decoded in order to read the latest revision. Consequently, the time needed to decode a file can increase significantly as that file's history grows. Some VCS systems, such as SVN's BDB back-end alleviate this problem by instead having the latest version of a file stored as the *reference file*, with deltas for previous revisions. Generally, it is less likely that a developer requires a revision the older it becomes. The weakness of this approach is that each time a new revision is created, all the previous revisions would have be recalculated.

SVN solves the problem of decoding multiple *delta files* by introducing *skip deltas*. By skipping some revisions, the *skip delta* technique attempts to reduce the number of dependency *delta files* to be accessed. It determines the predecessor revision by taking the revision number in binary representation, and flips the rightmost bit that has a value of 1. As Figure 5 shows, this scheme provides pathways that require fewer revisions to be decoded: shorter chains. When there are N revisions, this technique limits the maximum number of necessary decodings of a *delta file* to $\log_2 N$; there is a space penalty of $O(\log N)$, but an $O(N/\log N)$-time benefit[10].

Finding the Optimum Reference File The main challenge in single-dependency *collection-based differential compression* is finding an appropriate (existing) *delta file* to act as *reference file* to a newly added *delta file*. Ideally, one wants to choose a *delta file* that (in

[10] `svn.apache.org/repos/asf/subversion/trunk/notes/skip-deltas`

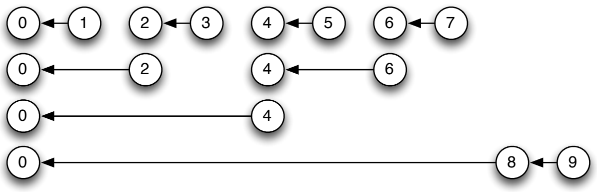


Figure 5: SVN skip-deltas (based on illustration from SVN documentation). Numbers represent revisions.

its decoded form) is the most similar to the *target file* being added to the collection.

Ouyang et al. (2002) observed that finding the optimal delta encodings between files in a collection, with single dependencies, can be reduced to finding the maximum-weight branching in a (edge-weighted) directed graph. The graph models the situation thus.

1. A node of the graph represents a *target file* that is to be added to the collection.
2. The weight of an edge from node r to node t is the compression saving obtained when *target file* t is delta encoded with respect to *reference file* r.
3. There is a *null node*, which has no edges directed towards it.
4. The weight for the edge originating from the *null node* to another node f represents the compression benefit obtained if the file f were compressed with respect to itself.

Figure 6 shows an illustration of the possible pairwise compression arrangements.

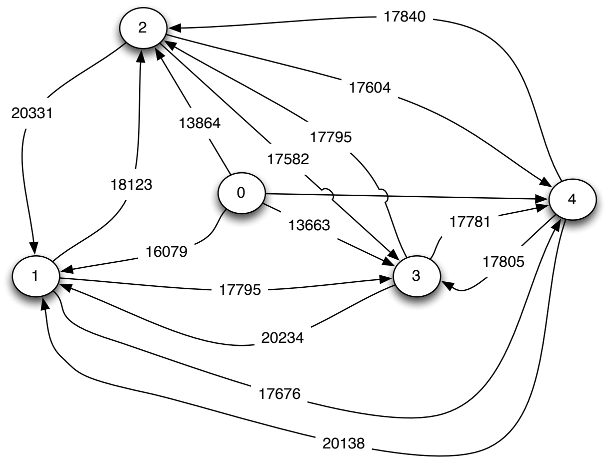


Figure 6: Directed weighted graph showing compression savings between pairwise delta encodings (based on an illustration by Ouyang et al. (2002)).

Finding an optimal branching in the directed graph provides the optimal compression possible from a *two-file* tool. However, as this approach does not scale to large collections of files, Ouyang et al. (2002) experimentally compared a number of file-clustering heuristics. They achieved significant compression improvements compared to concatenating the files in the collection and then applying the `gzip` tool.

Recursive Dependencies An example of a proposed system that has *delta files* with multiple dependencies is Chan & Woo's (1999) cache-based technique for optimized Web transfers. This system works by having the server encode the *target file* according to a selected set of *reference files* to generate a *delta file*; this *delta* is then transmitted to the client, which has copies of the *reference files* in its cache. The system scales well for two reasons: first, the number of *reference files* is bounded by the selection heuristic; second, the *delta file* is only for transfer, and is decoded immediately at the client. The *reference files* are not stored as *delta files*, and there is no need to recursively decode all dependent *delta files* to access the file transferred.

Delta decoding scalability becomes an issue when a repository both stores a collection as *delta files*, and uses these *delta files* as *reference files* for files that are added in encoded form. Consequently, there are long chains of dependencies, which (only) grow as more files arrive in the collection.

Multiple References When the *differential compression* encoding scheme allows for a *delta file* to have multiple *reference files*, the recursive dependency problem becomes infeasible except in cases where one wishes to decode the entire collection of files at once. The multiple files introduce a multiplier effect for each level of encoding recursion. We explore this further in the next section.

4 Experimental Illustration

We explore in detail experimentally, and propose solutions to, some of the scalability issues raised in the previous section. We ascertain the scalability problems in multiple-dependency *collection-based differential compression*. The experiment incrementally builds a differentially compressed collection where all files are stored as *delta files*.

First, a hash-based dictionary is instantiated, where each entry stores a file identifier and offset. As each file is added to the collection, it is assigned a unique file identifier and then scanned and fingerprints are generated using the Karp & Rabin (1987) rolling hashing method. Each hash is then added into the hash table, such that it is associated with its corresponding file identifier and position within the file.

There are two approaches to storing hashes in the dictionary. The first stores all hashes, while the second stores only each sequential non-overlapping hash in the dictionary, disregarding the rest (Bentley & McIlroy 1999). The latter is a simple way to reduce the size of the dictionary, though compromises the detection of smaller strings. This study takes the first approach.

When matches are identified, the matching substrings are replaced with *copy instructions* that specify a file identifier, an offset to begin copying and the number of bytes to copy. All other strings, those not matched, are 'encoded' as *add instructions*, which include the unmatched strings verbatim. The algorithm has a recursive decoding function that reads each *delta file* referenced by each *copy instruction*, and

Table 1: Summary of datasets

Dataset	# Files	Total Bytes
CSDMC Training	4327	27,913,226
CSDMC Testing	4292	27,265,343
Enron Corpus	14,357	41,084,874
Linux Kernel 3.9.2	42,412	479,610,289
Python 3.3.2	3785	65,582,523
Ruby 2.0.0-p195	4079	63,919,656
Matplotlib 1.2.1	4122	62,104,168

decoding recursively until no more *copy instructions* are found. Finally, once a *target file* is successfully added to the collection and stored as a *delta file*, as a validation step it is once again extracted by converting it into its original expanded form. The experiment records the direct dependencies that each *delta file* has, that is which *reference files* it refers to in its *copy* instructions. The experiment then determines the total number of dependencies, direct or recursive, required to decode the *delta file*.

Datasets The Enron Corpus dataset has about 500,000 emails from Enron's senior management[11] (Klimt & Yang 2004). It was released to the public by the United States Federal Energy Regulatory Commission.

The CSDMC dataset was designed for spam classification experiments. Its curators partitioned it into a testing and training set[12].

We also test against development repositories that are composed principally of source code, as well as some technical documentation. The examples are the Linux kernel, Python and Ruby interpreters, and the `matplotlib` plotting package (Hunter 2007), which incidentally produced the plots in this paper. Table 1 gives an overview of the datasets.

It should be noted that these datasets are treated in the experiments as collections of similar independent files. That is, email headers and file names are not used to infer relationships between files in order to choose *reference files*. This is because the paper is investigating *multiple dependency collection-based* repositories for similar files, rather than for example, compressing the datasets as a set of *serial* or *single dependency collection-based* repositories.

4.1 Results

Figure 7 shows the average number of direct dependencies of *delta files* that are added into the email repositories. The number of dependencies rapidly increases as the hash table is filled; then, as the hash table is saturated, the average number of dependencies levels off at under 4.5 for all three datasets.

This number of directly connected files appears not to be very high. However, due to the multiplicative effect at every level of recursion, it means that opening a single file requires a very large number of other files be opened. Worse still, this number of total dependencies constantly increases, and seems to be linear in the size of the collection (Figure 8).

[11] www.cs.cmu.edu/~enron/

[12] csmining.org

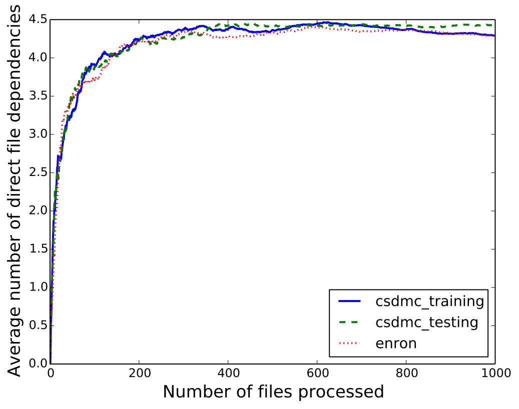


Figure 7: Number of direct dependencies for email datasets, with 32-byte hashing.

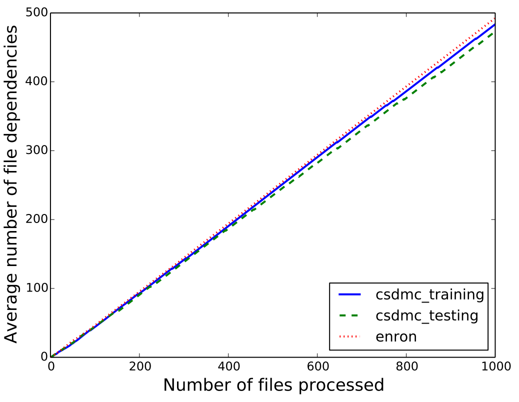


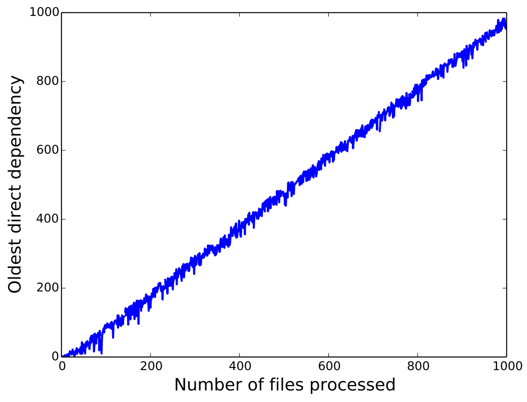
Figure 8: Total count of dependencies for email datasets, with 32-byte hashing.

This linear relationship is acceptable if one intends to decompress the entire collection in insertion order. However, it negates the advantage that collection-based *differential compression* has over *combine and compress* methods. To access a single file, one needs to decompress all, or at least a very large portion, of the collection.

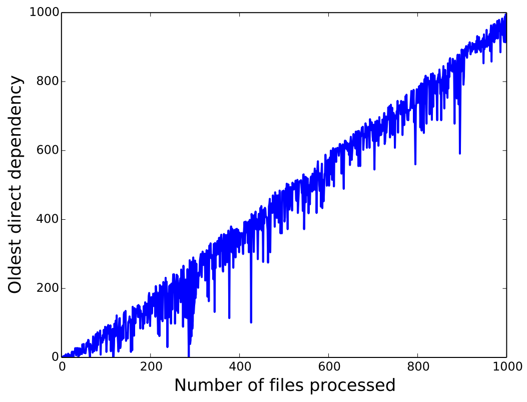
The linear blowout in dependence is not intrinsically due to newer *delta files* having a large number of files to choose to refer to. It arises, as a consequence of the design of the fingerprint hash table, which favors dependencies to recent files. Figures 9a and 9b demonstrate that, as files are inserted, newer files replace the older ones as *reference files*.

So far, we have examined only the properties of collections of emails. On the source file datasets, the dependency structure seems far less stable (Figures 10a and 10b). The Ruby dataset is similar to the email collection, though with fewer than two direct dependencies on average per file. On the other hand, at various points, as files are added, the number of dependencies in the Linux kernel collection seems to increase quite suddenly, possibly due to the large variation in file type.

The `matplotlib` collection appears to have very high direct dependency initially, then tends towards

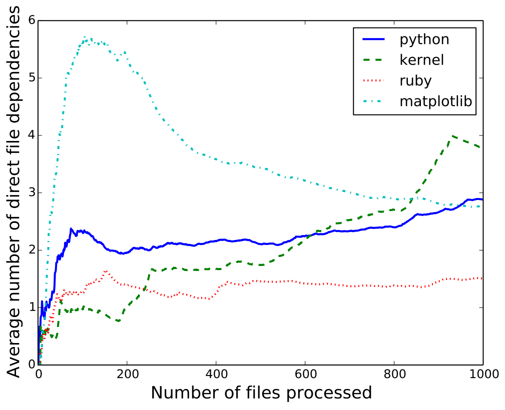


(a) CSDMC Testing dataset.

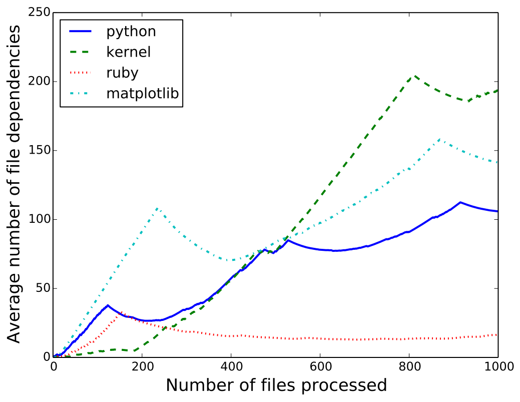


(b) Enron dataset.

Figure 9: Oldest direct dependency as a function of number of files inserted, with 32-byte hashing.



(a) Direct dependencies.



(b) Total dependencies.

Figure 10: Number of dependencies for source datasets, with 32-byte hashing.

a count more like the email collections.

The instability in the increase of dependencies in the source datasets suggests that more dissimilar datasets are advantageous for *multiple dependency collection-based* repositories in terms of reducing the large dependency chains, though this would come with a lower overall compression trade-off.

Though we leave it for future study, it would be interesting to investigate how the dependency relationship varies according to the order in which the files are added to the collection. For instance, is a randomized order helpful?

5 Resolving the Scalability Issue

Increasing Hashing Block Size To reduce the number of dependencies, our first strategy is increasing the fingerprinting block size. This increase forces the algorithm to find fewer direct dependencies and larger matches. Figures 11a and 11b show the results of changing from 32 to 64 and 128-byte blocks.

However, this hardly reduces the overall number of dependencies (Figures 12a and 12b). For the CSDMC datasets, it reduces the rate of increase, but the problem still persists. Despite the block size increase, added files still tend to refer to recent files (Figures 13a and 13b).

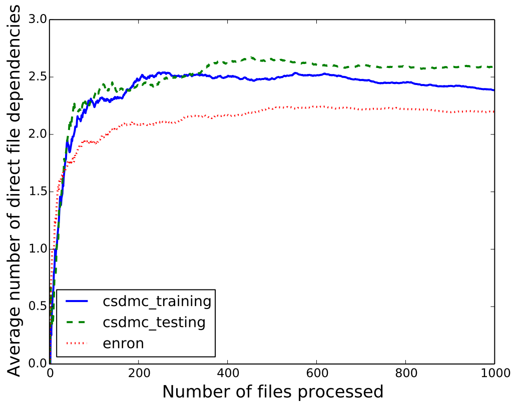
Increasing Size of Hash Table The next strategy to reduce dependency chains is to increase the size of the hash *table* itself. A larger hash table is more likely to have older entries: this encourages references to older files, requiring a shorter chain of dependencies. So we repeated the experiment with 32-byte hashes, but this time with a much larger hash table (2,097,143 vs 32,749 bytes).

Unfortunately, there is still a linear increase in the total number of dependencies (Figure 14a). Even though the *differential compression* algorithm now builds references to much 'earlier' files (Figure 14b), it seems to find direct dependencies to more files (compare Figure 14c with Figure 7).

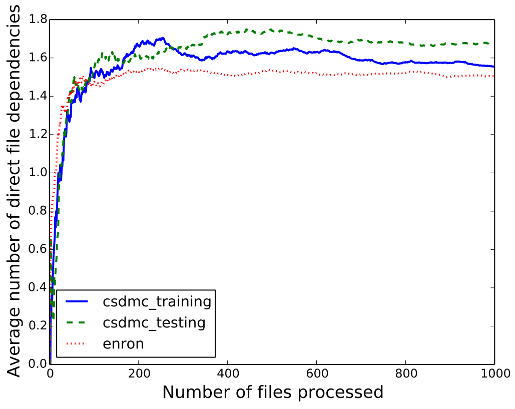
Furthermore, files inserted in close proximity to each other are generally more likely to be similar in content, and this effect cannot be neutralized by larger hash tables.

Hash Table Replacement Policy So far, the hash table policy has been that newer entries displace older entries. This encourages *delta files* to be encoded against more recently added files, which increases the number of dependencies. Instead, in the next experiment, the *differential compression* algorithm will favor older hash table entries.

Figure 15a shows the number of direct dependencies for the Enron dataset. Figure 15b shows that, in contrast to previous experiments, the curve

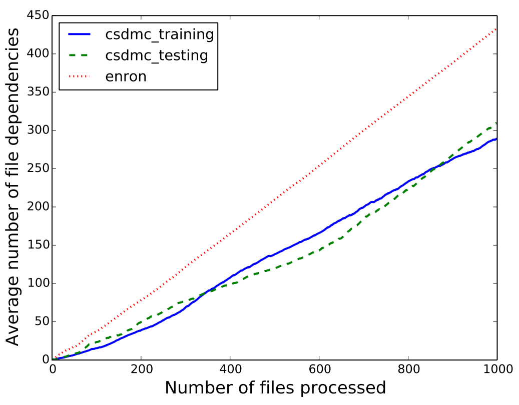


(a) Email datasets with 64-byte hashing.

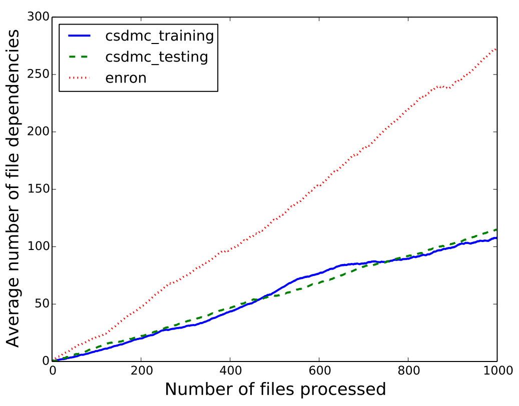


(b) Email datasets with 128-byte hashing.

Figure 11: Number of direct dependencies for larger block size.



(a) Email datasets with 64-byte hashing.



(b) Email datasets with 128-byte hashing.

Figure 12: Total number of dependencies for larger hash blocks.

for the total number of dependencies becomes sub-linear. Finally, scalability has improved significantly. Newer files are linking directly to very old files (Figure 15c), thus significantly reducing chain length.

The point at which the curves start flattening depends on dictionary size. In this experiment, the hash table dictionary had 131,071 entries.

Though this policy change appears to resolve the issue of scalability, it does so only by preventing the algorithm from incorporating new information. In some sense, the hash table for the file collection has become stale.

A more sophisticated alternative would cluster the files based on similarity and allocate a separate hash tables per group. This would constrain the number of files each file could refer to, but at the same time ensure that it refers to the most relevant content. It seems more appealing than simply restricting the compression process to older content.

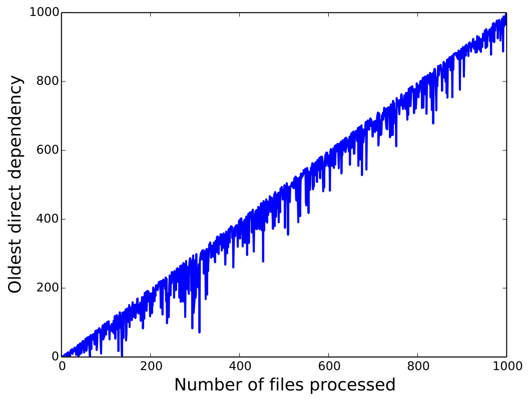
6 Conclusion

By taking advantage of high levels of redundancy that exist between files, and by storing files in their collections as *delta files*, repositories that incorporate *differential compression* store data efficiently. There are two key approaches. First, files can be compressed in a pairwise manner using a *two-file* compression tool. The challenge with this approach is ascertaining a method that can find an appropriately similar file in the collection. This method should ideally take constant or sub-linear time in the number of files in the collection.

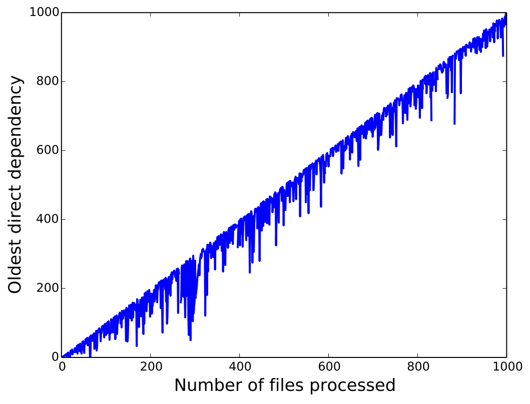
The second approach involves a hash-table dictionary to find components of the new file that match components of files already in the collection. Once added, the new file takes the form of a *delta file* that 'links' to multiple *reference files*. When a *delta file* has multiple dependencies, decompressing it might require recursively decoding many files. Worse still, the number of dependent *delta files* increases linearly with the number of files in the collection. The same problem exists with serially encoded *differentially compressed* collections, such as VCS systems; in this case, however, the decoding recursion is limited to revisions of one file. Similarly, with *differentially compressed* collections of *delta files* with single dependencies, as files are encoded against similar *reference files*, the problem would be constrained to each cluster of files.

For *collection-based* compression with multiple *reference files*, the compression algorithm has no constraints, and consequently the number of dependencies increases with the size of the collection.

If one wishes to decompress the entire the whole collection, as is the case with *combine and compress* methods, then this high level of *delta file* dependency is not a problem. However, in the context of Web-



(a) Enron dataset with 64-byte hashing.



(b) Enron dataset with 128-byte hashing.

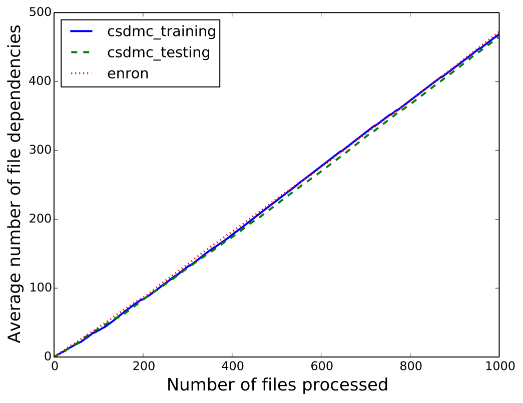
Figure 13: Oldest direct dependency for larger hash blocks.

based systems needing to access files within compressed collections, except when data is no longer accessed and archived, these high levels of dependencies between *delta files* are infeasible.

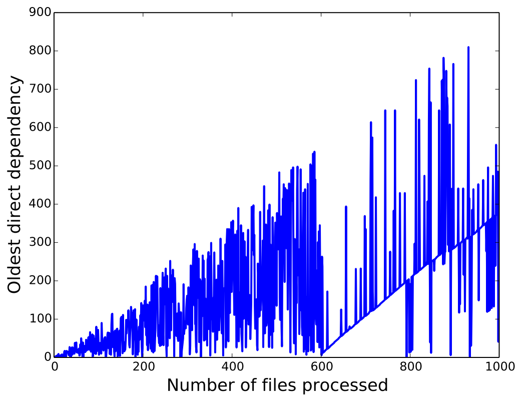
As we have shown, the number of dependencies can be constrained by increasing the size of the dictionary and applying a selection strategy that favors older entries over new. This, however, means that newly added files may not be future *reference file candidates*.

Collections composed of heterogeneous types of files, when stored in *Multiple dependency collection-based* repositories, generate smaller chains of file dependencies. In such cases, however, there would also be a reduced compression trade-off.

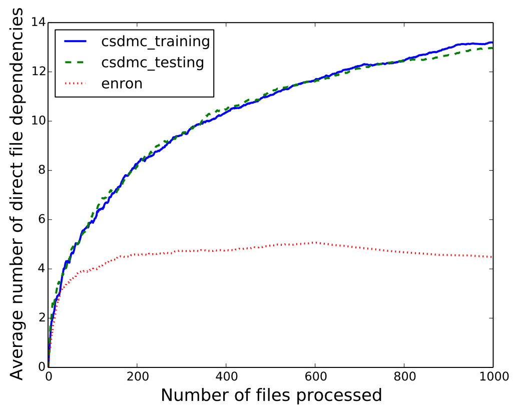
As future work, we recommend a clustering algorithm that applies a separate hash table dictionary to each cluster, thus constraining the dependency to that cluster of *delta files*.



(a) Total number of dependencies for email datasets.



(b) Oldest dependency for the Enron dataset.



(c) Number of direct dependencies for the Enron dataset.

Figure 14: Results for larger hash tables.

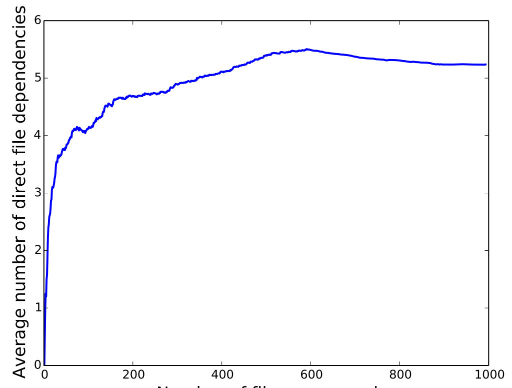
References

Ajtai, M., Burns, R., Fagin, R., de Long, D. & Stockmeyer, L. (2002), 'Compactly encoding unstructured inputs with differential compression', *Journal of the ACM* **49**(3), 318–67. doi: 10.1145/567112.567116.

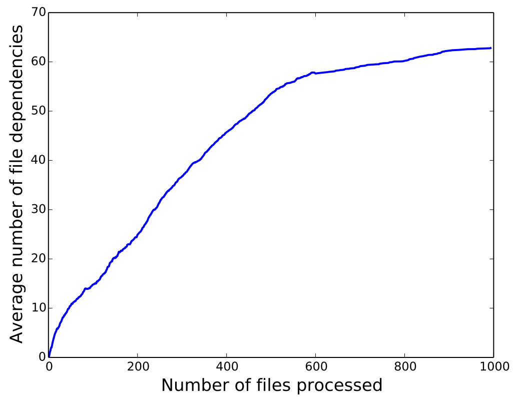
Bentley, J. & McIlroy, D. (1999), Data compression using long common strings, *in* 'DCC '99: Proc. IEEE Data Compression Conference', pp. 287–295. doi: 10.1109/DCC.1999.755678.

Bhagwat, D., Eshghi, K., Long, D. D. E. & Lillibridge, M. (2009), Extreme binning: Scalable, parallel deduplication for chunk-based file backup, *in* 'MASCOTS '09: Proc. IEEE Symp. Modeling, Analysis Simulation of Computer and Telecommunication Systems', pp. 1–9. doi: 10.1109/MASCOT.2009.5366623.

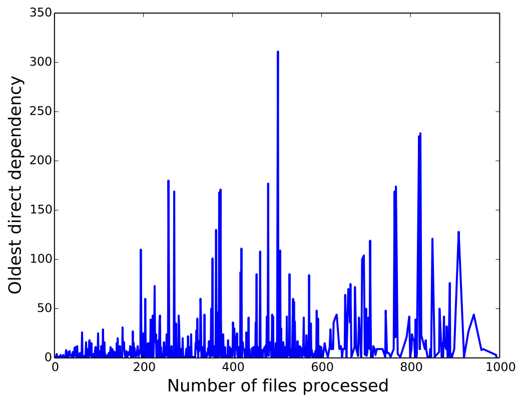
Chan, M. C. & Woo, T. Y. C. (1999), Cache-based compaction: A new technique for optimizing Web transfer, *in* 'INFOCOM '99: Proc. Joint Conf. IEEE



(a) Number of direct dependencies.



(b) Total number of dependencies.



(c) Oldest dependency.

Figure 15: Results for change in hash-table policy.

Computer and Communications Societies', Vol. 1, pp. 117–125. doi: 10.1109/INFCOM.1999.749259.

Douglis, F. & Iyengar, A. (2003), Application-specific delta-encoding via resemblance detection, *in* 'USENIX '03: Proc. USENIX Ann. Technical Conf.', pp. 113–26.

Hirai, J., Raghavan, S., Garcia-Molina, H. & Paepcke, A. (2000), 'Webbase: a repository of web pages', *Computer Networks* **33**, 277–93. doi: 10.1016/S1389-1286(00)00063-3.

Hunter, J. D. (2007), 'Matplotlib: A 2d graphics environment', *Computing in Science & Engineering* **9**(3), 90–95.

Karp, R. M. & Rabin, M. O. (1987), 'Efficient randomized pattern-matching algorithms', *IBM J. Res. Dev.* **31**(2), 249–260. doi: 10.1147/rd.312.0249.

Klimt, B. & Yang, Y. (2004), The Enron corpus: A new dataset for email classification research, *in* 'ECML '04: Proc. Eur. Conf. Machine Learning', pp. 217–226. doi: 10.1007/978-3-540-30115-8_22.

MacDonald, J. (2000), File system support for delta compression, Master's thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley.

Manber, U. (1994), Finding similar files in a large file system, *in* 'Proceedings of the USENIX Winter 1994 Technical Conference', WTEC'94, pp. 1–10.

Min, J., Yoon, D. & Won, Y. (2011), 'Efficient deduplication techniques for modern backup operation', *IEEE Trans. Computers* **60**(6), 824–840. doi: 10.1109/TC.2010.263.

Mogul, J. C., Douglis, F., Feldmann, A. & Krishnamurthy, B. (1997), 'Potential benefits of delta encoding and data compression for HTTP', *SIGCOMM Comput. Commun. Rev.* **27**(4), 181–194. doi: 10.1145/263109.263162.

Ouyang, Z., Memon, N., Suel, T. & Trendafilov, D. (2002), Cluster-based delta compression of a collection of files, *in* 'WISE '02: Proc. Conf. Web Information Systems Engineering, 2002', pp. 257–266. doi: 10.1109/WISE.2002.1181662.

Peel, A., Wirth, A. & Zobel, J. (2011), Collection-based compression using discovered long matching strings, *in* 'CIKM '11: Proc. ACM Int. Conf. Information and Knowledge Management', CIKM '11, pp. 2361–2364. doi: 10.1145/2063576.2063967.

Rochkind, M. J. (1975), 'The source code control system', *IEEE Tr. on Software Engineering* (4), 364–370. doi: 10.1109/TSE.1975.6312866.

Spring, N. T. & Wetherall, D. (2000), A protocol-independent technique for eliminating redundant network traffic, *in* 'SIGCOMM '00: Proc. Conf. Applications, Technologies, Architectures, and Protocols for Computer Communication', pp. 87–95. doi: 10.1145/347059.347408.

Suel, T., Noel, P. & Trendafilov, D. (2004), Improved file synchronization techniques for maintaining large replicated collections over slow networks, *in* 'ICDE '04: Proc. Int. Conf. Data Engineering', pp. 153–64. doi: 10.1109/ICDE.2004.1319992.

Trendafilov, D., Memon, N. & Suel, T. (2002), zdelta: An efficient delta compression tool, Technical Report TR-CIS-2002-02, Polytechnic University.

Tridgell, A. & Mackerras, P. (1996), The rsync algorithm, Technical Report TR-CS-96-05, The Australian National University.

You, L. L., Pollack, K. T., Long, D. D. E. & Gopinath, K. (2011), 'PRESIDIO: A framework for efficient archival data storage', *ACM Trans. Storage* **7**(2), 6:1–6:60. doi: 10.1145/1970348.1970351.