File Recipe Compression in Data Deduplication Systems

Dirk Meister, André Brinkmann, Tim Süß
Johannes Gutenberg University Mainz

Abstract

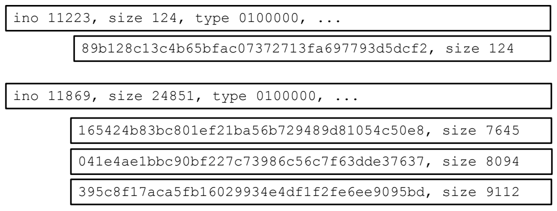
Data deduplication systems discover and exploit redundancies between different data blocks. The most common approach divides data into chunks and identifies redundancies via fingerprints. The file content can be rebuilt by combining the chunk fingerprints which are stored sequentially in a file recipe. The corresponding file recipe data can occupy a significant fraction of the total disk space, especially if the deduplication ratio is very high. We propose a combination of efficient and scalable compression schemes to shrink the file recipes' size. A trace-based simulation shows that these methods can compress file recipes by up to 93%.

1 Introduction

The amount of data, which needs to be stored, increases permanently. Deduplication has proven to be a viable approach to reduce the resulting capacity demands, especially for backup workloads. Typically, deduplication systems divide files or blocks into chunks and identify redundant chunks by comparing their cryptographic fingerprints. Different indexes are used to manage the relations between files and chunks, which require additional capacities beside the deduplicated data.

The *chunk index* contains the fingerprints of the stored chunks [11, 12, 17, 25]. Simply storing the chunk index on disk would lead to a *disk bottleneck*, as every write operation triggers multiple index lookups. Previous research focused on the chunk index and has been able to overcome the disk bottleneck [12, 31].

However, every deduplication system has an additional persistent index to store the information that is necessary to rebuild file content based on *file recipes* [4, 6, 13, 23, 25, 27–29], a term introduced by Tolia et al. [26]. A file recipe contains a list of chunk identifiers. Each of these chunk identifiers is the cryptographic fingerprint for one specific chunk. These fingerprints point to data chunks on storage, similar to block pointers in standard file systems point to disk blocks. Using the file recipe, the original file contents (denoted as logical data) can be reconstructed by using the uniquely identifiable chunk data. To reconstruct the logical data the fingerprints are read sequentially and their associated data chunks are loaded and concatenated. Figure 1 shows two exemplary file recipes.



```
ino 11223, size 124, type 0100000, ...
        89b128c13c4b65bfac07372713fa697793d5dcf2, size 124

ino 11869, size 24851, type 0100000, ...
        165424b83bc801ef21ba56b729489d81054c50e8, size 7645
        041e4ae1bbc90bf227c73986c56c7f63dde37637, size 8094
        395c8f17aca5fb16029934e4df1f2fe6ee9095bd, size 9112
```

Figure 1: Illustration of file recipes.

One difference to the file system block pointers used in standard file systems is the size of pointer data type. Usually the pointer in a standard file system is stored in at most 64 bit. All data deduplication systems, where it is known how the mapping from a file to the deduplicated contents is stored, use file recipes that store cryptographic fingerprints. In these systems the used fingerprints have a size of at least 20 byte [4, 13, 25, 27–29].

The research on data deduplication largely ignored the management of file recipes. The reason is that it is unlikely that file recipes are the performance bottleneck for the system throughput. However, the file recipes may occupy a significant portion of the overall storage in a backup deduplication system.

A file recipe stores 20 byte for each referenced chunk if SHA-1 is used. Therefore, the file recipes grow linearly with the size of the stored logical data. With an average chunk size of 8 KB, a TB of logical data requires 2.5 GB memory to store the corresponding file recipes. Assuming weekly full backups with a retention period of 52 weeks and a backup size of 20 TB, the file recipes have a size of at least 2.4 TB.

The physical capacity used to store the data on the other hand is only slowly growing over time. For example, Zhu et al. report a compression factor of 40.63 for daily backups by applying deduplication [31]. Assuming an internal data deduplication ratio within a backup run of $ir = 0.35$, a temporal deduplication ratio between weekly full backups of $tr = 0.95$, and a data compression ratio of 0.5, the post-deduplication data for one year has an estimated size of 33.5 TB for a weekly backup size of 20 TB.

However, daily full backups and/or longer retention periods become more common. If daily backups with a retention period of a year are used, assuming a slightly higher temporal deduplication between the daily backups of $tr = 0.99$, the post-deduplicated data has an estimated size of 43 TB for a 20 TB backup. The file recipes' size grows from 2.4 TB for weekly full backups to an estimated size of 17.1 TB. Therefore, they already occupy about 40% of the disk space that the deduplicated data requires to be stored.

An approach to reduce the size of the file recipes has an important impact on the overall storage usage of a data deduplication system, especially when considering daily full backups in an environment with a high deduplication ratio.

One way to reduce the file recipes' size is to use larger chunk sizes, e.g., 16 KB instead of 8 KB. This reduces the number of fingerprints stored in file recipes and therefore also the overall size of the file recipes. However, previous studies have shown that increased chunk sizes decrease the deduplication ratio [16, 18, 28]. We therefore explore alternatives in this work.

We are comparing four compression approaches, using unique properties of data deduplication systems. These approaches reduce the size of the file recipes by up to 93% without having a significant impact on the system throughput.

The first, already established approach suppresses the fingerprint of the chunk filled with zeros (the so-called zero chunk). This idea has been proposed by Wei et al. [29] and is described for completeness and comparison.

The other three approaches are novel contributions: The second approach assigns to each chunk fingerprint a shorter code word based on the number of stored fingerprints. It uses the chunk index to derive the code words, which form an alternative key for the chunk index entry. Thus, the code words can be used instead of the full fingerprints. The third approach uses the observation that the usage of fingerprints is highly skewed. Certain fingerprints are more likely than others. A small set of common fingerprints gets a short code word assigned using an explicit dictionary. Finally, the fourth approach uses the observation that it is possible to predict the next fingerprint in a backup data stream if the previous fingerprint is known. If the next fingerprint can be correctly predicted using the previous fingerprint as context, a special code word is stored as a flag instead of the original fingerprint.

The compression of file recipes is not the most pressing concern when designing a data deduplication system and developing an approach to achieve a good deduplication ratio with a high performance has been the focus of previous research. Nevertheless, file recipe compression allows to gain significant additional capacity savings.

All presented approaches are practical, efficient, and scalable for the usage in data deduplication systems. While they are not limited to a concrete deduplication system, the usability in practice may depend on the specifc system and its design decisions, e.g., the organization of the chunk index.

2 File Recipe Compression

The idea of file recipe compression is to assign (small) code words to fingerprints. The code word is then stored instead of the fingerprint in the file recipe. Different approaches to select the code words are explored in this paper.

We use three different backup data sets in this section to motivate the approaches based on observations from the data sets. The same data sets will later be used for the evaluation in the next section.

All traces use Content-defined Chunking with an expected chunk size of 8 KB [14]. The first data set, called *HOME1*, is based on the home directory file server of the University of Paderborn [16]. It is a time series containing 15 weekly full backup traces of the same file system of 440 GB (6.6 TB total). With data deduplication, the data set size can be reduced by a factor of 1:20 to 369 GB. The second data set, called *HOME2*, contains traced data from the home directory file server at Johannes Gutenberg University Mainz. This data set consists of 5 weekly backups forming a total data set of 4.8 TB. Data deduplication compresses the backup data set to 779.9 GB (factor 1:6). The last data set, called *ENG*, contains fingerprints for files from a file server of an engineering department backed up over 21 weeks (319 GB total). Data deduplication is able to shrink the physical size of this data set by a factor of 1:40 to 8.0 GB.

We describe the compression approaches in this section, followed by the evaluation in Section 3.

2.1 Zero-Chunk Suppression

Several authors have noted before that a few chunks are responsible for a high number of duplicates [8, 10, 11, 16, 29]. Especially noteworthy is the chunk completely filled with zeros ("zero chunk"). Jin et al. show that zero chunks are common in VM disk images [10].

Several deduplication systems apply a special handling for this chunk [11, 29]. It is easy to detect zero chunks and replace them with a special code word by pre-calculating the fingerprint of the chunk filled with zeros. Zero chunks are never looked up in the chunk index or stored on disk.

However, it can also be used to reduce the size of file recipes as, e.g., Wei et al. propose [29]. If the special code word assigned is small, e.g., 1 byte, the size of the file recipe is reduced. Alternatively, it is possible to store a bit mask in the file recipe similar to the way missing values are handled in some databases [24].

2.2 Chunk Index Page-oriented Approach

The second approach aims to assign a code word to all chunks that is not significantly longer than necessary to have a unique code word for each chunk in the system. The approach is called *chunk index page-oriented* because it uses chunk index's pages to assign code words. The approach assumes that the chunk index is implemented using a paged disk-based hash table and fingerprints are hashed to on-disk pages.

The code word consists of two parts: prefix and suffix.

The suffix is chosen using the least number of bits possible, which are still unique within the page. The code word is aligned to byte boundaries allowing a faster processing. The prefix of the code word denotes the page number. The combination of suffix and prefix together uniquely identify the fingerprint (see Figure 2).

The code word is an alternative key for the full fingerprint in the chunk index. The prefix is used to identify the index on-disk page where the fingerprint is stored on. Given the page contents, the fingerprint and its chunk index entry data can be found by searching in the page for the fingerprint with the matching suffix. Therefore, it is possible to lookup the chunk index entry corresponding to the fingerprint without an additional index lookup.

Alternatively, the suffix can be used as an index within the page if the chunk index entries are of a fixed size.

This approach assigns code words when the fingerprint is stored in the system for the first time. The complete fingerprint/code pair is stored in the container to allow reading an item from a container using only the code word.

2.3 Statistical Approaches

The zero-chunk suppression assigns short fingerprints to certain well-known fingerprints, while the chunk index

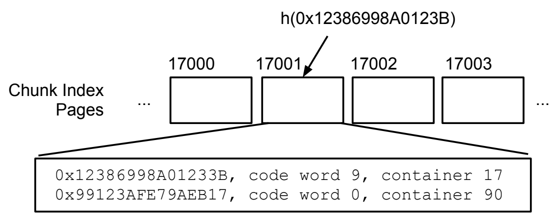

h(0x12386998A0123B)
17000
17001
17002
17003
Chunk Index Pages
...
...
0x12386998A01233B, code word 9, container 17
0x99123AFE79AEB17, code word 0, container 90

Figure 2: Illustration of the page-oriented approach. A fingerprint is hashed to an index page. A page-unique code word prefix is assigned to each fingerprint. The complete code word is 17001||9.

Table 1: Order-0 statistic for the *ENG* data set after 3 backups.

Fingerprint	Usage count	Probability	Entropy
0x518843...	380,508	0.108765	3.2
0x04F90E...	7,227	0.002066	8.9
...	...	...	...
0x435123...	1000	0.001	10.0
...	...	...	...
0x6B4D0A...	3	≈ 0	20.1

page-oriented approach provides a basic code word for all fingerprints in the system.

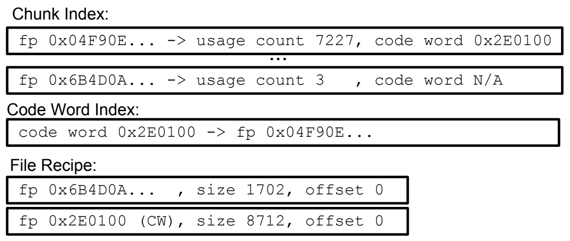
The statistical approaches presented in this section generalize the zero chunk suppression. They use statistics to assign shorter code words to fingerprints based on the probabilities of the chunk usages.

The first approach relies on the observation that the number of references to chunks is highly skewed [16]. The assigned code words are stored in an additional (in-memory) dictionary to allow a lookup from the code word to the fingerprint.

The approach uses the fingerprint's order-0 statistic. This means the statistical model relies on a fingerprint's usage without looking at its context, e.g., at previous fingerprints in the data stream.

Information theory provides a lower bound on the code word length that would be achievable with this statistical model. The information content is the sum of the fingerprint entropies. The entropy of a fingerprint h is $-log_2(\frac{\text{usage count of } h}{\text{total chunk usage}})$. The higher the probability of a fingerprint, the lower is the entropy and the shorter the optimal code word length. This skew can be observed in all three data sets introduced before. Table 1 shows examples of the entropy statistics of the *ENG* data set. The average entropy (and therefore optimal average code word length) is 17.3 bits per chunk using this statistical model and an optimal compression scheme.

The approach is similar to classical text compression using Huffman codes [9]. However, the compression of file recipes has unique properties that render it impossi-



```
Chunk Index:
fp 0x04F90E... -> usage count 7227, code word 0x2E0100
...
fp 0x6B4D0A... -> usage count 3   , code word N/A
Code Word Index:
code word 0x2E0100 -> fp 0x04F90E...
File Recipe:
fp 0x6B4D0A...  , size 1702, offset 0
fp 0x2E0100 (CW), size 8712, offset 0
```

Figure 3: Illustration of the statistical dictionary approach. The chunk with the fingerprint 0x04F90E... has, based on its high usage count, a short code word.

ble to use the classic compression approaches directly:

1. The size of the code alphabet is 2^{160} instead of 2^8. It is therefore not feasible to create full Huffman tree in memory.
2. Data deduplication has to support random accesses. Adaptive text compression decoders rely on parsing the full history of a compressed data stream to build up the same model as seen by the encoder. This is, e.g., the reason that *.tar.gz* files do not provide random access.

Since it is not practically possible to build a full huffman tree, we chose a relaxed way to use the skew of the chunks for file recipe compression: If the entropy of a chunk is below a certain threshold, a fixed size code word is assigned to the fingerprint. The check can either be performed in a background process or alternatively when a write command accesses a chunk.

However, the entropy is not known in advance and can only be estimated after some data has been written. Therefore, we only assign code words after one or more backups. The first fingerprints have to be stored unmodified, as no code word is available.

Once a code word is assigned, it can never be revoked as long as a file recipe uses that code word. If at some point a chunk usage falls below a threshold, we declare the code word as inactive and all new appearances again use the full fingerprint. Older backup runs are eventually evicted for the backup storage system. If there is no longer a reference to the code word stored in any file recipe, the code word is garbage collected and the code word becomes free again, e.g. it is possible to maintain a separate code word usage counter in addition to the chunk usage counter in the chunk index.

For most of the chunks, the entropy is nearly identical to the length of the code word assigned by the page-oriented method. It is, therefore, not worthwhile to assign an entropy-based code word to more than a small fraction of the chunk.

One important assumption is that it is effectively possible to estimate the overall probability of a chunk. The number of references to a chunk (usage count) is often collected for garbage collection purposes. By putting this usage count in relation to the total number of references, the probability of a chunk can be easily calculated.

The statistical dictionary approach uses the probability that a chunk fingerprint is referenced without using context information (order-0 statistic) to build a code word based on that.

The order-1 statistic looks at the previous chunk and calculates the probability of the fingerprint based on that context information.

The entropy of the order-1 model again provides us a theoretical lower bound. In the *ENG* data set, the entropy using the order-1 statistical model is 0.13 bits. The statistics in the other data sets are similar (0.14 bits in *HOME1*, 0.21 bits in *HOME2*).

The low entropy using this statistical model in backup data streams has two main reasons: The redundant data found by the deduplication process is usually clustered together and forms larger sequences of redundant and unique chunks, called “runs”. In the *HOME1* data set’s last backup the average run length of redundant chunks without a single unknown chunk breaking the series is 449.8. The average run length in the *HOME2* data set is shorter (262.9), and it is larger in the *ENG* data set (901.5). In all cases, there is a significant clustering, which forms long runs of redundant chunks. The backup runs change only slightly from one backup run to another and usually the backup is written sequentially. This is one of the assumptions that empower the locality preserving caching approaches used by EMC Data Domain [28].

This shows that there is only little uncertainty about the next fingerprint if the previous fingerprint is known. Capturing this prediction in a practical compression scheme for the file recipes promises significant compression.

However, the information to store the order-1 model grows quadratic in the number of chunks and is therefore not usable in practical deduplication systems.

The approach presented here relaxes the order-1 model and reduces the information kept per chunk to a constant size. Considering the low entropy of the order-1 model in the backup data streams, even a relaxed approach should be able to predict fingerprints with high accuracy.

We use the data stream algorithm *Misra-Gries* to approximate the detection of frequent fingerprint pairs [19]. We therefore store k fingerprints (or code words) in the chunk index entry. The parameter k denoting the number of possible frequent fingerprints is used to trade accuracy against memory overhead. We later compare different choices for k and also compare the relaxed approach with the full order-1 statistics.

When a file recipe is updated, the frequent item data

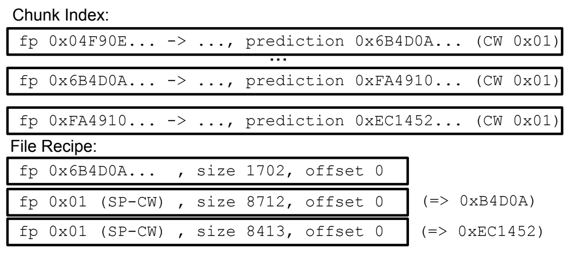

Chunk Index:
fp 0x04F90E... -> ..., prediction 0x6B4D0A... (CW 0x01)
...
fp 0x6B4D0A... -> ..., prediction 0xFA4910... (CW 0x01)
fp 0xFA4910... -> ..., prediction 0xEC1452... (CW 0x01)
File Recipe:
fp 0x6B4D0A... , size 1702, offset 0
fp 0x01 (SP-CW) , size 8712, offset 0
(=> 0xB4D0A)
fp 0x01 (SP-CW) , size 8413, offset 0
(=> 0xEC1452)

Figure 4: Illustration of the statistical prediction approach. If the fingerprint 0x6B4D0A... occurs after 0x04F90E..., the fingerprint is replaced by the code word 0x01.

structure of the previous fingerprint is updated with this fingerprint in the file recipe. Based on this, the data structure can be queried to receive the estimation for the fingerprint that most frequently followed the previous fingerprint. We select this fingerprint as the prediction candidate and assign a short 1 byte code word to it. Similar to the statistical dictionary approach a code word is not changed as long as a file recipe uses it. Instead, if eventually a different prediction candidate is chosen, another free code word is used.

When data is written to the system, the prediction candidate based on the previous chunk is checked. If a fingerprint matches the prediction candidate based on the previous fingerprint, the code word is stored in the file recipe. Figure 4 shows an example for this approach. Similarly to the zero chunk code word, a bit mask in the file recipe would be an alternative for an even tighter encoding.

Every b fingerprints in the file recipe, the full fingerprint (or a different kind of code word) is stored in the file recipe. This provides an anchor so that random accesses are possible. We (experimentally) found $b = 32$ to be a suitable value for a backup workload.

3 Evaluation

We evaluate the approaches using the following criteria:

1. Storage: Additional information that is stored per chunk in the chunk index or other indexes.
2. Memory: Amount of main memory per chunk used for compression. The memory cost does not include temporarily used memory, e.g., for paging on-disk data.
3. Assignment: Time when code word is assigned.
4. Background Processing: Is an additional background processing, e.g., in idle times, necessary.
5. Recipe Compression: Compression ratio of the file recipes, evaluated based on the trace data sets presented before. The size of the original file recipes, assuming 20 byte per chunk reference, is compared to the size after compression.

Table 2: Overview of recipe compression's properties and results. Numeric values in bytes per chunk.

	Storage	Memory	Assignm.	Backgr.
Zero-Chunks (ZC)	None	None	Direct	No
Page-based (PB)	≈ 1	None	Direct	No
Directory (SD)	≈ 0.24	≈ 0.24	Delayed	Yes
Prediction (SP)	$\approx$ 20-68	None	Delayed	No

The compression methods are first discussed individually. Later, we apply the methods together and present the combined compression ratios. Table 2 summarizes the properties 1-4. The compression ratios are shown in Table 3.

The *zero chunk suppression* (ZC) method does not cause any additional IO operations nor does it increases the storage requirements. In contrast, it saves a significant amount of chunk index lookup operations if the backup stream contains a high amount of zero chunks and has therefore a positive performance impact.

The compression results depend on the data set: In the ENG data set, 14.9% of the chunks references the zero chunk. The zero chunk suppression reduces the file recipes by 11.8%. On the other hand, the *HOME1* data set contains only 0.6% zero chunks and the *HOME2* data set contains only 0.3%. Here, the file recipes shrink by 0.5%, resp. 0.26%.

The code word length using the *page-oriented approach* (PB) depends on the scale of the system. In an 8 TB configuration with 2^{24} chunk index pages and a page size of 4 KB, each chunk index page will hold 64 chunks on average. The balls-into-bins theory [22] shows that with high probability no page has to store more than 97 chunks. Therefore, a suffix length of 7 bits is sufficient to enumerate all chunks in a page and to assign a unique suffix to each of them. The prefix would account for 24 bits, so that the final code word has 31 bits or 4 bytes when aligned to full bytes. This 4 byte code word would not be sufficient for a large configuration with, e.g., 256 TB back end storage, where we have to account for 2^{29} chunk index pages. Here, 5 byte would be sufficient.

The extra storage costs consist of the prefix that is stored in each chunk entry. With 4 byte code words, this method reduces the file recipes by 80% (75% with 5 byte). This method's advantage is that the savings are predictable and do not depend on the concrete data set.

When combined with other methods, the *page-oriented approach* provides a base compression for all fingerprints, while certain other approaches use even smaller code words.

Similar to the zero chunk approach, the savings of the *statistical dictionary approach* (SD) depend on the concrete data set. The higher the skew of the chunk usages, the better a dictionary compression performs.

We simulate this approach by assigning a 24-bit code word to each chunk with an entropy smaller than 85% of $log_2(n)$ where n denotes the number of chunks. Once a code word is assigned in the simulation, it is never deactivated or revoked.

Using this method, 0.36% of the chunks in the *ENG* data set have an assigned code word after 20 backup runs. The file recipe index size is reduced by 14.2%. In the *HOME1* data set, 0.25% of the chunks have a code word assigned by this method resulting in a compression ratio of 9.0%. In the *HOME2*, 0.07% of the chunks have a code word, which compresses the file recipes by 3.6%.

We need to store an additional index to resolve the code word back to its fingerprint. The index needs to store the code word and the corresponding 20 byte fingerprint. Assuming that 0.4% ($\approx 2^{-8}$) of the fingerprints are shortened, the reverse index can be stored in main memory. For a 8 TB configuration with around 2^{30} chunks, 4 million chunks get a code word assigned. Therefore, the code word index has a size of approximately 128 MB if full fingerprints are used or 32 MB if this approach is combined with the page-based approach.

The *statistical prediction approach* (SP) is simulated by assigning predictions after each backup run. Once a prediction is assigned, it is not changed. The compression ratios with $k = 2$ fields for the Misra-Gries algorithms are promising, the approach alone leads to a compression between 69.2% (*HOME2*) and 82.2% (*ENG*).

Increasing the Misra-Gries parameter k results in a better estimation of the most frequent following fingerprint and therefore in better predictions for the compression. With $k = 4$, the compression improves slightly: to 82.5% (*ENG*), 77.6% (*HOME1*), and 69.3% (*HOME2*). Larger values for k do not increase the compression substantially. Even if the full order-1 statistic is maintained (instead of using the Misra-Gries estimation) the compression only increases to 82.6%, 77.6%, and 69.5%.

The *statistical prediction approach* stores additional data in each chunk index entry: The k fingerprints/counter pairs for the Misra-Gries algorithm and a selected prediction fingerprint. Therefore, around 68 bytes need to be added for $k = 2$. This overhead can be reduced to $\approx$ 20 bytes if combined with the *page-oriented approach*.

The statistical approaches can be sensitive to long-term aging effects where, e.g., a fingerprint is no longer used in new backups and its entropy is increasing so that is would not get a code word assigned. This aging effects usually arise slowly. For the vast majority of the fingerprints, the dictionary and the predictions remain constant for a long time. In the *HOME1* data set, covering 15 weeks, on average 0.7% of the dictionary code words are deactivated per backup run, assuming that a code word is deactivated immediately after it crosses the entropy threshold. If the prediction is changed as soon as a different fingerprint is estimated as more common successor, on average 0.04% of the chunks change their prediction per week. A full evaluation of the aging effects is beyond the scope of this paper.

Table 3: Compression ratios (for all combinations)

ZC	PB	SD	SP	HOME1	HOME2	ENG
Yes				5.4%	2.6%	12.6%
	Yes			80.0%	80.0%	80.0%
		Yes		9.0%	3.6%	14.2%
			Yes	77.3%	69.2%	82.2%
Yes	Yes			80.1%	80.0%	80.2%
Yes		Yes		9.1%	3.6%	15.9%
Yes			Yes	77.3%	69.2%	84.1%
	Yes	Yes		80.5%	80.2%	80.8%
	Yes		Yes	92.2%	90.9%	93.0%
		Yes	Yes	79.0%	69.4%	83.7%
Yes		Yes	Yes	79,1%	69.5%	84.3%
Yes	Yes		Yes	92.2%	90.9%	93.3%
Yes	Yes	Yes		80.6%	80.2%	82.2%
	Yes	Yes	Yes	92.2%	90.9%	93.1%
Yes	Yes	Yes	Yes	92.3%	90.9%	93.3%

None of the approaches causes any additional IO operations or require extensive locking in the critical path. The computations are fast and could be parallelized. It is unlikely that the computations are a bottleneck for the system throughput.performance impact. All approaches except the zero-chunk suppression have to store additional information, e.g., in the chunk index. This overhead, which grows with the physical storage capacity, is amortized by the compression savings.

The approaches can be combined: A fingerprint is always replaced by its shortest code word and, thus, can achieve the highest compression. If all approaches are applied, the file recipes of all data sets are compressed by more than 90%. Most savings are created by the statistical prediction and the page-oriented approach. These two combined achieve a compression that is within 0.3% of the compression if all four approaches are combined. The already established approach of zero chunk suppression has only a small effect if it is used in combination with other compression methods. Similarly, the method with significant main memory overhead, the statistical dictionary approach, does not provide a substantial additional compression if it is used in combination.

4 Limitations

The approaches are designed to fit into the system architecture of the dedupv1 data deduplication system [17].

They are not limited to this system environment, but are not applicable in all cases. Without specific knowledge of internal design decisions, it is impossible to claim that the approaches work for a specific alternative system.

In this section, we discuss some of the limitations of the compression approaches. First, all approaches assume a chunking/fingerprinting based data deduplication system. Furthermore, the main focus are backup environments. All approaches, except the statistical prediction approach, work in other environments, too. E.g., Jin and Miller observed a significant fraction of zero-chunks in virtual machine disk images [10]. However, the file recipe overhead is only getting large for full backups with a high deduplication ratio. Recipe compression may not be important in non-backup systems.

Also, the statistical approaches assume the usage of a full chunk index. Deduplication approaches that avoid requesting the chunk index for most of the chunks cannot use the approaches directly.

On the other hand, the approaches might be tweaked. One such tweak is avoiding the file recipe compression when new data is written. Instead a background process may rewrite recently written file recipes. One property of the statistical compressions is that most of the compression is preserved even if the statistical model is a bit outdated.

It is an open problem how these compression approaches can be conceptually combined with approaches as the container-locality caching approach by Zhu et al. [31]. Another open problem are long-term aging effects. The current data indicates that the long-term effects can be handled using the deactivation approaches sketched above, but a deeper investigation of aging effects would be worthwhile.

5 Related Work

Compression techniques are used by most deduplication systems to reduce the size of chunks after the duplication identification step [17,21,28,31].

Multiple ways to reduce the size of the chunk index have been proposed. Sparse indexing reduces the size of the chunk index by only storing there a subset of the chunks [12]. Another approach is to store only short fingerprints in an in-memory chunk index and to additionally compare the chunk data byte-by-byte to be safe from (in this setting likely) hash collisions [2].

Balachandran and Constantinescu propose to use runs of hashes for file recipe compression [3]. If a run of hashes occurs twice in a data stream, they replace it with the fingerprint of the first chunk and the length of the repeated sequence. Constantinescu et al. propose to combine repeated sequences of adjacent chunks to super-chunks [5]. Any chunk is then either merged into a super chunk or it is not repeated. However, both works do not describe how to find the runs or super chunks in an efficient manner.

JumboStore reduces the size of file recipes by chunking them into indirection nodes [6]. These nodes can be reused for identical and similar files.

The skew in the chunk usage distribution and the popularity of the zero chunk have been noted before [8, 10, 11,16,29]. Wei et al. replace the zero chunk with a build-in codeword [29].

Patterson proposed using two separate index structures: One mapping from the fingerprint to a sequentially increased code word and one from the code word to the on-disk location [20]. The code word length is in the order of the logarithm of the number of stored chunks. Therefore, it produces code words of similar length than the page-based approach.

Also, the page-based approach is similar to dictionary encoding in databases (see, e.g., Abadi et al. [1]). However, database dictionary encoding is usually only applied to columns with a limited, fixed cardinality and it uses a separate lookup index. Our approach works for billions of chunks without additional index lookups.

The statistical dictionary approach are related to classical Huffman codes [9] in that both aim to reduce the code word size based on the order-0 statistic of the data. The statistical dictionary approach differs in the way the code words are assigned. Furthermore, it relaxes the compression for the usage as file recipe compression.

Compression of index data structures is state-of-the-art in areas like databases (e.g., [1, 7]) or information retrieval (e.g., [15, 30]). The techniques used there, e.g., page-local compression, run-length encoding, and delta-compression within pages, are not directly applicable in data deduplication.

6 Conclusion

We presented new compression approaches for file recipes in deduplication systems. A combination of these approaches allows shrinking the file recipes by up to 93%. Since file recipes can be responsible for a significant fraction of the physical disk usage of deduplication systems, these results enable significant overall savings.

7 Acknowledgments

We would like to thank our shepherd and our reviewers for their helpful feedback. This work was partially supported by the German Federal Ministry of Economics and Technology (BMWi) under project grant SIMBA and by the EU commission within the SCALUS early researchers graduate school.

References

[1] Abadi, D., Madden, S., and Ferreira, M. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (2006), pp. 671–682.

[2] Alvarez, C. NetApp Deduplication for FAS and V-Series Deployment and Implementation Guide. Tech. rep., NetApp, 2009.

[3] Balachandran, S., and Constantinescu, C. Sequence of Hashes Compression in Data De-duplication. In *Proceedings of the 18th Data Compression Conference (DCC)* (2008), pp. 671–682.

[4] Bhagwat, D., Eshghi, K., Long, D., and Lillibridge, M. Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup. In *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)* (2009), pp. 1–9.

[5] Constantinescu, C., Pieper, J., and Li, T. Block Size Optimization in Deduplication Systems. In *Proceedings of the 19th Data Compression Conference (DCC)* (2009), p. 442.

[6] Eshghi, K., Lillibridge, M., Wilcock, L., Belrose, G., and Hawkes, R. Jumbo Store: Providing Efficient Incremental Upload and Versioning for a Utility Rendering Service. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)* (2007), pp. 123–138.

[7] Holloway, A. L., Raman, V., Swart, G., and DeWitt, D. J. How to Barter Bits for Chronons: Compression and Bandwidth Trade Offs for Database Scans. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (2007), pp. 389–400.

[8] Hong, B., and Long, D. D. E. Duplicate Data Elimination in a SAN File System. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)* (2004), pp. 301–314.

[9] Huffman, D. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE 40*, 9 (1952), 1098–1101.

[10] Jin, K., and Miller, E. L. The Effectiveness of Deduplication on Virtual Machine Disk Images. In *Proceedings of the 2nd Israeli Experimental Systems Conference (SYSTOR)* (2009), p. 7.

[11] Kaiser, J., Meister, D., Brinkmann, A., and Effert, S. Design of an Exact Data Deduplication Cluster. In *Proceedings of the 28th IEEE Conference on Mass Storage Systems and Technologies (MSST)* (2012), pp. 1–12.

[12] Lillibridge, M., Eshghi, K., Bhagwat, D., Deolalikar, V., Trezise, G., and Camble, P. Sparse Indexing: Large Scale, Inline Deduplication using Sampling and Locality. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)* (2009), pp. 111–123.

[13] Lokeshwari, Y. V., Prabavathy, B., and Babu, C. Optimized Cloud Storage with High Throughput Deduplication Approach. In *Proceedings of the International Conference on Emerging Technology Trends* (2011), pp. 32–37.

[14] Manber, U., et al. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference* (1994), San Fransisco, CA, USA, pp. 1–10.

[15] Manning, C. D., Raghavan, P., and Schütze, H. *Introduction to Information Retrieval*. Cambridge University Press, 2009.

[16] Meister, D., and Brinkmann, A. Multi-level Comparison of Data Deduplication in a Backup Scenario. In *Proceedings of the 2nd Israeli Experimental Systems Conference (SYSTOR)* (2009), pp. 8:1–8:12.

[17] Meister, D., and Brinkmann, A. dedupv1: Improving Deduplication Throughput using Solid State Drives. In *Proceedings of the 26th IEEE Conference on Mass Storage Systems and Technologies (MSST)* (2010), pp. 1–6.

[18] Meyer, D. T., and Bolosky, W. J. A Study of Practical Deduplication. *ACM Transactions on Storage 7*, 4 (2012), 14:1–14:20.

[19] Misra, J., and Gries, D. Finding Repeated Elements. Tech. rep., Cornell University, Ithaca, NY, USA, 1982.

[20] Patterson, H. Data Storage using Identifiers. US Patent 7,143,251, November 2006.

[21] Quinlan, S., and Dorward, S. Venti: A new Approach to Archival Storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)* (2002).

[22] Raab, M., and Steger, A. "Balls into Bins" - A Simple and Tight Analysis. In *Proceedings of the 2nd International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)* (1998), pp. 159–170.

[23] Shilane, P., Wallace, G., Huang, M., and Hsu, W. Delta compressed and deduplicated storage using stream-informed locality. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems* (2012), pp. 10–15.

[24] Stonebraker, M., Held, G., Wong, E., and Kreps, P. The Design and Implementation of INGRES. *ACM Transactions on Database Systems 1*, 3 (1976), 189–222.

[25] Tan, Y., Jiang, H., Feng, D., Tian, L., and Yan, Z. CABdedupe: A Causality-Based Deduplication Performance Booster for Cloud Backup Services. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2011), pp. 1266–1277.

[26] Tolia, N., Kozuch, M., Satyanarayanan, M., Karp, B., Bressoud, T., and Perrig, A. Opportunistic use of Content Addressable Storage for Distributed File Systems. In *Proceedings of the 2003 USENIX Annual Technical Conference* (2003), pp. 127–140.

[27] Ungureanu, C., Atkin, B., Aranya, A., Gokhale, S., Rago, S., Calkowski, G., Dubnicki, C., and Bohra, A. HydraFS: A High-throughput File System for the HYDRAstor Content-addressable Storage System. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)* (2010), pp. 225–238.

[28] Wallace, G., Douglis, F., Qian, H., Shilane, P., and Smaldone, S. Characteristics of Backup Workloads in Production Systems. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)* (2012), pp. 1–16.

[29] Wei, J., Jiang, H., Zhou, K., and Feng, D. MAD2: A Scalable High-throughput Exact Deduplication Approach for Network Backup Services. In *Proceedings of the 26th IEEE Conference on Mass Storage Systems and Technologies (MSST)* (2010), pp. 1–14.

[30] Witten, I. H., Moffat, A., and Bell, T. C. *Managing Gigabytes*. Morgan Kaufmann, 1999.

[31] Zhu, B., Li, K., and Patterson, H. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)* (2008), pp. 18:1–18:14.