ChunkFS: A Tool for Data Deduplication Methods Comparison

Oleg Piletskii, Viacheslav Gorikhovskii
Saint Petersburg State University
Saint Petersburg, Russia
piletskii.oleg@gmail.com, v.gorikhovsky@spbu.ru

***Abstract*—The dramatic rise in amount of data worldwide over the past years is a critical issue for storage and backup systems with no obvious and simple solution available. One of the main techniques to effectively store large amounts of data is deduplication, based on eliminating redundant data, of which there is a lot. The most expensive stage of deduplication process is chunking, taking up to 90% of time. Many algorithms have emerged in the recent years, with the goal to optimize space savings and throughput of the process, stating better effectiveness each time. Despite there being so many chunking algorithms, there are very few systems that can be used to compare them with each other.**

In this paper we present ChunkFS, a tool to compare deduplication techniques that allows to easily integrate different chunking and hashing algorithms, as well as storage types, and gather the necessary metrics to determine the most suitable one. We use it to compare some of the best performing algorithms and find out that SuperCDC outperforms every other one in speed, but not in speed savings, although with many parameters that can be tuned, it hints that with further research using the tool better results can be achieved. Besides Content Defined Chunking algorithms, other techniques can be compared using ChunkFS, such as Frequency Based Chunking, but that is out of scope of this paper.

I. Introduction

In the recent years the amount of data created and stored has become exceedingly large. According to several sources [1], [2], the global datasphere, which is the combination of data generated, captured or replicated through the digital content from all around the globe, contained 33 zettabytes of data in 2018, and its size will increase to 175 zettabytes in 2025. As a result of that rapid increase storing all that data becomes increasingly difficult and costly, especially for the IT companies and industries, which also want to be able to work on it and retrieve it quickly enough [3]. One of the main scenarios for data storage are backups, which have to be made regularly to protect the sensitive data and therefore take much space. To fight that, many techniques that eliminate redundant data are used, such as deduplication and compression [4].

A. Data Deduplication Methods

Data deduplication is a way of compressing information by eliminating repeating blocks of data, usually at least a few kilobytes in size. One flaw of this method is that it cannot detect duplicates of less than minimal block size, but it is a trade-off for the speed that can be achieved on the large workloads. The main way it is done consists of 5 stages, which are splitting the data stream into small chunks of data, calculating their fingerprints, indexing the chunks and removing repeated chunks, and saving them to the disk or other underlying storage.

The most expensive stage by time and resources is chunking since the whole file must be read byte-by-byte, performing some calculations in the process. The easiest way to implement it is by using Fixed Size Chunking [5], which splits data evenly. This approach, although very fast, is not very good at finding repeating data. Moreover, if there is just a single byte added to the data somewhere in the middle, then all chunks after it are considered new, even though they are in fact unchanged. This is called boundary-shifting problem, showcased on Fig. 1. Here, the data is almost unmodified, but all chunks, starting from the second, are considered new.

To combat this problem, another approach is used. Most chunking algorithms, published in the recent years, belong to the Content Defined Chunking algorithm family, which is based on splitting data in chunks based on the local content of the data stream. If the local content is unchanged, then the chunk boundaries are also unchanged, and the data deduplication ratio does not suffer [6].

To determine chunk boundaries, in most algorithms sliding window is used. Hash of the sliding window is calculated, and if it satisfies some condition, the boundary is set and the chunk is cut.

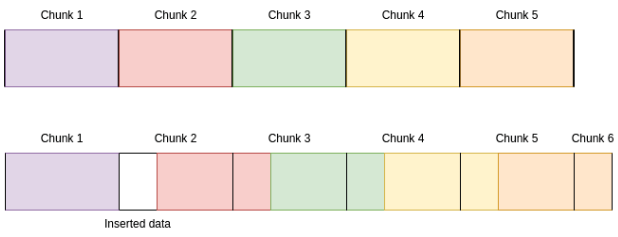

Fig. 1. Boundary shifting problem

B. Algorithms comparison

The main metrics by which CDC algorithms can be compared are deduplication ratio, which shows how well the algorithm finds duplicate data in the dataset, and the speed with which it processes the data. Other metrics include average

chunk size, chunk size distribution, and the deduplication ratio which also includes the keys' sizes.

However, there are still some questions which those metrics cannot give an answer to. For example, how would a chunking algorithm influence a whole file system? How do different types of storage influence algorithm efficiency? Can it help with faster writing or reading to the disk?

The main focus in the CDC comparison papers was the algorithm effectiveness in some small environment, with the main scenario being running the algorithm on some data stream a single time and gathering some metrics. This approach does not take into consideration the effects an actual system may have, such as interaction of the chunking subsystem with other subsystems.

In this paper we present ChunkFS, a simple in-memory file system. Its main purpose is to provide a unified way to compare different deduplication methods, most notably CDC algorithms, in an integrated environment with many additional features. It is written in Rust programming language, utilizing its modern ideas and memory safety.

II. Related works

In this section we analyze some of the existing chunking algorithms and show their advantages and disadvantages.

1) RabinCDC: RabinCDC [7] is one of the first content defined chunking algorithms, based on calculating sliding window's polynomial Rabin hash. It is quite simple in implementation, but the hash calculation is inefficient and expensive.

2) GearCDC: GearCDC [8] provides another way to calculate sliding window hash that uses only a single operation of bit shift, one array indexing operation and one addition. This allows for 3x speed increase and no loss in deduplication ratio. It is also used as a base for other optimized algorithms.

3) LeapCDC: LeapCDC [9] shows another way of generating chunks that also provides a performance boost, although it is slightly harder to modify than GearCDC, and uses slightly different technique than the usual sliding window. For every byte there is a check for some number of so-called windows of the same size located before said byte, each shifted by a single byte. If all windows satisfy a preset condition, then the chunk is cut. If not, the bytes corresponding to number of checked windows can also be skipped, since those windows certainly will not satisfy the condition. Window check is 5 index operations and 4 XOR operations.

4) RapidCDC: RapidCDC [10] uses the fact that the similar chunks are very likely to be located next to each other, forming repeating groups of chunks. With the hash of the next chunk it also records the size of the next chunk, which serves as a hint for finding a possible border in order to not skim through the known chunk using the sliding window. And if it does not find a single group of chunks, its performance only decreases by 10 percent compared to GearCDC. This algorithm uses the idea of storing additional information besides the chunk border itself.

5) FastCDC: FastCDC [11] is an improved and heavily modified version of GearCDC and is one of the fastest CDC algorithms to date. It fixes the following problems:

- Small size of the sliding window, which negatively impacts the deduplication ratio, since the chunk border depends on less number of bytes.
- Even though hash is calculated very fast, the process of checking hash takes up to 60% of time that the algorithm takes to complete.

In order to fix those problems the window is increased by padding the mask with zeroes. The fingerprint is then challenged against the mask, and if it satisfies a simple condition, then the chunk is cut. To reach higher deduplication ratio normalized chunking is used, which is a technique that helps to distribute chunk sizes more evenly: for chunk positions which correspond to chunks smaller than 8 kilobytes more significant bits in mask are used which causes hash judgment to be more strict. When the chunk is bigger than 8 kilobytes, less significant bits in mask are used. It causes chunk size distribution to be normalized around a specified region. Another optimization is rolling two bytes each time, one odd and one even. This causes a 30-40 percent increase in speed compared to rolling by a single byte.

6) QuickCDC: QuickCDC [12] utilizes similar strategy to RapidCDC but doesn't require similar chunks to be present in succession which allows it to work on isolated repeating chunks, whereas RapidCDC cannot do that. The main idea behind QuickCDC is to find repeating chunks using the first and last three bytes, storing this information in the tables along with chunk sizes, to find the corresponding chunks easily.

7) SuperCDC: SuperCDC [13] combines the ideas of FastCDC and RapidCDC. It takes calculation-efficient processing with a stream informed design, meaning that it incorporates fast rolling hash, smart chunking judgment and chunk size normalization from FastCDC. It builds upon its ideas, utilizing 3 chunking masks to reduce number of max size chunks. SuperCDC also utilizes a lightweight chunking map which is maintained in memory and provides next chunk size hints which allows skipping whole sequences of chunks.

8) UltraCDC: UltraCDC [14] is an algorithm that excels on strings with low entropy, the type of data that other algorithms struggle on. It differs from other algorithms in a sense that it doesn't compute hash of the sliding window but instead uses Hamming distance from the symmetric predefined pattern as a chunking condition. To find low entropy strings, it incorporates jumping, which is achieved by having two windows as buffers. Low entropy strings are then identified by the number of jumps exceeding a predefined value. Other optimizations such as normalized chunking are used.

9) Speculative Jump: Jump-based chunking [15] is an approach that is an improvement of the LeapCDC. It is an optimization that can be inserted into any other CDC algorithm. The approach is based on the idea that sliding the window byte-by-byte is often unnecessary and the portions of the input can be skipped by jumping, but unlike LeapCDC, it suggests infrequent jumps that don't break the continuity of

data. Another key design is embedded masks, i.e. constructing a bigger mask to overlap both mask that determines the jump condition and the mask that determines chunk cut point.

10) SeqCDC: SeqCDC [16] is an algorithm that is geared towards chunks with larger sizes. It operates in a different way from the other algorithms, not using hash and looking for the sequences of monotonically increasing/decreasing bytes, depending on the mode of operation. It can also skip certain regions of data if it detects a byte that breaks the sequence, e.g. a smaller byte if the increasing mode of operation is chosen. Once a sequence of the set length that corresponds to the mode of operation is found, the chunk is cut.

A. Algorithms comparison

There is a large amount of different chunking algorithms, some of which were not present in this paper. Their description can be found in other review papers, such as [17].

Despite there being such a big number of different algorithms which state comparable effectiveness and speed to each other, there is very few comparison systems. Most of them emerged in the recent years, for example DedupBench [18]. It is a set of scripts written using C++ which let the user deduplicate a single data stream, with configurable chunking and hashing algorithms. New algorithms can be quite easily integrated with the DedupBench and tested. It can collect the metrics listed, such as deduplication ratio and throughput.

Another one is cdc-algorithm-tester [19], written using Rust and R. It is a CLI app which allows gathering the same metrics as before and contains implementations of many different algorithms.

Those tools allow comparing algorithms and gathering metrics for a simple scenario which consists solely of running an algorithm on a data stream. They try to get rid of other factors that may influence the results, for example by putting the whole file in RAM and not writing anything to disk in the process. This approach doesn't take into account how a file system may actually influence the chunking throughput and the effects it may have on algorithm effectiveness. This is fixed by ChunkFS.

III. ChunkFS Description

ChunkFS is a simple file system with a single directory storing all the files. It allows writing and reading to the files, files are accessed by their names. When the user wishes to create a file, a name and a chunker must be provided, with which the data in the file will be deduplicated. After that a file handle is received to the said file, using which write and read operations can be performed. After the process has finished, the handle can be closed, then re-opened in read-only mode allowing to read the contents, verifying that they are correct.

Inside, ChunkFS is divided into two main parts, ChunkStorage and FileLayer. The first one is responsible for storing the data, the chunks, as the name says, and is common for all the files in the system. The second one is responsible for storing file metadata and it provides access to the content of the files to the user. Those systems communicate using segment hashes and are united in a structure called FileSystem, which is accessible by the user and defines the main operations that can be conducted.

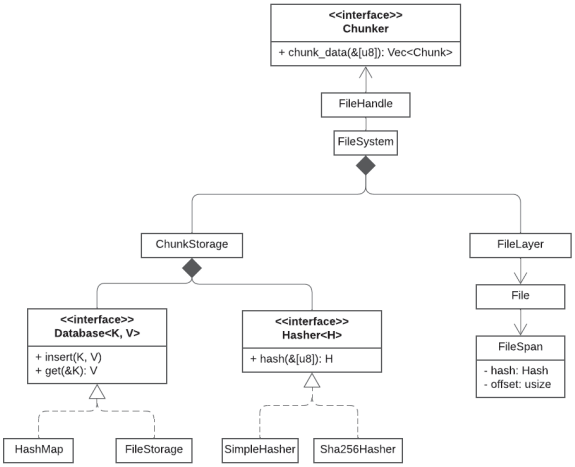

Fig. 2. ChunkFS architecture

The architecture of the file system in shown on Fig. 2 using pseudo UML. To provide variability in storing data and using different algorithms, ChunkFS provides three main traits:

- Database is an abstraction over a key-value storage that provides main operations: inserting, getting, checking and deleting a key-value pair. This allows using different data structures as a chunk storage, e.g. a hash map, a B-Plus tree, or a simple file storage.
  There is also a further abstraction over a database that can be iterated, provided using the IterableDatabase trait. It is useful for cases such as collecting information about chunk distribution, or performing some analysis on the chunks.
- Chunker is an abstraction over a chunking algorithm. An object that implements that trait is used within an open file session, but is assumed to only store information about the algorithm it encapsulates, and maybe some other information, such as QuickCDC table. That enables reusing it with different files and reusing its stored contents.
  Its main method receives a data slice as an input and an empty allocated vector, and returns a vector of chunks as an output. Chunk in this case is information about the start position and the length of the chunk, hence it only provides information about the given slice, and does not store actual data.
  Another method returns an estimate chunk count for the given slice, so that the vector which will store the chunks can be pre-allocated before the chunking process. This is used to get rid of the allocation time when measuring chunking speed.
- Hasher is an abstraction over a hashing algorithm, and it is used to hash the chunks, before putting them into the

database.

Files in FileLayer are stored as a collection of spans, using which the file can be reconstructed and returned to the user. The span represents a segment of the file, containing its hash and offset.

Some of the content defined chunking algorithms were implemented for ChunkFS, such as Leap-based CDC, SeqCDC, SuperCDC, UltraCDC, RabinCDC. They can be found in a separate crate cdc-chunkers on crates.io, with the source code available on GitHub[1]. To create them, the minimum, average and maximum chunk sizes must be provided as parameters. FastCDC implementation that can be found on crates.io was integrated with ChunkFS. It was made possible by implementing Chunker trait for all of them, allowing their usage within the file system.

ChunkFS also allows for further analysis and deduplication of chunks by having a trait that processes chunks from the source database and sends the modified data into another database. This is a process called scrubbing that can be ran at any time.

Benching tools are also bundled with ChunkFS, available via the bench module. It contains a structure called CDCFixture that contains a file system. It provides some basic operations to collect the most necessary metrics, such as measuring write time, read time, chunk time and hash time on a single write and on a repeated write, calculating deduplication ratio on a dataset using some specified chunker, and calculating information about the data that's already in the file system, such as chunk size distribution or average chunk size. Those operations, except for the last one, take a chunker and a dataset as an input.

Dataset is a separate structure that only stores the name, size and path of the dataset file. It is then opened using a separate method that gives out a File instance that can be used to read the contents, only reading the necessary portions into RAM and not the whole file.

To collect metrics, the user can write a simple Rust script that provides a dataset and feeds it into the fixture to then gather the information. Multiple examples are provided in the root directory in the crate.

Another option is to use the bundled command-line tool that allows either inputting all necessary parameters by hand, such as path to dataset, chunk sizes, chosen algorithms and database, or to run from a predefined configuration file. The information on how to use it is provided in the repository. It supports the scenarios of writing a dataset to the storage, it can be chosen to clean up all the existing data in the storage, or to write over and over again. It can be specified to run the script several times. Data can be written to the database before the measurements start. After the tool has finished, all collected metrics are saved to the table, path to which is specified by the user.

Besides gathering metrics, ChunkFS provides ways to generate datasets. The methods are

- Generating a dataset using fio tool with the set size and deduplication percentage, i.e. the percentage of repeating buffers.
- Generating a dataset with the set size based on some distribution.
- Generating a dataset from some other that is already written to the file system, by rearranging the segments and reaching the desired deduplication ratio.

ChunkFS is available on GitHub[2] and crates.io[3].

IV. Evaluation

1) Datasets: For complex evaluation, multiple datasets sources are used that resemble common backup data. Those include

- Virtual Machine images [20]
- OpenStreetMaps backups [21]
- Linux kernel [22]
- Enron, the set of emails and messages [23]
- Docker images from the official website, e.g. WordPress [24] and nodejs [25]
- Website snippets

Those datasets provide a wide variety of real world data that contains lots of repeating information and hence can be used to compare different algorithms.

For the current evaluation, the following datasets have been chosen:

- Arch Linux images dated 1st December 2024 and 15th December 2024
- Linux kernel version 6.12, different release candidates
- OpenStreetMaps data for Russia, two full backups

We use some of the dataset snippets for our current evaluation. Since chunking algorithms work on single files, those workloads were put in tarballs.

2) Evaluated algorithms: The following algorithms have been evaluated: RabinCDC, LeapCDC, FastCDC, UltraCDC, SuperCDC, SeqCDC. The minimum, average and maximum chunk sizes were chosen to be 4, 8 and 16 KB respectively. For hashing SHA-256 implementation provided by sha2 crate [26] was used. RAM was used as a storage in order to minimize effects of the disk on the throughput.

3) Testbed: Algorithms were evaluated on a machine with 40 GB of RAM and 12th Gen Intel Core i5-1240P. Ubuntu 24.04 was used as an operating system, code was compiled using rustc 1.84.0.

4) Metrics: The following metrics were calculated for each workload:

- Deduplication ratio: it is calculated as ratio of dataset size to size of the data after it was written to the file system.
- Full deduplication ratio: it also includes key sizes used to access the chunks.
- Average chunk size: average size of all chunks in the storage.

[1]https://github.com/Piletskii-Oleg/rust-chunking

[2]https://github.com/Piletskii-Oleg/chunkfs

[3]https://crates.io/crates/chunkfs

TABLE I. DATASETS CHOSEN FOR CURRENT EVALUATION

Dataset	Size	Description
LNX	12.8 GB	Linux kernel version 6.12, different release candidates
OSM	8.2 GB	OpenStreetMaps data for Russia, two full backups
VM	4.2 GB	Arch Linux images dated 1st December 2024 and 15th December 2024

- Chunk sizes distribution: it is plotted on a graph, showing how many chunks there was of roughly the same size.
- Chunking throughput: it shows time taken for the algorithms to chunk the entire dataset.
- Hashing throughput: it shows time taken for the algorithms to hash the resulting chunks.
- Write and read speed of the file system: how much time was taken for writing the dataset into the file system and then reading it into a byte vector from the file system.

The main scenario which was evaluated and for which metrics were collected is writing a file to the file system and reading it back.

A. Space Savings

One of two main characteristics of the algorithm is how well it can save space on the disk, measured using deduplication ratio and full deduplication ratio. Those values do not differ that much, as can be seen from Fig. 3, and the algorithms achieve comparable deduplication ratio to each other, although it can be seen that for Linux kernels LeapCDC and SeqCDC are slightly better at finding duplicates, while for Arch Linux images SuperCDC and UltraCDC show slightly worse results than other algorithms.

B. Average Chunk Size

Average chunk size is shown on Fig. 5. Although the parameter was set to 8 KB in all scenarios, that value is only achieved by LeapCDC, with other algorithms either having too big or too small chunks. But the value is notably preserved across all datasets, which shows consistency of the algorithms.

C. Algorithmic Throughput

Another very important characteristic of the CDC algorithm is its throughput, and how it impacts the overall system efficiency. As seen on Fig. 4, chunking throughput differs drastically for each algorithm, where the clear winner is SuperCDC algorithm. Despite being a bit weaker than other algorithms in deduplication ratio, it reaches speeds 3 and 4 times higher than other ones on OpenStreetMaps and VM datasets, although it doesn't show that good of results on Linux kernels dataset.

Other algorithms unfortunately do not perform as well as SuperCDC, with FastCDC being the second fastest and LeapCDC and UltraCDC being comparatively equal, but that can be attributed to chunk sizes, and it's possible that with other parameters results could be better.

RabinCDC is the slowest algorithm of them all, which is due to inefficient inner hash calculation to determine the chunk boundary, which other algorithms handle in a different, more efficient way.

D. Chunk Distribution

Chunk distribution is shown on Fig. 7 for SuperCDC as an example. It shows that for LNX dataset there is quite a lot of chunks with maximum allowed size, which impacts the overall performance of the algorithm, while for two others most chunks are very small, which leads to frequent 4 KB jumps which help traverse the dataset much faster. This may also lead to the deduplication ratio being slightly lower compared to other algorithms.

E. ChunkFS Throughput

Fig. 6 shows the write speed and read speed of different datasets into and out of ChunkFS respectively. Differences in throughput between different chunking algorithms are seen in write speed into ChunkFS, although SuperCDC doesn't outperform other algorithms as much as it did in clean form, and its impact on the speed is not as severe. Even then it is a clear winner.

Overall, the speed has fallen from around 1 GB/s for most algorithms down to 300-400 MB/s for all of them, with RabinCDC only reaching 200 MB/s. The decrease in speed can be attributed to there being other long operations such as hashing the chunks, cloning some values and memory allocation.

Read speed is basically independent of an algorithm, although that can be due to how it is calculated: all file spans and then the corresponding chunks are continuously read into the memory, while writing is done in 1 megabyte blocks. Another interesting difference is that on VM dataset the throughput is approximately 20% faster than for the other ones, probably due to its smaller size.

V. CONCLUSION

In this paper we presented ChunkFS, a tool specifically designed for benchmarking and analyzing different deduplication methods, such as Content Defined Chunking algorithms. We show the main metrics which determine how well algorithms behave in comparison to each other. We introduce the most commonly used data sources which can be used to run the algorithms on, that resemble real backup data.

One of the main drawbacks of existing comparison tools is that they only analyze the raw algorithm efficiency, not considering how a file system can impact the results. This is taken into consideration in ChunkFS, which is, in itself, a file system, albeit a simple one. It allows for analysis on an integrated scenario with different storages and the interaction

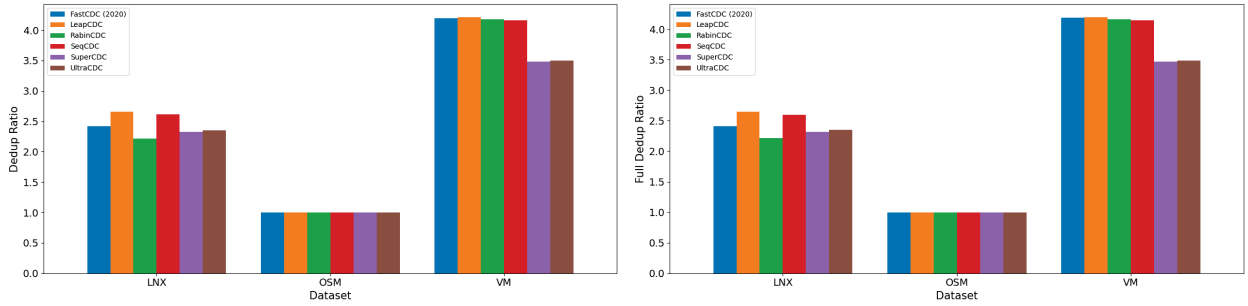

Fig. 3. Deduplication ratio and full deduplication ratio

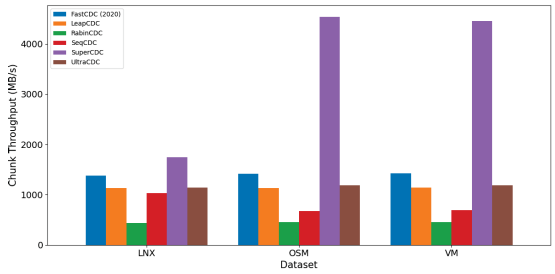

Fig. 4. Algorithmic throughput

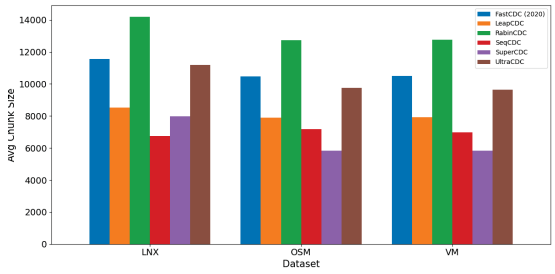

Fig. 5. Average chunk size

between file system components. Another improvement over existing tools is being able to collect all the necessary metrics using a simple command and a configuration file, and and easy-to-parse result.

We have conducted an experiment using ChunkFS, utilizing 6 different chunking algorithms on several datasets, and saw that SuperCDC is the most effective algorithm in terms of raw throughput, while other algorithms perform better when it is needed to save as much space as possible. This data is more or less consistent with what is seen on an integrated environment in ChunkFS, in terms of reading and writing speed.

It follows that the ChunkFS tool can be used to effectively compare different chunking algorithms and collect the necessary metrics by using simple Rust scripts or command-line interface with configuration files. It was only tested on RAM storage, but it supports other kinds of key-value storage types such as databases based on LSM or other kinds of trees, which can themselves introduce many side effects and change the evaluation outcome. That should be studied in the future, for example effects of sled [27] or rocksdb [28], which are most prominent databases for rust.

There are some Content Defined Chunking algorithms that were not evaluated in this study, but should be. Besides these, other deduplication techniques include Similarity Based Chunking, based on delta coding and finding similarities in chunks, and Frequency Based Chunking. Those are some of the yet unexplored but very promising fields, which can also be analyzed using ChunkFS, as it provides the necessary machinery for them.

Acknowledgment

This work was supported by St. Petersburg State University (Pure ID 116636233).

References

[1] P. Prajapati and P. Shah, “A review on secure data deduplication: Cloud storage security issue,” *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 7, pp. 3996–4007, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157820305140

[2] W. Xia, L. Pu, X. Zou, P. Shilane, S. Li, H. Zhang, and X. Wang, “The design of fast and lightweight resemblance detection for efficient post-deduplication delta compression,” *ACM Trans. Storage*, vol. 19, no. 3, Jun. 2023. [Online]. Available: https://doi.org/10.1145/3584663

[3] H. B. Jehlol and L. E. George, “Big data backup deduplication: A survey,” *Int. J. Sci. Res. Sci. Eng. Technol.*, pp. 174–191, 2022.

[4] X. Zou, W. Xia, P. Shilane, H. Zhang, and X. Wang, “Building a high-performance fine-grained deduplication framework for backup storage with high deduplication ratio,” in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 19–36.

[5] P. Krishnaprasad and B. A. Narayamparambil, “A proposal for improving data deduplication with dual side fixed size chunking algorithm,” in *2013 Third International Conference on Advances in Computing and Communications*, 2013, pp. 13–16.

[6] D. Feng, *Data deduplication for high performance storage system*. Springer, 2022.

[7] A. Muthitacharoen, B. Chen, and D. Mazières, “A low-bandwidth network file system,” vol. 35, 12 2001, pp. 174–187.

[8] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Y. Zhou, “Ddelta: A deduplication-inspired fast delta compression approach,” *Performance Evaluation*, vol. 79, pp. 258–272, 2014, special Issue: Performance 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166531614000790

[9] C. Yu, C. Zhang, Y. Mao, and F. Li, “Leap-based content defined chunking — theory and implementation,” in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015, pp. 1–12.

[10] F. Ni and S. Jiang, “Rapidcdc: Leveraging duplicate locality to accelerate chunking in cdc-based deduplication systems,” in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 220–232. [Online]. Available: https://doi.org/10.1145/3357223.3362731

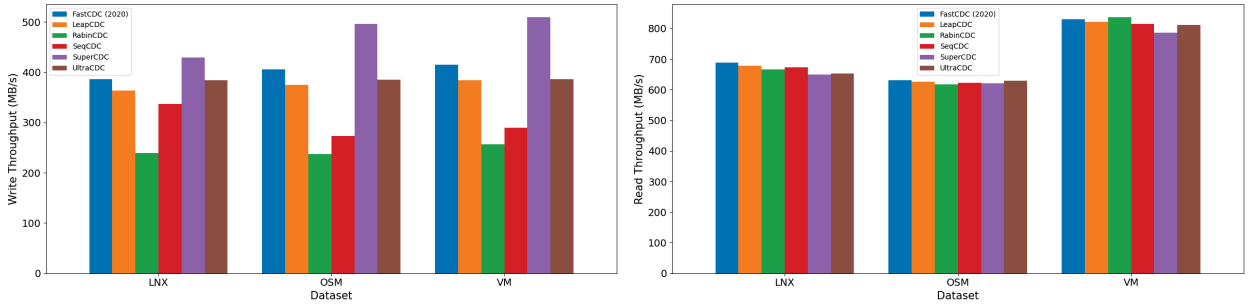

Fig. 6. ChunkFS throughput

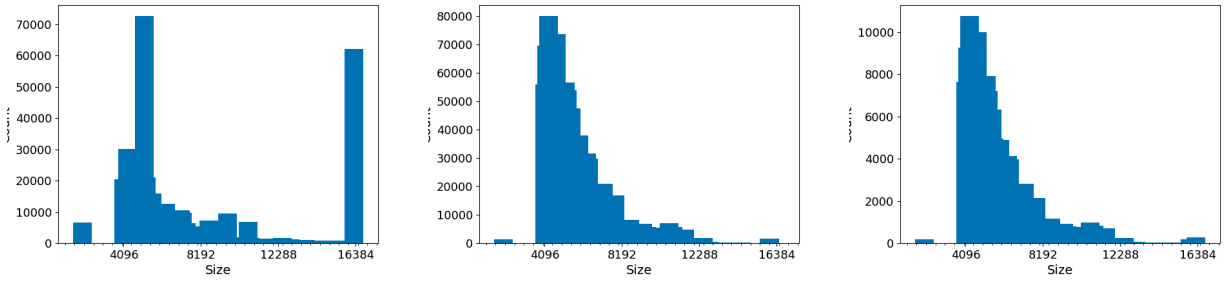

Fig. 7. SuperCDC Chunk Distribution for LNX, OSM and VM datasets

[11] W. Xia, X. Zou, H. Jiang, Y. Zhou, C. Liu, D. Feng, Y. Hua, Y. Hu, and Y. Zhang, "The design of fast content-defined chunking for data deduplication based storage systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 9, pp. 2017–2031, 2020.

[12] Z. Xu and W. Zhang, "Quickcdc: A quick content defined chunking algorithm based on jumping and dynamically adjusting mask bits," in *2021 IEEE Intl Conf on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking (ISPA/BDCloud/SocialCom/SustainCom)*, 2021, pp. 288–299.

[13] B. Wan, L. Pu, X. Zou, S. Li, P. Wang, and W. Xia, "Supercdc: A hybrid design of high-performance content-defined chunking for fast deduplication," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*, 2022, pp. 170–178.

[14] P. Zhou, Z. Wang, W. Xia, and H. Zhang, "Ultracdc:a fast and stable content-defined chunking algorithm for deduplication-based backup storage systems," in *2022 IEEE International Performance, Computing, and Communications Conference (IPCCC)*, 2022, pp. 298–304.

[15] X. Jin, H. Liu, C. Ye, X. Liao, H. Jin, and Y. Zhang, "Accelerating content-defined chunking for data deduplication based on speculative jump," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 9, pp. 2568–2579, 2023.

[16] S. Udayashankar, A. Baba, and S. Al-Kiswany, "Seqcdc: Hashless content-defined chunking for data deduplication," in *Proceedings of the 25th International Middleware Conference*, 2024, pp. 292–298.

[17] A. H. Adhab and N. A. Hussien, "Techniques of data deduplication for cloud storage: A review," *International Journal of Engineering Research and Advanced Technology*, vol. 8, no. 04, pp. 07–18, 2022.

[18] A. Liu, A. Baba, S. Udayashankar, and S. Al-Kiswany, "Dedupbench: A benchmarking tool for data chunking techniques," in *2023 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, 2023, pp. 469–474.

[19] M. Gregoriadis, L. Balduf, B. Scheuermann, and J. Pouwelse, "A thorough investigation of content-defined chunking algorithms for data deduplication," *arXiv preprint arXiv:2409.06066*, 2024, unpublished.

[20] *OpenStack. Virtual Machine Images*. [Online]. Available: https://docs.openstack.org/image-guide/obtain-images.html

[21] *OpenStreetMaps backups*. [Online]. Available: https://wiki.openstreetmap.org/wiki/Backup

[22] *The Linux Kernel Archives*. [Online]. Available: https://www.kernel.org/

[23] *Enron Email Dataset*. [Online]. Available: https://www.cs.cmu.edu/~enron/

[24] *Wordpress Docker official images*. [Online]. Available: https://hub.docker.com/_/wordpress

[25] *nodejs Docker official images*. [Online]. Available: https://hub.docker.com/_/node

[26] *The Rust community's crate registry. Crate sha256*. [Online]. Available: https://crates.io/crates/sha2

[27] *High-performance embedded database*. [Online]. Available: https://docs.rs/sled/latest/sled/

[28] *High-performance embedded database*. [Online]. Available: https://docs.rs/rocksdb/latest/rocksdb/