SeqCDC: Hashless Content-Defined Chunking for Data Deduplication

Sreeharsha Udayashankar
s2udayas@uwaterloo.ca
University of Waterloo
Canada

Abdelrahman Baba
ababa@uwaterloo.ca
University of Waterloo
Canada

Samer Al-Kiswany
alkiswany@uwaterloo.ca
University of Waterloo
Canada

ABSTRACT

Data deduplication is critical to cloud storage providers and is widely employed to conserve server-side storage space. Data chunking is an important aspect of deduplication, being directly responsible for storage space savings and end-to-end system throughput. While deduplication systems deployed in production favor larger chunk sizes, existing data chunking algorithms are slow and offer minimal throughput increases with increasing chunk size.

We present SeqCDC, a chunking algorithm that leverages content-based data skipping and lightweight boundary judgement to improve chunking throughputs. SeqCDC's chunking throughput is higher at larger chunk sizes. Our evaluation shows that SeqCDC can improve chunking throughput by 1.5× – 3.1× over the state-of-the-art while achieving similar space savings benefits, across a variety of datasets.

CCS CONCEPTS

• **Information systems** → **Deduplication**.

KEYWORDS

Data Chunking, Content-Defined Chunking, Content-Defined Skipping, Data Deduplication

ACM Reference Format:
Sreeharsha Udayashankar, Abdelrahman Baba, and Samer Al-Kiswany. 2024. SeqCDC: Hashless Content-Defined Chunking for Data Deduplication. In *25th International Middleware Conference (MIDDLEWARE '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3652892.3700766

1 INTRODUCTION

Data generation rates have skyrocketed in recent years, leading to the explosion of the amount of data stored on the cloud [1]. Cloud storage providers employ numerous mechanisms to deal with this data influx, such as distributed file systems [2, 3], novel storage architectures [4, 5], data compression [6, 7] and data deduplication [8, 9].

Data deduplication has been widely employed in production by cloud storage providers such as Microsoft [10], EMC [11] and IBM [12]. A large percentage of the data stored by these providers is redundant [10]. Data deduplication helps identify and eliminate these redundant portions, reducing storage costs by up to 80% [11, 13]. Deduplication can be performed at the *file-level*, i.e. on entire files, or the *block-level*, after dividing files into blocks [14].

Block-level deduplication achieves far superior space savings when compared to file-level deduplication [15]. The division of files into these blocks (or *chunks*) is achieved using data chunking algorithms [15], which dictate the space savings achieved by the deduplication system as a whole. Data chunking algorithms fall into two categories: fixed-size and content-defined chunking (CDC).

Fixed-size chunking divides files into chunks of an equal pre-specified size. While this approach offers extremely high chunking throughput, it achieves poor space savings as it is vulnerable to byte shifts [15]. Content-defined chunking (CDC) algorithms [15–17] instead use the file's contents to define chunk boundaries, effectively handling the byte-shifting problem. Numerous data chunking algorithms are in use today and can be broadly divided into hash-based [15, 17–19] and hashless algorithms [16, 20, 21]. Hash-based algorithms use hash functions to derive chunk boundaries while hashless algorithms treat each byte as a value and define chunk boundaries using conditions such as local minima or maxima. Note that in both cases, a collision-resistant hash algorithm [22] is later used to hash each chunk for *fingerprinting*.

The chunking throughput of CDC algorithms is limited by their need to scan each file entirely to determine chunk boundaries. They are also designed to target datasets that benefit from smaller chunks of size 512 B – 4 KB. However, such small chunks increase metadata overhead [8] and impact system throughput due to the random access and frequent transfer of small chunks. Consequently, production systems favor storing fewer and larger chunks for datasets such as VM backups, pitting them at odds against CDC algorithms.

We present SeqCDC, a CDC algorithm geared towards larger chunk sizes. SeqCDC uses two optimizations to improve throughput for large chunks: *lightweight boundary judgment* and *content-based data skipping*. Lightweight boundary judgment reduces the overhead involved in determining the location of chunk boundaries within the file. Content-based data skipping selectively skips scanning regions of the file. Our evaluation compares SeqCDC to five state-of-the-art chunking algorithms using a variety of datasets (§6). We show that SeqCDC improves chunking throughput by 1.5×–3.1× compared to the state-of-the-art CDC algorithms while achieving comparable space savings. We have made our code public by integrating it with DedupBench[1] [23].

Middleware '24, 2024, Hong Kong, China

ACM ISBN 979-8-4007-0623-3/24/12
https://doi.org/10.1145/3652892.3700766

[1] https://github.com/UWASL/dedup-bench

2 BACKGROUND AND MOTIVATION

Data deduplication [10, 15] is used by cloud-storage providers to detect duplicate data, allowing them to eliminate the costs associated with its storage and transmission. Data deduplication consists of the following steps [8]:

- *File Chunking:* Splitting a file into chunks using a data chunking algorithm is one of the primary steps in data deduplication. Deduplicating these chunks provides higher space savings benefits than file-level deduplication [15].
- *Chunk Hashing:* Each data chunk is hashed using a collision-resistant hashing algorithm [22, 24] to obtain a fingerprint.
- *Fingerprint Comparison:* The fingerprint is compared against a database of previously observed fingerprints. A duplicate fingerprint, i.e. one that has been observed before, indicates an underlying duplicate chunk, which can be eliminated.
- *Data Storage:* Non-duplicate data chunks are saved on the storage medium and their fingerprints are added to the fingerprint database.

Chunking is the most critical part of this pipeline, it occurs on the critical path during data uploads and directly impacts the overall space savings and throughput associated with deduplication systems. Space savings represent the total space conserved by using deduplication.

$$Space\ savings = \frac{Original\ Size - Deduplicated\ Size}{Original\ Size} \times 100 \quad (1)$$

The size of the fingerprint database is tied to the average chunk size. Smaller average sizes lead to more chunks and more fingerprints, thus increasing the size of the database and associated fingerprint comparison overheads. To minimize this overhead, deduplication systems in production tend to favor larger chunk sizes.

2.1 Content-Defined Chunking (CDC) Algorithms

Numerous data chunking algorithms [15–18, 20, 21] have been proposed for data deduplication. Content Defined Chunking (CDC) algorithms slide a fixed-size window over the data within the source file. When the window's data meets pre-specified conditions, they insert a *chunk boundary* at the end of the window. By repeating this across the entire file, they divide it into data chunks. Each CDC algorithm has parameters which can be tuned to change the average size of generated chunks.

CDC algorithms are classified into hash-based and hashless algorithms. Hash-based chunking algorithms, such as Rabin's chunking [15] insert chunk boundaries only when the hash value of the window's data matches a pre-specified mask. The hashing algorithms used here are typically not collision resistant. For instance, within Rabin's Chunking (RC) [15], chunk boundaries are inserted when the lower order 13 bits of the hash value are zero. While Rabin's chunking achieves high deduplication ratios, it is very slow. TTTD [18] uses Rabin's hashing but achieves improved space savings by simultaneously checking for two hash value conditions. The secondary condition is always computed but only used if a boundary is not found beyond a pre-specified size with the primary condition.

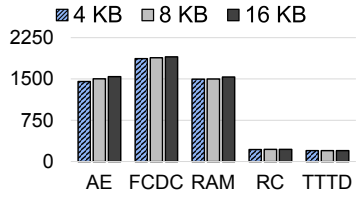


Figure 1: Chunking throughput (MB/s) on randomized data

The Gear hashing function [19] has a lower overhead than Rabin's hashing. FastCDC (FCDC) [17] uses Gear hashing and implements two optimizations to improve chunking throughput: skipping the scanning of data up to the minimum chunk size at the beginning of each chunk, and normalizing chunk sizes i.e. dynamically relaxing the boundary condition to ensure that generated chunks are close to the expected average chunk size.

On the other hand, hashless algorithms such as Asymmetric Extremum (AE) [16] and Rapid Asymmetric Maximum (RAM) [20] also slide fixed-size windows over the source data. AE attempts to identify a window such that the starting byte's value is greater than all the bytes before it and not less than the other bytes within the window. When such a window is found, AE inserts a chunk boundary at the end of the window. AE is $4-5\times$ faster than Rabin's chunking. Rapid Asymmetric Maximum [20] inserts chunk boundaries when the byte immediately outside the window has a value greater than the maximum valued byte within the window. These algorithms avoid hashing when determining chunk boundaries, reducing the computational overhead and achieving high chunking throughput [23, 25].

2.2 Chunking Throughput Analysis

We compare the chunking throughput of the state-of-the-art CDC algorithms with three average chunk sizes: 4 KB, 8 KB, and 16 KB. Figure 1 shows the throughput achieved by AE [16], FastCDC [17], RAM [20], Rabin's Chunking [15] and TTTD [18] when chunking a 1GB file containing random data. The details of our testbed are in Section 6.

We note that the throughputs of all these algorithms do not scale with chunk size, as they are primarily designed to target smaller chunk sizes. However, deduplication systems in production favor larger chunk sizes on datasets such as VM backups, to minimize fingerprint comparison overheads (§2). As a result of these CDC algorithms, deduplication systems suffer from poor chunking speeds.

3 SEQCDC'S DESIGN

SeqCDC is designed to insert chunk boundaries when fixed-length sequences of monotonically increasing/decreasing bytes are detected. SeqCDC can operate in either `Increasing` mode i.e. targeting increasing order sequences or `Decreasing` mode. Note that these two modes are exclusive of each other. Figure 2 shows an example of SeqCDC's operation. §4 discusses how SeqCDC is resistant to byte shifting.

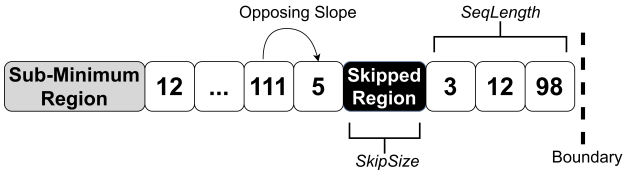


Figure 2: An example of a chunk generated by SeqCDC

SeqCDC utilizes three parameters: *SeqLength*, *SkipTrigger* and *SkipSize*, each described in detail in the following subsections. SeqCDC includes three optimizations we discuss in detail: lightweight boundary detection, ignoring data at the beginning of a chunk, and content-based data skipping within each chunk.

3.1 Lightweight Boundary Detection

To avoid complex hashing operations, SeqCDC treats each byte within the data stream as an independent value similar to existing hashless CDC algorithms. However, to improve chunking throughput even further, SeqCDC reduces the overheads associated with boundary detection by avoiding minima/maxima searches.

Instead, SeqCDC looks for *fixed-length monotonically increasing/decreasing* sequences of bytes and inserts chunk boundaries whenever such a sequence is detected. The sequence must have a length of *SeqLength* to be considered a boundary candidate. Once such a sequence is found, a boundary is inserted at its end.

Modes of operation. SeqCDC can be used in `Increasing` or `Decreasing` mode. Both these modes are exclusive of each other. While in `Increasing` mode, SeqCDC targets monotonically increasing sequences. On the other hand, it targets monotonically decreasing sequences in `Decreasing` mode. Depending upon dataset characteristics, one mode may be more effective than the other.

Figure 2 shows an example of SeqCDC operating in `Increasing` mode with a *SeqLength* of 3. A chunk boundary is inserted after the byte with value 98, as it forms an increasing sequence with the bytes preceding it.

3.2 Ignoring Sub-minimum Regions

SeqCDC utilizes the concept of ignoring data at the beginning of each chunk introduced within previous literature [17, 26] to increase chunking throughput. SeqCDC skips scanning data equal to the (*minimum_chunk_size* - *SeqLength*) at the beginning of each chunk (`"Sub Minimum Region"` in Figure 2).

Increasing the minimum chunk size allows SeqCDC to skip over larger amounts of data at the beginning of each chunk, increasing chunking throughput. However, when performed excessively, this may negatively impact space savings on some datasets. The minimum chunk size for SeqCDC is 25-50% the average chunk size, similar to existing CDC algorithms [17, 26].

3.3 Content-based Data Skipping

SeqCDC additionally improves chunking throughput by skipping scanning certain data regions when looking for chunk boundaries (`"Skipped Region"` in Figure 2). Randomly skipping data regions can negatively affect space savings. To avoid this, SeqCDC adopts a novel content-based data skipping mechanism i.e. data regions are skipped over only when skip conditions are met.

SeqCDC skips scanning data within *volatile regions* i.e. data regions with byte sequences in an order opposing the target sequence. For instance, when in `Increasing` mode, regions with decreasing order sequences are considered volatile. When *SkipTrigger* pairs of bytes in opposing order are detected, SeqCDC decides that the current region is volatile and skips scanning the next few bytes in the hope of landing in a favorable region. For instance, in Figure 2, the skip condition is triggered after the byte with a value of 5, causing the next *SkipSize* bytes to be ignored. The *SkipSize* is kept small (256-512 bytes), in order to avoid skipping over large sections of data. After a skip is triggered, SeqCDC resets its counters and resumes scanning for boundaries.

Larger *SkipSizes* improve chunking throughput. While larger *SkipSizes* are feasible for larger chunks, they may result in a disproportionately high amount of data skipped within smaller chunks, negatively affecting space savings. SeqCDC overcomes this by adjusting the *SkipSize* from $256-512$ bytes, depending on the expected average chunk size.

Data skipping can potentially impact byte shifting resistance. SeqCDC therefore trades off a small amount of byte-shifting resistance to achieve superior chunking throughput. §4 discusses this trade off in greater detail. Additionally, in our evaluation (§6.1), we show that this minimally impacts deduplication space savings in real datasets.

4 HANDLING BYTE SHIFTING

Byte shifting can occur within any region of the input data stream, causing certain chunk boundaries to change. Byte shifting can span one or more bytes and take the form of insertions or deletions. In general, sub-minimum and content-defined skipped regions affect SeqCDC's byte-shifting resistance. Figure 3 shows three chunks with four boundaries B_1 - B_4. Each chunk has a corresponding sub-minimum region (M_1 - M_3) at the beginning of the chunk. The figure also shows two regions V_1 and V_2 skipped via SeqCDC's content-defined skipping i.e., using *SkipTrigger*.

Skipped region impact: When byte-shifts occur, boundaries may be moved in and out of skipped regions (both sub-minimum and content-defined). For instance, consider S_1 in Figure 3 which occurs in the sub-minimum region M_1. If byte-shift S_1 causes a boundary previously hidden within M_1 to be pushed outside, a new chunk may be created, thus splitting Chunk 1. This may in turn lead B_2 to be pushed into M_2, affecting subsequent chunks as well. Similarly, a deletion may cause B_2 to be hidden within M_1, leading to chunk mergers. Additionally, boundaries may be moved in and out of regions previously skipped with *SkipTrigger* and *SkipSize* by shifts such as S_2, causing chunk splits and mergers.

While this behavior can theoretically impact a large number of chunks, it only impacts a limited number of chunks within real datasets. This is why many CDC algorithms such as FastCDC [17] and TTTD [26] use sub-minimum skips in production. SeqCDC also minimizes the impact caused by content-defined skipping by keeping the *SkipSize* between 256-512 bytes. Thus, despite data skipping, SeqCDC achieves space savings values competitive with those of other CDC algorithms as shown in §6.1. Finally, we note

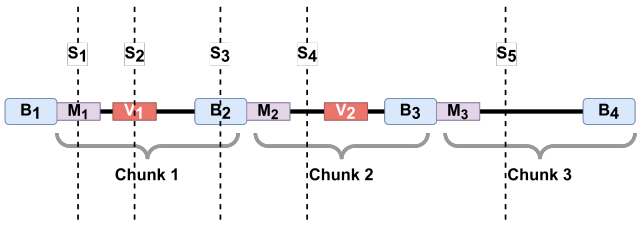


Figure 3: Handling byte-shifts with SeqCDC

	SeqLength	*SkipTrigger*	*SkipSize*
4 KB	5	55	256
8 KB	5	50	256
16 KB	5	50	512

Table 1: SeqCDC parameter values

that all CDC algorithms have pathological data patterns i.e. data which can be engineered to ensure that they are ineffective.

In the rest of this section, we focus on the more common kind of byte-shifting i.e. those that do not result in new boundaries being uncovered or hidden, drastically affecting chunks.

Within the boundary sequence. The probability of such byte shifting occurring is rare in real datasets, as *SeqLength* typically ranges from 3 to 7 bytes. If byte shifting occurs within a sequence, the boundary may no longer exist. Thus, scanning will continue until the next sequence is detected. In the figure, S_3 may cause B_2 to no longer exist. Thus, the next boundary will be after B_3, causing Chunks 1 and 2 to be merged while subsequent chunks are unaffected.

Between sequences and outside skipped regions. Byte-shifts such as S_4 are the ones most commonly seen in real datasets. They occur outside of skipped regions and do not drastically change the chunk structure. If S_4 does not create a new boundary sequence, the existing boundary sequence B_3 shifts. Thus, Chunk 2 changes while subsequent chunks are unaffected. On the other hand, if S_4 does create a new boundary sequence, a boundary is inserted after it, changing Chunk 2. This may result in B_3 being moved into the sub-minimum region of the next chunk and being skipped. Thus, the next boundary will be inserted after B_4, causing Chunk 3 to change. Thus, Chunks 2 and 3 are affected while others are unaffected.

Maximum chunk size. If a shift S_5 causes the maximum chunk size to be reached, a boundary is inserted at the maximum chunk size. This will cause the chunk to split into multiple chunks. For instance, if a boundary is inserted after S_5 in Figure 3, Chunk 3 will be split into two if no subsequent boundary sequences are hidden.

5 IMPLEMENTATION

We implement SeqCDC using ~250 lines of C++ code. We integrated SeqCDC with our previous work DedupBench [23]. We optimized the computation of *SeqLength* and *SkipTrigger* using the `std::signbit` function to reduce the number of branch conditions. SeqCDC is compatible with all file and data formats.

Dataset	Size	Information	XC
DEB	40 GB	65 Debian VM Images obtained from the VMware Marketplace [29]	28.1%
DEV	230 GB	100 backups of a Rust [30] nightly build server	90.9%
LNX	65 GB	160 Linux kernel distributions in TAR format [31]	34.8%
RDS	122 GB	100 Redis [32] snapshots with `redis-benchmark` runs	33.7%
TPCC	106 GB	25 snapshots of a MySQL [33] VM running TPC-C [34].	54.6%

Table 2: Dataset information. Note that *XC* is fixed-size chunking at 4 KB.

Obtaining parameter values. SeqCDC has three configurable parameters apart from its mode: *SeqLength*, *SkipTrigger* and *SkipSize*. To obtain the parameter value combination needed to generate a given average chunk size, we first used Monte-Carlo simulations [27] on randomized data streams. For example, to identify the parameter values for an average of 4 KB, we conducted simulations on randomized data to identify candidate combinations resulting in an average chunk size of 4 KB. Following this, we experimented on one of our datasets (`DEB` from §6) to pick the best performing combination, which we use across all the datasets in our evaluation. The final parameter values used within our evaluation (§6) to obtain average chunk sizes of 4-16 KB are shown in Table 1.

The results outlined in §6 show that SeqCDC outperforms other CDC algorithms on all datasets with these chosen parameters. Thus, it is not necessary to run an extensive parameter search per dataset, instead obtaining suitable parameters using the method described above. However, to tune SeqCDC to a specific dataset, better parameter combinations may be obtained using such a search.

6 EVALUATION

In this section, we evaluate SeqCDC's space savings and chunking throughput and compare it to the state-of-the-art CDC algorithms.

Testbed. We use an AMD EPYC Rome machine from CloudLab [28] for our evaluation. The machine consists of a 16-core AMD 7302P CPU with hyperthreading, 128 GB RAM and a 25 GBps Mellanox NIC. All our results are the average of 5 runs with a standard deviation of less than 5%.

Alternatives. We compare the following CDC algorithms:

- *AE*: The Asymmetric Extremum [16] algorithm.
- *FCDC*: FastCDC [17] with a normalization level of 2.
- *RC*: Rabin's chunking algorithm [15].
- *RAM*: Rapid Asymmetric Maximum [20].
- *SeqCDC*: We only report the results for SeqCDC in `Increasing` mode. The results for `Decreasing` mode are similar.
- *TTTD*: Two-Threshold Two-Divisor Algorithm [18].

We use minimum and maximum chunk sizes of $\frac{1}{2}\times$ and $2\times$ the expected average chunk size, in line with previous studies [17, 26]. The only exception is that for a small average chunk size of 4 KB, we use a minimum size of 1 KB.

Datasets. Table 2 shows the datasets used within our evaluation as well as the space savings achieved by using fixed-size chunking

(XC) on them with an average size of 4KB. By comparing the space savings achieved by XC on these datasets to those achieved by the CDC algorithms (Table 3), we note that the datasets possess varying degrees of byte-shifting. For instance, XC achieves a space savings of only 54.6% on TPCC while CDC algorithms achieve 86-87%. Finally, we note that the datasets represent diverse workloads such as database backups, VMs, and Linux kernel code.

6.1 Space Savings

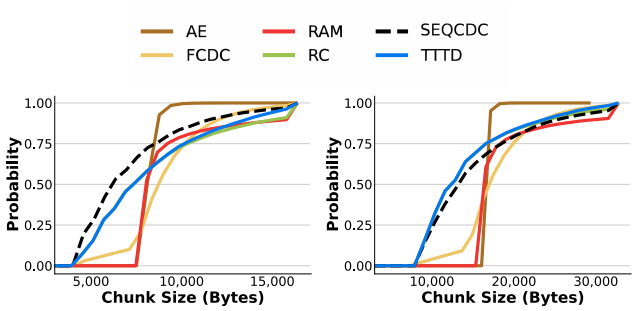
Table 3 shows the space savings achieved by all the alternatives across datasets. We note that all CDC algorithms achieve superior space savings when compared to fixed-size chunking (XC from Table 2), showing that they effectively handle byte-shifting.

The space savings achieved by all algorithms decreases with increasing chunk size. The space savings degradation between 4 KB and 16 KB average chunk sizes on the DEV, RDS, and TPCC datasets is ~6% across all CDC algorithms. However, as the total number of chunks at 16 KB is far lower than that at 4 KB, the size of the fingerprint database and fingerprinting overheads are significantly smaller with 16 KB chunks. Similarly, the best chunk size configuration for the DEB dataset is 8 KB. This demonstrates why deduplication systems favor larger chunk sizes on some datasets.

SeqCDC achieves similar space savings to all the other CDC algorithms on these datasets across chunk sizes. The best algorithm for space savings varies depending on the combination of dataset and target chunk size. For instance, at 4 KB, TTTD and RAM achieve the best space savings on DEV and RDS respectively. Similarly, at 8 KB, SeqCDC and TTTD achieve the best space savings on DEB and TPCC respectively. *On all datasets except* LNX, *SeqCDC either is the best or achieves space savings within* 4% *of the best performer.*

On the other hand, the LNX dataset presents a case favoring smaller chunk sizes. The space savings degradation caused by moving from average chunk sizes of 4 KB to 16 KB is ~30% across algorithms, which far outweighs any gains within fingerprint indexing. Nevertheless, SeqCDC achieves space savings within 6% of the best performer on this dataset.

Chunk size distribution. Figure 4 shows a CDF of chunk sizes from all algorithms on the TPCC dataset at average sizes of 8 KB and 16 KB. Rabin's Chunking (RC) and TTTD exhibit similar distributions since TTTD only differs from RC by the use of a backup divisor. AE and RAM exhibit different distributions when compared



(a) 8 KB (b) 16 KB

Figure 4: CDF of chunk sizes on TPCC

Dataset	CDC	4KB	8KB	16KB
DEB	AE	41.99%	33.23%	21.47%
	FCDC	43.83%	36.97%	27.77%
	RC	44.38%	36.16%	27.22%
	RAM	42.98%	34.63%	22.61%
	TTTD	**45.06%**	37.13%	**27.94%**
	SeqCDC	42.77%	**37.76%**	27.62%
DEV	AE	98.00%	97.72%	97.21%
	FCDC	98.17%	98.06%	97.90%
	RC	98.21%	98.09%	97.97%
	RAM	98.05%	97.79%	97.31%
	TTTD	**98.22%**	**98.10%**	**97.98%**
	SeqCDC	98.13%	98.03%	97.82%
LNX	AE	59.41%	45.33%	31.67%
	FCDC	59.16%	44.35%	33.64%
	RC	67.02%	49.28%	35.40%
	RAM	57.94%	42.90%	29.12%
	TTTD	**68.46%**	**51.06%**	**36.92%**
	SeqCDC	63.13%	49.46%	33.26%
RDS	AE	94.66%	92.86%	91.04%
	FCDC	93.82%	92.04%	90.15%
	RC	94.31%	92.32%	90.57%
	RAM	**95.67%**	**94.09%**	**92.03%**
	TTTD	95.2%	93.30%	91.55%
	SeqCDC	94.86%	92.54%	88.78%
TPCC	AE	86.58%	84.96%	81.58%
	FCDC	87.18%	86.74%	86.17%
	RC	87.24%	86.80%	86.37%
	RAM	86.71%	85.21%	81.67%
	TTTD	**87.29%**	**86.84%**	**86.40%**
	SeqCDC	87.04%	86.68%	85.83%

Table 3: Space savings of CDC techniques

to hash-based algorithms. SeqCDC exhibits a chunk size distribution similar to TTTD and RC. We observed similar results across all our datasets and chunk sizes.

6.2 Chunking Throughput

Figure 5 shows the chunking throughput of all the alternatives on all datasets. The chunking throughput for each algorithm only marginally varies between datasets. When examining chunking throughputs across average chunk sizes instead, we note that the throughput only minimally scales with size for all algorithms other than SeqCDC, similar to our analysis in §2.2.

Among the hash-based algorithms, Rabin's chunking [15] and TTTD [18] are the slowest, only achieving 200 – 220 MB/s. TTTD is slightly slower than Rabin's chunking due to its extra comparison condition (§2.1). The poor chunking throughput of both algorithms is due to the high computational cost of Rabin's hashing, as pointed out in previous literature [17]. FastCDC fares significantly better and achieves 2000 – 2700 MB/s. FastCDC's throughput increase

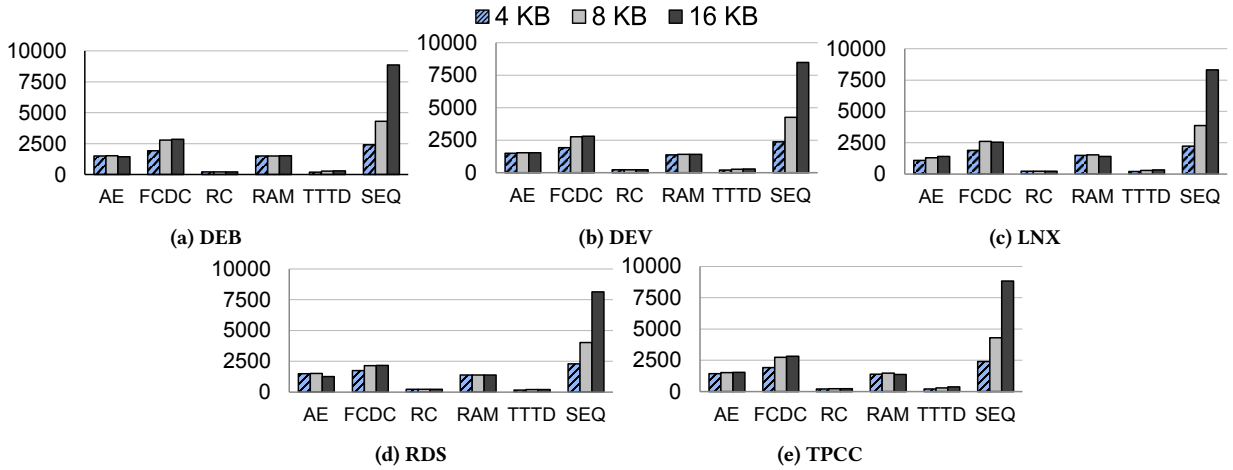


Figure 5: Chunking throughput (MB/s) comparison across datasets. Note that *SEQ* is SeqCDC.

from 4 KB to 8 KB is due to the increased ratio of minimum chunk size to average chunk size (25% to 50%).

Hashless algorithms such as AE [16] and RAM [20] achieve higher chunking throughput than Rabin's chunking and TTTD. They both achieve throughputs of 1450 − 1500 MB/s across all datasets. RAM is computationally less expensive than AE [20], leading it to perform better in certain cases (such as LNX at 4 KB).

SeqCDC consistently achieves higher chunking throughput than all other CDC algorithms at average chunk sizes of 8 KB and 16 KB. At a chunk size of 8 KB, it achieves 4000 − 4250 MB/s, 1.5× and 2.8× better than FastCDC and AE/RAM respectively. At a chunk size of 16 KB, it achieves a chunking throughput of ~8000-8800 MB/s, 3.1× and 5.8× better than FastCDC and AE/RAM respectively. SeqCDC's increase in throughput from 8 KB to 16 KB is primarily due to the increasing *SkipSize* from 256 bytes to 512 bytes.

At a chunk size of 4 KB, SeqCDC's *SkipTrigger* is increased by 10% to constrain the amount of data skipped. SeqCDC still achieves 2400 MB/s, 0.2× and 0.6× faster than FastCDC and AE/RAM respectively. Thus, *SeqCDC is faster than all other CDC algorithms by* 1.5 × −3.1× *at larger chunk sizes and* 0.2× *at smaller chunk sizes.*

Throughput breakdown. Figure 6 shows the impact of each optimization (§3) on SeqCDC's throughput. Lightweight boundary judgement *(BDRY)* results in a throughput of ~2300 MB/s. When sub-minimum ignore is enabled *(MIN)*, this throughput increases by 1.55×–2×. Finally, adding content-based skipping *(ALL)* allows SeqCDC to achieve its peak throughput of 8000–8500 MB/s seen previously in Figure 5.

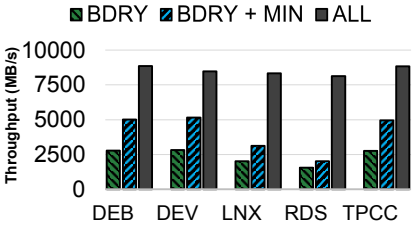


Figure 6: SeqCDC throughput breakdown - 16 KB

7 RELATED WORK

Accelerating Deduplication. Numerous efforts have been made to accelerate the other phases involved in data deduplication. StoreGPU [35] accelerates hashing operations using GPUs to speed up fingerprint comparison. Silo uses locality-based optimizations to improve fingerprint comparison [36]. These efforts are orthogonal to ours as we focus on the file chunking phase within deduplication.

Other Chunking Optimizations. RapidCDC [37] and QuickCDC [38] use locality-based optimizations to speed up chunking for duplicate chunks. MUCH [39] parallelizes chunking using multiple threads. SeqCDC is compatible with any of these techniques as they all rely on implementing optimizations on top of existing CDC techniques. MII [40] uses a sequence-based approach to chunk data but their approach results in inflexible chunk sizes and low throughput.

8 CONCLUSION

Deduplication systems in production employ larger chunk sizes due to reduced fingerprinting overheads. However, state-of-the-art CDC algorithms are designed to target smaller average chunk sizes, suffering from poor chunking throughput at larger sizes.

We present SeqCDC, a CDC algorithm that achieves higher chunking speeds than the state-of-the-art. SeqCDC leverages content-based data skipping and hashless lightweight boundary judgement to improve chunking throughput by 1.5 × −3.1× while achieving similar deduplication space savings.

9 ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Tobias Distler, for their feedback. This research was supported by an NSERC Discovery Grant and an Acronis Research Grant. Sreeharsha is supported by an Ontario Graduate Scholarship.

REFERENCES

[1] Arne Holst. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025. *Statista, June*, 2021.

[2] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, oct 2003.

[3] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[4] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[5] Garth A Gibson and Rodney Van Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, 2000.

[6] Debra A Lelewer and Daniel S Hirschberg. Data compression. *ACM Computing Surveys (CSUR)*, 19(3):261–296, 1987.

[7] David Salomon. *Data compression*. Springer, 2002.

[8] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[9] Nagapramod Mandagere, Pin Zhou, Mark A Smith, and Sandeep Uttamchandani. Demystifying data deduplication. In *The ACM/IFIP/USENIX Middleware'08 Conference Companion*, pages 12–17, 2008.

[10] Dutch T Meyer and William J Bolosky. A study of practical deduplication. *ACM Transactions on Storage (ToS)*, 7(4):1–20, 2012.

[11] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, San Jose, CA, February 2012. USENIX Association.

[12] Larry Coyne, Sandra Moulton, and Carlos Alvarez. *IBM System Storage N Series Data Compression and Deduplication: Data ONTAP 8.1 Operating in 7-mode*. IBM Redbooks, 2012.

[13] Sean Quinlan and Sean Dorward. Venti: A New Approach to Archival Data Storage. In *Conference on File and Storage Technologies (FAST 02)*, Monterey, CA, January 2002. USENIX Association.

[14] Qinlu He, Zhanhuai Li, and Xiao Zhang. Data deduplication techniques. In *2010 international conference on future information technology and management engineering*, volume 1, pages 430–433. IEEE, 2010.

[15] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *The eighteenth ACM symposium on Operating systems principles*, pages 174–187, 2001.

[16] Yucheng Zhang, Hong Jiang, Dan Feng, Wen Xia, Min Fu, Fangting Huang, and Yukun Zhou. AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1337–1345. IEEE, 2015.

[17] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 101–114, 2016.

[18] Kave Eshghi and Hsiu Khuern Tang. A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR*, 30(2005), 2005.

[19] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation*, 79:258–272, 2014. Special Issue: Performance 2014.

[20] Ryan NS Widodo, Hyotaek Lim, and Mohammed Atiquzzaman. A new content-defined chunking algorithm for data deduplication in cloud storage. *Future Generation Computer Systems*, 71:145–156, 2017.

[21] Nikolaj Bjørner, Andreas Blass, and Yuri Gurevich. Content-dependent chunking for differential compression, the local maximum approach. *Journal of Computer and System Sciences*, 76(3-4):154–203, 2010.

[22] Andrew W Appel. Verification of a cryptographic primitive: Sha-256. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(2):1–31, 2015.

[23] Alan Liu, Abdelrahman Baba, Sreeharsha Udayashankar, and Samer Al-Kiswany. Dedupbench: A Benchmarking Tool for Data Chunking Techniques. In *2023 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 469–474, 2023.

[24] Ronald Rivest. The MD5 message-digest algorithm. Technical report, 1992.

[25] Mu'men Al Jarah, Sreeharsha Udayashankar, Abdelrahman Baba, and Samer Al-Kiswany. The impact of low-entropy on chunking techniques for data deduplication. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, pages 134–140, 2024.

[26] Teng-Sheng Moh and BingChun Chang. A running time improvement for the two thresholds two divisors algorithm. In *The 48th Annual Southeast Regional Conference*, pages 1–6, 2010.

[27] Christopher Z Mooney. *Monte carlo simulation*. Number 116. Sage, 1997.

[28] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[29] VMWare. VMWare marketplace. https://marketplace.cloud.vmware.com/services, 2023.

[30] Rust. GitHub - rust-lang/rust: Empowering everyone to build reliable and efficient software. https://github.com/rust-lang/rust, 2023.

[31] Linux. The Linux Kernel Archives. https://www.kernel.org/, 2023.

[32] Redis. Redis. https://redis.io/, 2023.

[33] MySQL. MySQL. https://www.mysql.com/, 2023.

[34] Transaction Processing Council. TPC-C Overview. https://www.tpc.org/tpcc/detail5.asp, 2023.

[35] Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, George Yuan, and Matei Ripeanu. StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems. In *The 17th International Symposium on High Performance Distributed Computing*, HPDC '08, page 165–174, New York, NY, USA, 2008. Association for Computing Machinery.

[36] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. Similarity and Locality Based Indexing for High Performance Data Deduplication. *IEEE Transactions on Computers*, 64(4):1162–1176, 2015.

[37] Fan Ni and Song Jiang. RapidCDC: Leveraging Duplicate Locality to Accelerate Chunking in CDC-Based deduplication systems. In *The ACM Symposium on Cloud Computing*, SoCC '19, page 220–232, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Zhen Xu and Wenbo 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)*, pages 288–299, 2021.

[39] Youjip Won, Kyeongyeol Lim, and Jaehong Min. MUCH: Multithreaded Content-Based File Chunking. *IEEE Transactions on Computers*, 64(5):1375–1388, 2015.

[40] Changjian Zhang, Deyu Qi, Zhe Cai, Wenhao Huang, Xinyang Wang, Wenlin Li, and Jing Guo. Mii: A novel content defined chunking algorithm for finding incremental data in data synchronization. *IEEE Access*, 7:86932–86945, 2019.