D

Delta Compression Techniques

Torsten Suel
Department of Computer Science and Engineering, Tandon School of Engineering, New York University, Brooklyn, NY, USA

Synonyms

Data differencing; Delta encoding; Differential compression

Definition

Delta compression techniques encode a target file with respect to one or more reference files, such that a decoder who has access to the same reference files can recreate the target file from the compressed data. Delta compression is usually applied in cases where there is a high degree of redundancy between target and references files, leading to a much smaller compressed size than could be achieved by just compressing the target file by itself. Typical application scenarios include revision control systems and versioned file systems that store many versions of a file or software or content updates over networks where the recipient already has an older version of the data. Most work on delta compression techniques has focused on the case of textual and binary files, but the concept can also be applied to multimedia and structured data.

Delta compression should not be confused with Elias delta codes, a technique for encoding integer values, or with the idea of coding sorted sequences of integers by first taking the difference (or delta) between consecutive values. Also, delta compression requires the encoder to have complete knowledge of the reference files and thus differs from more general techniques for redundancy elimination in networks and storage systems where the encoder has limited or even no knowledge of the reference files, though the boundaries with that line of work are not clearly defined.

Overview

Many applications of big data technologies involve very large data sets that need to be stored on disk or transmitted over networks. Consequently, data compression techniques are widely used to reduce data sizes. However, there are many scenarios where there are significant redundancies between different data files that cannot be exploited by compressing each file independently. For example, there may be many versions of a document, say a Wikipedia article, that differ only slightly from one to the next. Delta compression techniques attempt to exploit such redundancy between pairs or groups of files to achieve better compression.

S. Sakr, A. Zomaya (eds.), *Encyclopedia of Big Data Technologies*,
https://doi.org/10.1007/978-3-319-63962-8_63-1

We define delta compression for the most common case of a single reference file. Formally, we have two strings (files): $f_{tar} \in \Sigma^*$ (the target file) and $f_{ref} \in \Sigma^*$ (the reference file). Then the goal for an encoder E with access to both f_{tar} and f_{ref} is to construct a file $f_\delta \in \Sigma^*$ of minimum size, such that a decoder D can reconstruct f_{tar} from f_{ref} and f_δ. We also refer to f_δ as the *delta* of f_{tar} and f_{ref}.

Given this definition, research on delta compression techniques has focused on three challenges: (1) How to build delta compression tools that result in f_δ of small size while also achieving fast compression and decompression speeds and small memory footprint. (2) How to use delta compression techniques to compress collections of files, i.e., how to select suitable reference files given a target file, or how to select a sequence of delta compression steps between files to achieve good compression for a collection ideally while also allowing fast retrieval of individual files. (3) How to apply delta compression in various application scenarios.

Key Research Findings

String-to-String Correction and Differencing

Some early related work was done by Wagner and Fisher (1974), who studied the *string-to-string correction problem*. This is the problem of finding the best (shortest) sequence of insert, delete, and update operations that transform one string f_{ref} to another string f_{tar}. The solution in Wagner and Fisher (1974) is based on finding the longest common subsequence of the two strings using dynamic programming and adding all remaining characters to f_{tar} explicitly. However, the string-to-string correction problem does not capture the full generality of the delta compression problem. In particular, in the string-to-string correction problem, it is implicitly assumed that the data common to f_{tar} and f_{ref} appears in (roughly) the same order in the two files, and the approach cannot exploit the case of substrings in f_{ref} appearing repeatedly in f_{tar}.

This early work motivated Unix tools such as *diff* and *bdiff*, which create a *patch*, i.e., a set of edit commands, that can be used to update the reference file to the target file. Such tools are widely used to create patches for software updates, and one advantage is that these patches are easily readable by humans who need to understand the changes. However, the tools typically do not generate delta files of competitive size, due to (1) limitations in the type of edit operations that are supported, as discussed in the previous paragraph, and (2) the absence of native compression methods for reducing the size of the patch. The latter issue can be partially addressed by applying general purpose file compressors such as *gzip* or *bzip* to the generated patches, though this is still far from optimal.

The first problem was partially addressed by Tichy (1984), who defined the *string-to-string correction problem with block moves*. For a file f, let $f[i]$ denote the ith symbol of f, $0 \leq i < |f|$, and $f[i, j]$ denote the block of symbols from i until (and including) j. Then a *block move* is a triple (p, q, l) such that $f_{ref}[p, \ldots, p + l - 1] = f_{tar}[q, \ldots, q + l - 1]$, representing a nonempty common substring of f_{ref} and f_{tar} of length l. The file f_δ can now be constructed as a minimal *covering set* of block moves, such that every $f_{tar}[i]$ that also appears in f_{ref} is included in exactly one block move.

Tichy (1984) also showed that a greedy algorithm results in a minimal cover set, leading to an f_δ that can be constructed in linear space and time using suffix trees. However, the multiplicative constant in the space complexity made the approach impractical. A more practical approach uses hash tables with linear space but quadratic time worst-case complexity (Tichy 1984). Subsequent work in Ajtai et al. (2002) showed how to further reduce the space used during compression while avoiding a significant increase in compressed size, while Agarwal et al. (2004) showed how to reduce size by picking better copies than a greedy approach. Finally, Xiao et al. (2005) showed bounds for delta compression under a very simple model of file generation where a file is edited according to a two-state Markov process that either keeps or replaces content.

Delta Compression Using Lempel-Ziv Coding

The block move-based approach proposed in Tichy (1984) leads to a basic shift in the development of delta compression algorithms, as it suggests thinking about delta compression as a sequence of copies from the reference file into the target file, as opposed to a sequence of edit operations on the reference file. This led to a set of new algorithms based on the Lempel-Ziv family of compression algorithms (Ziv and Lempel 1977, 1978), which naturally lend themselves to such a copy-based approach while also allowing for compression of the resulting patches.

In particular, the LZ77 algorithm can be viewed as a sequence of copy operations that replace a prefix of the string being encoded by a reference to an identical previously encoded substring. Thus, delta compression could be viewed as simply performing LZ77 compression with the file f_{ref} representing previously encoded text. In fact, we can also include the part of f_{tar} that has already been encoded in the search for a longest matching prefix. A few extra changes are needed to get a practical implementation of LZ77-based delta compression. Implementations of this approach include vdelta (Hunt et al. 1998), several versions of *xdelta* (MacDonald 2000), *zdelta* (Trendafilov et al. 2002), the recent and very fast *ddelta* (Xia et al. 2014b) and *edelta* (Xia et al. 2015), and a number of implementations following the *vcdiff* differencing format.

A Sample Implementation

We now describe a possible implementation in more detail, using the example of *zdelta*. In fact, *zdelta* is based on a modification of the *zlib* library implementation of *gzip* by Gailly and Adler (Gailly 2017), with some additional ideas inspired by *vdelta* (Hunt et al. 1998). Note that *zlib* finds previous occurrences of substrings using a hash table. In *zdelta*, this is extended to several hash tables, one for each reference file and one for the already coded part of the target file. The table for f_{tar} is essentially handled as in *zlib*, where new entries are inserted as f_{tar} is traversed and encoded. The table for f_{ref} can be built beforehand by scanning f_{ref}, assuming f_{ref} is not too large. When looking for matches, all tables are searched to find the best match. Hashing of substrings is done based on the initial three characters, with chaining inside each hash bucket.

As observed in Hunt et al. (1998), in many cases, the location of the next match in f_{ref} is a short distance after the end of the previous match, especially when the files are very similar. Thus, the locations of matches in f_{ref} are usually best represented as offsets from the end of the previous match in f_{ref}. However, sometimes there are isolated matches in other parts of f_{ref}. This motivates *zdelta* to maintain two pointers into each reference file and to code locations by storing which pointer is used, the direction of the offset, and the offset itself. Pointers are initially set to the start of the file and afterward usually point to the ends of the previous two matches in the same reference file. If a match is far away from both pointers, a pointer is only moved to the end of it if it is the second such match in a row. Other more complex pointer movement policies might give additional moderate improvements.

To encode the generated offsets, pointers, directions, match lengths, and literals, *zdelta* uses the Huffman coding facilities provided by *zlib*, while *vdelta* uses a byte-based encoding that is faster but less compact.

Window Management in LZ-Based Delta Compressors

The above description assumes that reference files are small enough to completely fit into the hash table. However, this is usually not realistic. By default, *zlib* only indexes a history of up to 32 KB of already processed data in the hash table, in two blocks of 16 KB each. Whenever the second block is filled, the first one is freed up, temporarily reducing indexed history to 16 KB. This is for efficiency reasons: Indexing more than 32 KB would result in larger hash tables that lead to increased L1 cache misses (as L1 cache sizes have not increased much over the last two decades) and, more importantly, result in very long hash chains for common three-character substrings that would be traversed when searching for matches.

In the target file, we could just index the last up to 32 KB in the hash table by their first 3 bytes, as done in *zlib*. But which parts of the reference files should we index? There are several possible choices, as follows:

- **Sliding Window Forward:** In *zdelta*, a 32 KB window is initially placed at the start of the reference file and then moved forward by 16 KB whenever the median of the last few copies comes within a certain distance from the end of the current window. This works well if the content is reasonably aligned between the reference and target case, which is the common case in many applications, but can perform very badly when large blocks of content occur in a very different order in the two files.
- **Block Fingerprints:** Another possible approach chooses the parts in the reference files that should be indexed based on their similarity to the content currently being encoded. This is done by computing fingerprint for certain substrings and then indexing parts of the reference file that share many fingerprints with the content we are currently encoding. This approach was considered by Korn and Vo in the context of implementing *vcdiff* (Korn and Vo 2002).
- **Hashing Longer Substrings:** Another approach indexes the entire reference file and avoids long hash chains by hashing substrings much longer than three characters. This allows finding copies from anywhere in the reference and target files, as long as copies are longer than the hashed substrings. This approach was taken in earlier versions of *xdelta*, making it very robust against content reordering between the files, but compression may be slightly worse than other approaches on well-aligned files, since only large copies are supported. However, the approach can be combined with the others above, to allow both longer and shorter copies, by first using larger fixed-size chunks as proposed in Bentley and McIlroy (1999) and Tridgell (2000) or with content-defined chunking and then finding smaller copies more locally; see, e.g., *ddelta* (Xia et al. 2014b).

Compressing Collections of Files

Larger collections of files can be compressed by repeatedly applying delta compression to pairs or groups of files. This raises the question of what reference files to choose for which target files in order to achieve a cycle-free compression of the entire collection with minimum compressed size. In some scenarios, it is also crucial to be able to quickly decompress individual files without having to decompress too many other files.

In the case of revision control systems, older files are usually compressed using a newer version as a reference file, since newer versions are more frequently accessed. However, retrieving an old version could be very slow if it requires decompressing a long chain of more recent versions before reaching the one that is needed. For this reason, such systems may often create shortcuts, where versions are occasionally encoded using a much later version as reference, at the cost of a slight increase in size.

Delta compression can also be used to compress collections of non-versioned files that share some degree of similarity. For the case of a single reference file, finding an optimal and cycle-free assignment of reference files to target files can be modeled as an *optimum branching* problem on a complete graph (Tate 1997; Ouyang et al. 2002), which can be solved in $|V|^2$ time on dense graphs where $|V|$ is the number of vertices. For the case of more than one reference file, the problem is known to be NP-complete (Adler and Mitzenmacher 2001). However, the cost of computing the edge weights of the input graph is prohibitive, motivating (Ouyang et al. 2002) to propose using text clustering to identify for each target file a small set of promising reference file candidates. Work in Bagchi et al. (2006) showed that an optimum branching on a reduced degree graph of highest weight edges in fact provides a provable approximation ratio to the solution on the full graph.

Work by Molfetas et al. (2014b,a) studied how to optimize and trade off access speed and

compression ratio in delta-compressed file collections, e.g., by avoiding references to substrings in certain files when other copies could be used instead. It is shown that significant improvements are possible by judiciously picking copies during compression.

However, for bulk archival and retrieval of large collections of non-versioned web pages, work in Trendafilov et al. (2004), Chang et al. (2006), and Ferragina and Manzini (2010) suggests that better speed and compression can be achieved by applying optimized compression tools for archiving large data sets on top of a suitable linear ordering of the files, say one obtained via text clustering or URL ordering. The main reason is that such tools can identify repeated substrings over very long distances, while delta compressors are limited to one or a few reference files.

Examples of Applications

There are a number of scenarios where delta compression can be applied in order to reduce networking and storage cost. A few of them are now discussed briefly.

- **Software Revision Control Systems:** Delta compression techniques were originally proposed and developed in the context of systems used for maintaining the revision history of software projects and other documents (Berliner 1990; Rochkind 1975; Tichy 1985). In such systems, there are multiple, often almost identical, versions of each document, including different branches, that have to be stored, to enable users to retrieve and roll back to past versions. See Hunt et al. (1998) for more discussion on delta compression in the context of such systems.
- **Software Patch Distribution:** Delta compression techniques are used to generate software patches that can be efficiently transmitted over a network in order to update installed software packages. A good delta compressor can significantly reduce the cost of rolling out updates, say, for security-critical updates to popular software that need to be quickly disseminated. The case of updating automobile software over wireless links was recently discussed in Nakanishi et al. (2013), while Samteladze and Christensen (2012) addresses efficient app updates on mobile devices. Tools such as *bspatch* and *bsdiff* are especially optimized for the case of updating executables, and even better algorithms for this case are proposed in Percival (2006) and Motta et al. (2007).
- **Improving Data Downloading:** There is a long history of proposals to employ delta compression to improve web access, by exploiting the similarity between the current and an older cached version of a page or between different pages on the same site. A scheme called *optimistic delta* was proposed in Banga et al. (1997) and Mogul et al. (1997), where a caching proxy hides server latency by first returning a potentially outdated cached version, followed if needed by a small patch once the server replies. In another approach, a client with a cached old version sends a tag identifying the version as part of the HTTP request and then receives a patch (Housel and Lindquist 1996; Delco and Ionescu, xProxy: a transparent caching and delta transfer system for web objects. Unpublished manuscript, 2000). A specialized tool called *jdelta* for XML data in push-based services is proposed in Wang et al. (2008)

  It is well known that pages from the same web site are often very similar, mostly due to common layout and menu structure and that this could be exploited with delta compression. Work in Chan and Woo (1999) and Savant and Suel (2003) studies how to identify cached pages that make good reference files for delta compression, while Douglis et al. (1997) proposes a similar idea for shared dynamic pages, e.g., different stock quotes from a financial site.

  While delta compression has the potential to significantly reduce latency and bandwidth usage, particularly over slower mobile links, such schemes have only seen limited

adoption. Examples of widely used implementations are the *Shared Dictionary Compression over HTTP* mechanism implemented by Google in the Chrome browser and in their Brotli compressor (Alakuijala and Szabadka 2016) (which is based on the vcdiff differencing format and employs a mechanism similar to delta compression) and the use of delta compression for reducing network traffic in Dropbox (Drago et al. 2013).

- **Delta Compression in File and Storage Systems:** Delta compression is also useful for versioning file systems that keep old versions of files. The *Xdelta File System* (MacDonald 2000) aimed to provide efficient support for delta compression at the file system level using *xdelta*. Delta compression is also used for redundancy elimination in backup systems. In particular, Kulkarni et al. (2014), Shilane et al. (2012), and Xia et al. (2014b,a) showed that adding delta compression between similar files or chunks of data on top of duplicate elimination (removal of identical files or chunks) can give significant additional size reductions, though at some processing cost. Delta compression was implemented, e.g., in the EMC Data Domain line of products.
- **Efficient Storage of File Collections:** As discussed in the previous section, delta compression has been considered as a way to improve compression of general collections of files that share some degree of similarity; see, e.g., Ouyang et al. (2002). However, subsequent work (Trendafilov et al. 2004; Chang et al. 2006; Ferragina and Manzini 2010) indicates that this approach is often outperformed by optimized compression tools for archiving large data sets, after suitable reordering of the files.
- **Exploring File Differences:** Delta compression tools can be used to visualize differences between documents. For example, the *diff* utility displays the differences between two files as a set of edit commands, while the *HtmlDiff* and *topblend* tools in Chen et al. (2000) visualize the difference between HTML documents. Here, the focus is less on compressed size and more on readability.

Future Directions for Research

Delta compression is a fairly mature technology, and future gains in compression may thus be modest for general scenarios. However, there are still a number of challenges that remain, including the following:

- **Speed Improvements:** Recent years have seen significant improvements in speed for many compression techniques, in some cases due to use of available data parallel (SIMD) instruction sets in modern processors. Examples of recent high-speed methods for delta compression are *ddelta* (Xia et al. 2014b) and *edelta* (Xia et al. 2015), and future work should further improve on these results.
- **Alternative Approaches:** Almost all current delta compressors use LZ-based approaches. While this is a natural approach, there might be alternative approaches (e.g., methods based on Burrows-Wheeler compression Burrows and Wheeler 1994) that could lead to improvements. On a more speculative level, researchers have recently started using recurrent neural networks for better compression, and such an approach might also lead to better delta compression.
- **Formal Analysis of Methods:** Current delta compression algorithms are very heuristic in nature, and there is limited work on formally analyzing their performance or optimality with respect to information theoretic measures. An exception is the work in Xiao et al. (2005), which shows some bounds under a very simple model of document generation, but there is a need for more formal analysis.
- **Integration in Storage Systems:** As discussed, delta compression techniques have been deployed inside storage systems in conjunction with other redundancy elimination techniques. In such systems, it can be challenging to identity suitable reference files for a target file that needs to be stored and to decide when it is worth to apply delta compression techniques, which require fetching the reference files for coding

and decoding, in conjunction with other redundancy elimination techniques. Future research could look for new methods that use fingerprints and associated indexes to quickly find good reference files or explore better tradeoffs between compression and the amount of knowledge that is needed about the reference data (where delta compression is the case of full knowledge).

Cross-References

▸ Redundancy Elimination in Networks and Storage Systems

References

Adler M, Mitzenmacher M (2001) Towards compressing web graphs. In: IEEE data compression conference

Agarwal R, Amalapuraru S, Jain S (2004) An approximation to the greedy algorithm for differential compression of very large files. In: IEEE data compression conference

Ajtai M, Burns R, Fagin R, Long D, Stockmeyer L (2002) Compactly encoding unstructured inputs with differential compression. J ACM 49(3):318–367

Alakuijala J, Szabadka Z (2016) Rfc7932: Brotli compressed data format. Available at https://tools.ietf.org/html/rfc7932

Bagchi A, Bhargava A, Suel T (2006) Approximate maximum weighted branchings. Inf Process Lett 99(2): 54–58

Banga G, Douglis F, Rabinovich M (1997) Optimistic deltas for WWW latency reduction. In: USENIX annual technical conference

Bentley J, McIlroy D (1999) Data compression using long common strings. In: IEEE data compression conference

Berliner B (1990) CVS II: Parallelizing software development. In: Winter 1990 USENIX conference

Burrows M, Wheeler D (1994) A block-sorting lossless data compression algorithm. Technical report. 124, SRC. Digital Systems Research Center, Palo Alto

Chan M, Woo T (1999) Cache-based compaction: a new technique for optimizing web transfer. In: INFOCOM conference

Chang F, Dean J, Ghemawat S, Hsieh W, Wallach D, Burrows M, Chandra T, Fikes A, Gruber R (2006) Bigtable: a distributed storage system for structured data. In: Seventh symposium on operating system design and implementation

Chen Y, Douglis F, Huang H, Vo K (2000) Topblend: an efficient implementation of HtmlDiff in Java. In: WebNet 2000 conference

Douglis F, Haro A, Rabinovich M (1997) HPP: HTML macro-preprocessing to support dynamic document caching. In: USENIX symposium on internet technologies and systems

Drago I, Bocchi E, Mellia M, Slatman H, Pras A (2013) Benchmarking personal cloud storage. In: Internet measurement conference

Ferragina P, Manzini G (2010) On compressing the textual web. In: ACM international conference on web search and data mining

Gailly J (2017) zlib compression library, version 1.2.11. Available at https://zlib.net

Housel B, Lindquist D (1996) WebExpress: a system for optimizing web browsing in a wireless environment. In: ACM conference on mobile computing and networking, pp 108–116

Hunt J, Vo KP, Tichy W (1998) Delta algorithms: an empirical analysis. ACM Trans Softw Eng Methodol 7:192–213

Korn D, Vo KP (2002) Engineering a differencing and compression data format. In: USENIX annual technical conference, pp 219–228

Kulkarni P, Douglis F, LaVoie J, Tracey JM (2014) Redundancy elimination within large collections of files. In: USENIX annual technical conference

MacDonald J (2000) File system support for delta compression. MS thesis, University of California, Berkeley

Mogul JC, Douglis F, Feldmann A, Krishnamurthy B (1997) Potential benefits of delta-encoding and data compression for HTTP. In: ACM SIGCOMM conference, pp 181–196

Molfetas A, Wirth A, Zobel J (2014a) Scalability in recursively stored delta compressed collections of files. In: Second Australasian web conference

Molfetas A, Wirth A, Zobel J (2014b) Using inter-file similarity to improve intra-file compression. In: IEEE international congress on big data

Motta G, Gustafson J, Chen S (2007) Differential compression of executable code. In: IEEE data compression conference

Nakanishi T, Shih H, Hisazumi K, Fukuda A (2013) A software update scheme by airwaves for automotve equipment. In: International conference on information, electronics, and vision

Ouyang Z, Memon N, Suel T, Trendafilov D (2002) Cluster-based delta compression of a collection of files. In: Third international conference on web information systems engineering

Percival C (2006) Matching with mismatches and assorted applications. PhD thesis, University of Oxford

Rochkind M (1975) The source code control system. IEEE Trans Softw Eng 1:364–370

Samteladze N, Christensen K (2012) Delta: delta encoding for less traffic for apps. In: IEEE conference on local computer networks

Savant A, Suel T (2003) Server-friendly delta compression for efficient web access. In: 8th international workshop on web content caching and distribution

Shilane P, Huang M, Wallace G, Hsu W (2012) WAN optimized replication of backup datasets using stream-informed delta compression. In: USENIX symposium on file and storage technologies

Tate S (1997) Band ordering in lossless compression of multispectral images. IEEE Trans Comput 46(45): 211–320

Tichy W (1984) The string-to-string correction problem with block moves. ACM Trans Comput Syst 2(4): 309–321

Tichy W (1985) RCS: a system for version control. Softw Pract Exp 15:637–654

Trendafilov D, Memon N, Suel T (2002) zdelta: a simple delta compression tool. Technical report. Polytechnic University, CIS Department

Trendafilov D, NMemon, Suel T (2004) Compressing file collections with a TSP-based approach. Technical report TR-CIS-2004-02. Polytechnic University

Tridgell A (2000) Efficient algorithms for sorting and synchronization. PhD thesis, Australian National University

Wagner RA, Fisher MJ (1974) The string-to-string correction problem. J ACM 21(1):168–173

Wang J, Guo Y, Huang B, Ma J, Mo Y (2008) Delta compression for information push services. In: International conference on advanced information networking and applications – workshops

Xia W, Jiang H, Feng D, Tian L (2014a) Combining deduplication and delta compression to achieve low-overhead data reduction on backup datasets. In: IEEE data compression conference

Xia W, Jiang H, Feng D, Tian L, Fu M, Zhou Y (2014b) Ddelta: a deduplication-inspired fast delta compression approach. Perform Eval 79:258–272

Xia W, Li C, Jiang H, Feng D, Hua Y, Qin L, Zhang Y (2015) Edelta: a word-enlarging based fast delta compression approach. In: USENIX workshop on hot topics in storage and file systems

Xiao C, Bing B, Chang GK (2005) Delta compression for fast wireless internet downloads. In: IEEE GlobeCom

Ziv J, Lempel A (1977) A universal algorithm for data compression. IEEE Trans Inf Theory 23(3):337–343

Ziv J, Lempel A (1978) Compression of individual sequences via variable-rate coding. IEEE Trans Inf Theory 24(5):530–536