Accordion: Multi-Scale Recipes for Adaptive Detection of Duplication

Russell Lewis and John H. Hartman
Department of Computer Science, University of Arizona
{russelll,jhh}@email.arizona.edu

Abstract

A *recipe* is metadata that describes the contents of a file as a sequence of blocks identified by their hash. Using recipes, one can rapidly compare the contents of two files without reading the files themselves. Unfortunately, recipes present a space/precision tradeoff: small block sizes will maximize the duplication that is discoverable, but large block sizes produce small recipes that can be compared more quickly. In this paper, we present Accordion, a toolset for the creation and use of multi-scale recipes – that is, recipes that include blocks at several different scales. We demonstrate two duplication-detection algorithms – one optimized for situations where lots of duplication is expected, and another for those where the existence of duplication is uncertain.

1 Introduction

Storage systems often contain lots of duplication; backup systems, archives, and Internet mirrors contain lots of duplicate data, either in space (different files with the same data) or time (different versions of the same file often contain duplication) [1] [2] [3] [4]. One de-duplication strategy is to use *recipes* to describe files, allowing for rapid detection of duplication [5] [6]. A recipe is metadata that describes the contents of a file as a sequence of blocks identified by their hash. However, recipe-based systems face a block-size tradeoff; small block sizes detect more duplication, but large block sizes produce shorter recipes, which are less costly to store, transmit, and compare [7]. Most systems choose an appropriate block size empirically, by testing on a representative workload. However, no single block size is optimal for all files.

To solve this problem, we developed a suite of tools called Accordion that create and use multi-scale recipes. A multi-scale recipe is the union of many different single-scale recipes; it can be used to detect large-block duplication rapidly and yet can also provide small-block detection if required.

Accordion implements two different file comparison algorithms. The first is designed to minimize seeks within the recipe. When large amounts of duplication are available, this algorithm discovers that duplication extremely rapidly (up to 80x faster than single-scale recipes in our experiments); however, its worst case is roughly 2x worse than single-scale. The second algorithm has better worst-case performance (matching single-scale in almost every case), but makes many more small seeks through the recipe.

2 Recipe Format and Generator

Some recipe systems use fixed block sizes [3] [7]; however, most are content-aware, and use blocks of variable lengths whose boundaries are determined by the file contents [2] [5]. This allows the systems to detect duplication even when data has been shifted by insertion or deletion.

Accordion is also content-aware, but uses fixed-size, non-aligned (and thus overlapping) blocks. This design was the natural consequence of a more fundamental design choice: Accordion uses Winnowing [8] to select block boundaries, rather than the more conventional Rabin fingerprinting algorithm [9].

Rabin fingerprinting, which is used by most content-aware systems, scans the file with a Rabin polynomial hash; block boundaries are established when the hash value is congruent to some magic value, modulo some divisor. While simple, this algorithm does not provide lower or upper bounds on block sizes. Thus, such systems must override the block boundaries in some scenarios – joining blocks that are too small and breaking up those that are too large. This solution, while workable, is undesirable, as it introduces an algorithm step (block join decisions) that is not content-aware, and thus increases the chances of false negatives when comparing files.

Winnowing uses a different approach. Like Rabin fingerprinting, Winnowing generates a stream of hashes from the file; however, instead of choosing block boundaries based on individual hashes, it runs a sliding window along the stream of hashes. At each window position, the minimum hash is chosen as the fingerprint (that is, a (hash,offset,size) tuple); usually, the same hash is chosen over and over as the window slides. On average, Winnowing chooses a fingerprint for every *w/2* bytes of the file, where *w* is the size of the sliding window. More importantly for our purposes, no two fingerprints can be more than *w* bytes from each other. (See the Winnowing paper [8] for proofs of both of these assertions.)

2.1 Modifying the Hash Strategy

The original Winnowing algorithm has two passes. In the first pass, a Rabin polynomial is used to generate a hash; in the second pass, fingerprints are selected. The selected fingerprints report the first-pass hashes as the hashes for the blocks.

Rabin hashes are used – even though they are weak hashes – because the hash must run at every position in the file; thus, a strong hash (such as SHA1) is impractical. The Winnowing design can lead to false positives, but the authors found this acceptable because all duplicates would be subsequently examined by a human. This compromise is unacceptable for recipes, where false positives can lead to data corruption.

Thus, Accordion adds a third pass to Winnowing. In the first pass Accordion uses a Rabin polynomial hash; it then selects fingerprint *locations* using a sliding window on that stream. Accordion then executes a SHA1 hash at the chosen locations, and the SHA1 values are reported as the hashes in the recipe.

In addition, Accordion uses a single first-pass hash, which is shared by all scales, and the size of the hash window is relatively small – no larger than the smallest requested recipe block size. This means that the Rabin hash can be efficiently generated; it also means that – as detailed in the next section – all of the scales slide their fingerprint-selection windows over the same stream of first-pass hashes.

2.2 Sliding Window Issues

Running Winnowing at very large scales can be expensive, as the sliding window must find the minimum value in a very large set of hashes. The Winnowing paper suggests that one keep track of the running "best" fingerprint in a window – but this requires a periodic brute-force scan of the entire window when a very-good fingerprint finally leaves the window. This is unworkable when window sizes are very large – not only would it be computationally expensive, it also would require that Accordion store a very long stream of hashes in memory.

Thus, Accordion culls the list of hashes as it progresses through the file; it only keeps the "interesting" hashes. A hash is interesting if it is the minimum hash in a range stretching from its position to the end of the sliding window; if the hash is the minimum in that range, then it is possible that it might be selected as a fingerprint in the future. This leads to an invariant: every hash in the list is less than or equal to the hash that follows it.

As the window slides, Accordion updates the list of interesting hashes. This involves three steps. First, at every new position, Accordion adds a new hash at the end of the list – since the most recent hash is always the minimum in a range of length one (even if it might become uninteresting at the very next window position).

Second, this new hash may violate the invariant of the list, as it may have a lower value than some or all of the elements already on the list. Thus, the new hash is compared to the element immediately before it (that is, the last hash from the old list); if the old element has a larger hash than the new, then the old is removed. This process repeats until the invariant is re-established.

Finally, Accordion examines the head of the list; if it is now so old that it is outside of the sliding window, it is removed.

Thus, with amortized constant cost per byte in the file, Accordion maintains a list of the interesting hashes – and, by definition, the oldest hash in the list is always the fingerprint for the current window position.

This algorithm is easily generalized to multi-scale operation: Accordion simply chooses, at every position of the sliding window, a fingerprint for each scale, selecting the oldest interesting hash within that window size.

3 Algorithms and Experimental Protocols

When Accordion compares two recipes, it must skip over some of the recipe entries – because a multi-scale recipe is the union of many different single-scale recipes, and the total size is approximately twice the size of a single-scale recipe using the smallest block size. (The multi-scale recipe is twice the size because Accordion builds recipes for every power of 2).

Thus, each time that Accordion has a hash hit – that is, whenever a duplicate block is found in a file – it records the range of the file covered by that block. When Accordion later encounters recipe entries that cover the same range (or some subset of it), it skips over those (now redundant) recipe entries.

Therefore, in this paper we will rate the algorithms by

	offset	size	hash
1.	0x0000	0x2000	... (miss)
2.	0x0000	0x1000	... (miss)
3.	0x0789	0x1000	... (hit)
4.	0x1234	0x1000	... (miss)
5.	0x1c02	0x1000	... (miss)
6.	0x0000	0x0800	... (hit)
7.	0x0789	0x0800	... (hit)
8.	0x0e16	0x0800	... (hit)
9.	0x1234	0x0800	... (miss)
10.	0x1c02	0x0800	... (miss)

Table 1: An Example Recipe (for top-down)

	offset	size	hash
1.	0x0000	0x0800	... (hit)
2.	0x0000	0x1000	... (miss)
3.	0x0000	0x2000	... (miss)
4.	0x0789	0x0800	... (hit)
5.	0x0789	0x1000	... (hit)
6.	0x0e16	0x0800	... (hit)
7.	0x1234	0x0800	... (miss)
8.	0x1234	0x1000	... (miss)
9.	0x1c02	0x0800	... (miss)
10.	0x1c02	0x1000	... (miss)

Table 2: An Example Recipe (for bottom-up)

the number of recipe entries read. Our goal is to ensure that Accordion's algorithms perform no worse than single-scale comparison of the same files; of course, sometimes, Accordion will perform far better.

3.1 Algorithm 1: top-down

Accordion's top-down file comparison algorithm assumes that the recipe entries are already sorted, first by block size (descending) and then by file offset (ascending). The algorithm reads the recipe in order, only skipping over recipe entries that have been previously covered at larger scales.

The intuition behind this algorithm is simple: it attempts to detect duplication at the largest scale, and then recurses down – but only into parts of the file that have not already been covered by hash hits. When huge duplication exists (as with duplicate files, archives, etc.), it is found very quickly.

For example, consider the recipe snippet in Table 1 (each line has been annotated as to whether the hash lookup is a hit or miss). In this example, Accordion reads lines 1 through 6; it finds two hits and four misses. None of the lines are skipped. However, lines 7 and 8 are both skipped – because each is entirely contained inside a previous hit (line 3). Lines 9 and 10 are *not* skipped – because they are not contained within any previous hit.

The downside of this algorithm is its poor worst case: if little duplication is found – or if it is not contiguous – then this algorithm reads the entire recipe, which is approximately twice the size of a single-scale recipe. The break-even point for this algorithm thus is a file that contains 50% contiguous duplication – in that case, the total cost is half the multi-scale recipe size, which is approximately the same as a single-scale recipe.

3.2 Algorithm 2: bottom-up

Accordion's bottom-up file comparison algorithm assumes that the recipe entries are sorted first by file offset (ascending) and second by block size (ascending). Like top-down, this algorithm skips over already-covered ranges; however, when it encounters a hash miss at a given file offset (at any scale), it skips over all of the remaining recipe entries at that file offset.

The intuition behind this algorithm is that if a small-scale block is not a duplicate then a large-scale block that contains it cannot be a duplicate either. If large, contiguous duplication is available, then this algorithm will find it quickly, as it rapidly works its way “up the tree” to the large blocks. But if no duplication is available, then it reads no more recipe entries than a single-scale recipe.

For example, consider the same recipe as before, but sorted for this algorithm, as listed in Table 2. This algorithm reads line 1, which is a hit; it thus continues to line 2. Line 2, however, is a miss – and thus, line 3 is skipped.

The downside of this algorithm is that it performs small seeks through the recipe. The algorithm will routinely skip over a handful of recipe entries at a time. Thus, this algorithm trades off continuity in recipe entries read in favor of reducing the worst-case number of recipe entries read.

3.3 Experimental Setup

We compared Accordion's two algorithms against a simple single-scale recipe, at a variety of scales. We used a variety of input files from prior art that are known to exhibit duplication, such as multiple versions of the same source code and related Linux install ISOs. We also tested pairs of files which we expected to have

	single-scale		multi-scale	
Block Size	Dup Found (MB)	Recipe Size (entries)	Dup Found (MB)	Recipe Size (entries)
1 KB	22.4	46889	22.6	99978
16 KB	22.4	3813	22.4	8292
256 KB	21.3	290	21.5	749
4 MB	4.6	36	4.6	44

Table 3: multi-scale vs. single-scale
Linux 2.0.9 to 2.0.10

no duplication, and manually constructed torture tests, to exercise the worst case of the algorithms.

4 Results

Our first experiment compared two versions of the Linux kernel source code; we use the files used by Tridgell in his rsync dissertation [5]. These were specifically chosen by Tridgell to demonstrate a pair of files with massive (but not complete) duplication. (See that paper for a discussion of how the files were generated.)

Table 3 summarizes this experiment. As expected, the multi-scale recipe was roughly twice the size of the single-scale recipe for the same minimum block size; also, the ranges covered by multi-scale were strict supersets of those covered by single-scale. These general properties held for all pairs of files that we tested.

Figure 1 shows the cost of our file comparison algorithms on the same pair of files; recall that the cost is defined as the number of recipe entries actually used by each algorithm. Since the number of entries used by different scales differ by orders of magnitude, these curves are normalized to the single-scale cost for each scale.

We see that since there is a large amount of duplication available (over 99% of the entire file), both of our algorithms use very few entries. In fact, at the smallest scale (1KB), top-down skips over almost the entire multi-scale recipe (automatically using huge blocks almost entirely), and thus uses only 622 entries. By contrast, single-scale uses 46889 entries at the 1KB scale – a difference of almost 80x.

Figure 2 shows the cost when there is less duplication available. In this test, we compared two Fedora 17 install ISOs. The files contained approximately 10% duplicate data (66 MB of duplicate data, scattered through files of size 646 MB and 695 MB), causing top-down to perform poorly. However, bottom-up worked remarkably well, beating single-scale at all scales.

This pattern continued over many different pairs of files: if the files had a high percentage of duplication,

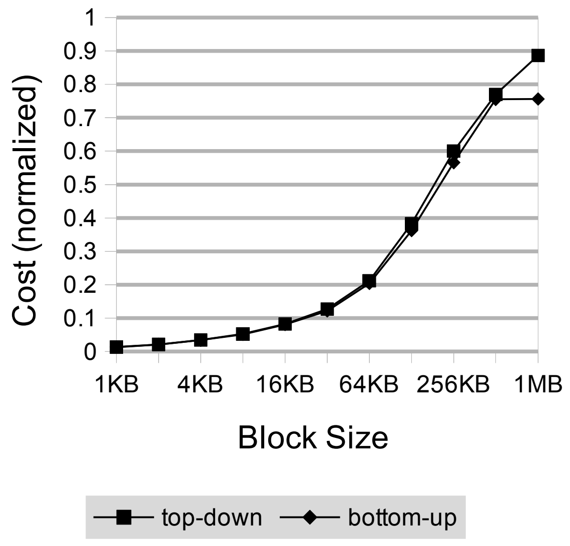

Figure 1: Algorithm Cost
linux-2.0.9 vs. linux-2.0.10

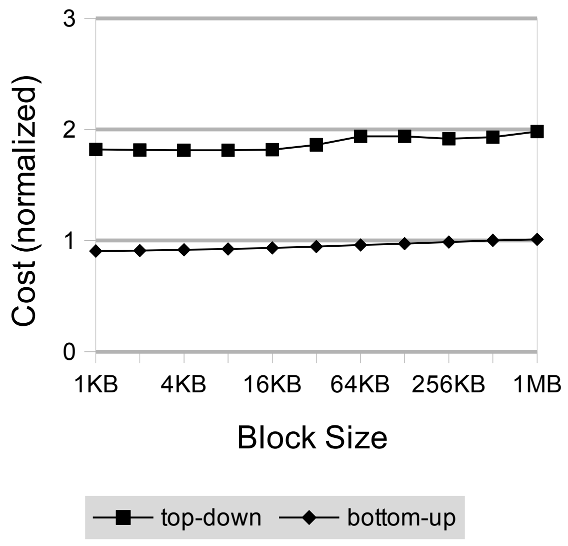

Figure 2: Algorithm Cost
Fedora-17-i686-Live-KDE.iso vs.
Fedora-17-i686-Live-Desktop.iso

then either algorithm worked well; if there was little duplication, then bottom-up would track well with single-scale, but top-down would perform poorly.

Finally, we constructed a torture test to expose the worst case of the bottom-up algorithm. We copied an existing file, and then zeroed the last byte in every 2K block. These two files had huge duplication, but almost no duplicated ranges 2K or longer; thus, bottom-up needed to read almost every 1K recipe entry, and also many of the 2K recipe entries. At 1K (the only scale where significant duplication was detected), bottom-up read about 30% more entries than single-scale.

We initially expected to see about 50% additional cost, but in hindsight 30% makes more sense: in a 2K range of the file, Winnowing selects roughly 4 fingerprints at the 1K block size and 2 at the 2K block size. On average, bottom-up reads every 1K recipe entry, and about half of the 2K entries (due to hash hits at the 1K size); thus, in a 2K window, single-scale reads about 4 recipe entries, and bottom-up reads around 5.

5 Related Work

A variety of systems have used hierarchical approaches to detecting duplication, including LBFS [2], Venti [3], and TAPER [10]; most are analogous to Merkle trees [11], where small-block hashes are grouped and then hashed again to generate large-block hashes. However, these groupings are not content-aware; thus, when data is inserted or deleted, duplication is likely to be overlooked at the large scale (even if it will be found, eventually, at smaller scales).

Multiround rsync [12] is a generalization of the rsync protocol that runs in several passes, with notable similarities to our top-down algorithm. However, that paper does not present any way to solve the worst-case cost (which is worse that rsync). Also, as it is focused on live comparisons over a network, it generates (offset, hash) pairs only as needed for a single comparison. Recipes, on the other hand, can be stored and then used for later comparisons, and contain all of the possible (offset, hash) pairs – a major advantage if the recipe is to be generated once, and then used later, such as in an all-to-all file comparison.

6 Conclusions

In this work, we presented Accordion, a toolset for creating and using multi-scale recipes. We demonstrated two algorithms for finding duplication using multi-scale recipes, each optimized for a different scenario. Top-down worked best when large amounts of duplication were available, and was up to 80x as efficient as single-scale recipes. Bottom-up was optimized for the worst-case (performing no worse than single-scale except in artificial torture tests), and was competitive with top-down in the best case.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Award Numbers DBI-0735191 and DBI-1265383. URL: www.iplantcollaborative.org

Preliminary work on multi-scale recipes was done in connection with CSc 630 (Advanced Topics in Software Systems) at the University of Arizona. Thanks to project partner Gavin Simmons for contributions to the early prototype and to Professors Larry Peterson and Todd Proebsting for inspiration and direction. Thanks to team member Illyoung Choi for feedback as we investigated these algorithms.

References

[1] Manber, Udi. "Finding Similar Files in a Large File System." Usenix Winter. Vol. 94. 1994.

[2] Muthitacharoen, Athicha, Benjie Chen, and David Mazieres. "A low-bandwidth network file system." ACM SIGOPS Operating Systems Review. Vol. 35. No. 5. ACM, 2001.

[3] Quinlan, Sean, and Sean Dorward. "Venti: A New Approach to Archival Storage." FAST. Vol. 2. 2002.

[4] Park, KyoungSoo, et al. "Supporting Practical Content-Addressable Caching with CZIP Compression." USENIX Annual Technical Conference. 2007.

[5] Tridgell, Andrew. Efficient algorithms for sorting and synchronization. (PhD thesis.) Canberra: Australian National University, 1999. http://gan.anu.edu.au/~brent/pd/Tridgell-thesis.pdf

[6] Tolia, Niraj, et al. "Opportunistic Use of Content Addressable Storage for Distributed File Systems." USENIX Annual Technical Conference, General Track. Vol. 3. 2003.

[7] Nath, Partho, et al. "Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines." management 7.5 (2006): 20.

[8] Schleimer, Saul, Daniel S. Wilkerson, and Alex Aiken. "Winnowing: local algorithms for document fingerprinting." Proceedings of the 2003 ACM SIGMOD international conference on Management of data. ACM, 2003.

[9] Rabin, Michael O. Fingerprinting by random polynomials. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[10] Jain, Navendu, Michael Dahlin, and Renu Tewari. "TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization." FAST. Vol. 5. 2005.

[11] Merkle, Ralph C. "A digital signature based on a conventional encryption function." Advances in Cryptology—CRYPTO'87. Springer Berlin Heidelberg, 1988.

[12] Langford, John. "Multiround rsync." (2001). Unpublished manuscript. http://www-2.cs.cmu.edu/~jcl/research/mrsync/mrsync.ps