Sorted Deduplication: How to Process Thousands of Backup Streams

Jürgen Kaiser, Tim Süß, Lars Nagel, André Brinkmann
Johannes Gutenberg University
Mainz, Germany
{*kaiserj, suesst, nagell, brinkman*}*@uni-mainz.de*

Abstract

The requirements of deduplication systems have changed in the last years. Early deduplication systems had to process dozens to hundreds of backup streams at the same time while today they are able to process hundreds to thousands of them. Traditional approaches rely on stream-locality, which supports parallelism, but which easily leads to many non-contiguous disk accesses, as each stream competes with all other streams for the available resources.

This paper presents a new exact deduplication approach designed for processing thousands of backup streams at the same time on the same fingerprint index. The underlying approach destroys the traditionally exploited temporal chunk locality and creates a new one by sorting fingerprints. The sorting leads to perfectly sequential disk access patterns on the backup servers, while only slightly increasing the load on the clients. In our experiments, the new approach generates up to 113 times less I/Os than the exact Data Domain deduplication file system and up to 12 times less I/Os than the approximate Sparse Indexing, while consuming less memory at the same time.

I. Introduction

The huge increase in data generating and processing value chains in industry and academia has dramatically increased the pressure on backup environments. Data has become a valuable asset itself and it is therefore important to protect it using highly reliable backup systems. Most backup systems have been based on tape environments until the late 2000s and tape is still one of the most cost-effective ways to store data. Disk-based backup environments profit from deduplication techniques which often significantly reduce the amount of data stored. These techniques makes disk an economically applicable backup technology.

Most deduplication systems split the backup data set into small chunks and identify redundancies within the data by comparing the chunks' fingerprints [1]–[5]. A new chunk of data is only stored if its fingerprint is not already contained in the so-called *chunk index*. For a later data restore, the systems create *recipes* which contain all information that is necessary to rebuild the stored data.

Look-ups in the chunk index have been the main performance bottleneck in deduplication systems as the complete index is typically too large to be held in main memory and index accesses generate random I/O. Therefore, deduplication system designers typically use at least one of the following techniques:

- They reduce the index size so that it fits in main memory and trade this for a reduced duplicate detection rate.
- They exploit chunk locality, i.e. *the tendency for chunks in backup data streams to reoccur together* [2] and prefetch chunk fingerprints block-wise from an internal data structure that catches this locality. For a single backup stream, prefetching can generate a near-sequential disk access [6].

Yet, even if every backup stream has this locality property, the disk-friendly access scheme would degrade to a random one when more backups are processed in parallel. In addition, the streams compete for the available memory space, as each additional stream reduces the stream-local cache sizes so that the number of I/O accesses increases. As a result the write performance decreases, especially when the system load is high. High system loads occur naturally in companies as users *prefer default scheduling windows during weekdays, resulting in nightly bursts of activity* [7]. Deduplication systems for online backups face the same challenges. Here, users might backup their data more or less evenly throughout the day, but the number of users can easily reach 10-100K and result in 1-10K streams at peak times. Hence, there is a need for systems that efficiently process thousands of streams without heavy hardware requirements.

In this work, we present a deduplication approach which is tailored to handle many streams and to avoid the memory problem. It achieves this goal by processing all streams in sorted order so that all streams access the same index region at the same time. This creates and exploits a high index locality. The number of I/Os of this approach only depends on the volume of the globally unique data, but is nearly independent from the number of concurrent data streams.

We compare the resulting system with the well-established approaches of Zhu et al. [1] and Lillibridge et al. [2]. In experiments, our system generates up to 113 times less I/Os than the one of Zhu et al. and up to 12 times less than the one of Lillibridge et al. At the same time, it still performs exact deduplication and consumes less main memory than the other systems.

II. Sorted Chunk Indexing

All deduplication systems try to avoid the chunk-lookup disk bottleneck. The main approach to this varies among the systems, but most systems try to exploit the chunk locality. This locality is used to prefetch and cache chunk identification information to save I/O operations.

However, this locality is stream-local and a system that processes thousands of streams should exploit a global locality. We create this by maintaining a sorted chunk index and process each stream coordinated in sorted order. This creates a perfect chunk index locality, reduces overall disk I/Os, and creates a sequential disk access pattern.

In the following, we describe this *sorted deduplication* and a sample system, *Sorted Chunk Indexing* (SCI) in detail.

A. Sorted Deduplication

Traditional deduplication systems can generate a non-sequential disk access pattern during prefetching for one of the following three reasons:

- **Chunk Sharing:** Some chunks appear multiple times across the same backup stream [6]. This causes repetitive prefetches of the fingerprints of the chunk and its neighbors. In addition, chunk sharing also can occur among different streams.
- **Independence:** Even if the chunk sharing among streams is low, the system generates random I/O since all streams are processed in parallel and all stream-local prefetches generate a global random access.
- **Aging:** Backups change over time. For some systems, this reduces the locality of internal data structures and can increases the number of prefetches.

A sorted deduplication can eliminate the random accesses as follows: For each sorted stream, the system iterates over the sorted chunk index and identifies the chunks in the stream on the fly. This eliminates any random I/O that is caused by stream-local chunk sharing because the sorted order, by definition, enforces that no chunk is revisited. In addition, there is no random I/O because of aging since changes in the backup stream change the set of chunks but not their order of appearance.

However, an independent processing of backup streams does not eliminate the random I/Os that are cause by inter-stream sharing and independence. For this, the system additionally must enforce the same processing speed for each stream so that each stream accesses the same chunk index position at the same time. This guarantees that the same I/O access can serve arbitrary many streams. The resulting I/O pattern becomes a completely sequential run.

B. System

As a proof of concept and for testing the sorted deduplication approach, we have developed a prototype implementation called *Sorted Chunk Indexing* (SCI). In SCI, clients perform the chunking and fingerprinting while the server identifies all chunks.

1) Server: The server is responsible for identifying new chunks in the backup streams. For simplicity, we first assume that all streams from the clients start at the same time. Later, we relax this and show how data streams can be handled if they appear spontaneously.

The server receives chunk fingerprints in sorted order. It utilizes the order by organizing the chunk index in a two-level log-structured merge-tree (LSM tree) [8]. The LSM tree stores the data sequentially and in sorted order in its leaf nodes on disk.

This allows processing all streams concurrently in a single sequential run over the tree's leaves. For each loaded leaf, the server processes all fingerprints that fall into the interval covered by that leaf before it advances to the next one. This ensures the same processing speed of all streams on server side. Since the LSM tree only modifies leaves when it merges the memory based first level into the second, each leaf is read-only and can be accessed by each stream in parallel. To further reduce the I/O accesses, the leaves are grouped into pages of the same size. Thus, the number of pages constitutes an upper bound for the number of generated I/Os during a backup run. New pages are added when the size of the memory-based first level structure of the LSM tree reaches a predefined threshold [8]. While the server processes a page, it prefetches the next one because it is likely that it will be used for a sufficient volume of backup data. Note that the probabilities of the pages to be hit by a fingerprint are proportional to the page's fingerprint range and these probabilities are therefore not uniform. In Section III-B we further investigate this probability.

Concurrent Data Streams: New clients are not forced to wait for an index run to complete. Instead, they can exploit that they can start transmitting their fingerprints at any point in the sorted list and, therefore, hook in the current index run. For this, each client first asks the server for the last fingerprint of the page currently being handled and start the transmission with the next bigger fingerprint. In addition, the system allows the clients to send multiple streams starting from different positions to overcome different clients' processing speeds. This introduces multiple, independent runs over the pages as indicated in Figure 1. However, the number of these runs must be limited as the disk access pattern

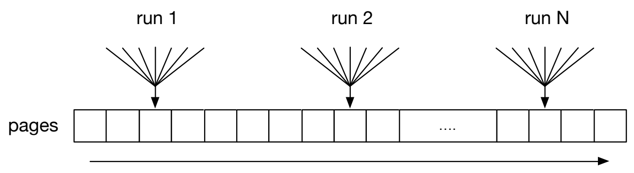


Figure 1. Multiple concurrent runs over the sorted chunk index.

becomes random for too many runs.

2) Client: Similar to DDBoost [9], the client first performs the chunking and fingerprinting of the backup data. Each client uses the same chunking and hash function method to enable the server to detect inter-stream redundancies. Next, it sorts the fingerprints and communicates with the server to detect the new fingerprints as shown in Figure 2. For this, it first sends the sorted list of fingerprints. During sending, it skips chunks that are redundant within the same backup. These chunks are directly detectable in sorted order because all duplicates of a fingerprint are arranged in one block in the list. The server replies with a list which contains the storage location (*container* ID) for each received fingerprint or a nil value if the chunk is new. The storage location is mandatory because the client is also responsible for creating the recipe information for restoring the backup data. The server cannot perform this task because it has no information about the original ordering of the chunks. Finally, the client sends the recipes and the raw data of the new chunks to the server, which then adds the storage location of new chunks to the recipes as it stores them. The raw chunks are sent in original order to preserve the chunk locality for the restore case. During restoration, the sorted order would create a random access to the containers. Note that the server fills containers only with chunks of the same stream, similar to other systems [1], [2].

The overhead for the client operations is as follows: The chunking and fingerprinting requires to read all backup data. The sorting is a standard sort algorithm requiring $\mathcal{O}(n \log n)$, where n is the number of chunks in the backup data before any deduplication. Building the messages for the server can be done in $\mathcal{O}(n)$. The number of network I/O operations depends on the number of messages the client must send. This number in turn depends on the available buffer space of the server per client. We assume that the buffer can hold $k \geq 2$ fingerprints and that the sever receives one message while processing the previous one. Hence, the client sends and receives $\mathcal{O}(\frac{2n}{k})$ messages. The next step is the update of the recipe. This requires $\mathcal{O}(n)$ operations if the client additionally maintains an index storing for each chunk pointers to its occurrences in the recipes. Additionally, the client performs $\mathcal{O}(n_{new})$ copy operations, where n_{new} is the number of new chunks.

Weak clients: Since backups usually are bigger than the available main memory, the client cannot hold all data in memory. Several trade offs are possible: First, the client can hold only the metadata in memory, i.e., the list of fingerprints. This mechanism allows the client to process the full backup in one processing round at the cost of one additional I/O round: the backup data is read once for the chunking and fingerprinting and additionally each new chunk has to be read a second time.

The client can also process the backup data window-wise such that all raw and metadata fit into memory. This prevents a second I/O on the client side, but induces several processing rounds on the server side. However, if many clients write their backup at the same time, the additional I/O costs are low because all I/Os are shared among the clients.

Weak connections: With a sufficient reliable and fast connection, the clients can send the data fast enough, i.e., all their fingerprints for the next page arrive before the server starts processing it. However, a bad network or internet connection may slow down single clients and, therefore, all clients in the same run. Hence, we allow the clients to adapt to the server's index processing speed by skipping fingerprints and sending them in a later round as shown in Figure 3. Even if clients must participate in multiple index runs, the resulting sending scheme is optimal from the clients' point of view because they can send the fingerprints at their maximum speed.

We computed the minimum transmission throughput of the clients to avoid the fingerprint skipping. This throughput mainly depends on the clients' backup size, the index processing speed, and the age of the system, i.e. the volume of unique chunks stored in the deduplication system. Figure 4 shows the minimum sending throughput of each client. For the figure we assumed an index entry size of 28 B (20 B SHA-1 fingerprint + 4 B container ID + 4 B reference counter) and that a client's backup consists of 16 GB of different chunks, each with an expected size of 8 KB. The threshold is high for younger systems since the chunk index is small and the system can perform an index run fast. For example, the clients must send their fingerprints with more than 8 MB/s if the system has stored 256 GB unique data and processes the chunk index ($\frac{256\text{GB } 28\text{B}}{8\text{KB}}$ =

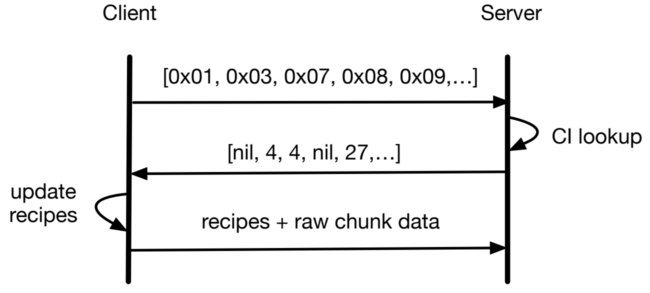


Figure 2. Communication steps between client and server.

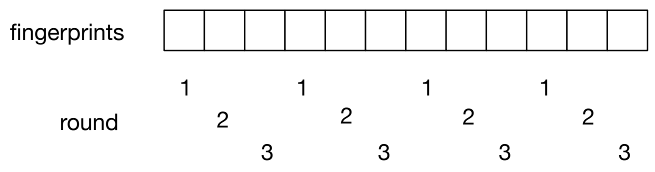


Figure 3. Sending schema of weaker clients.

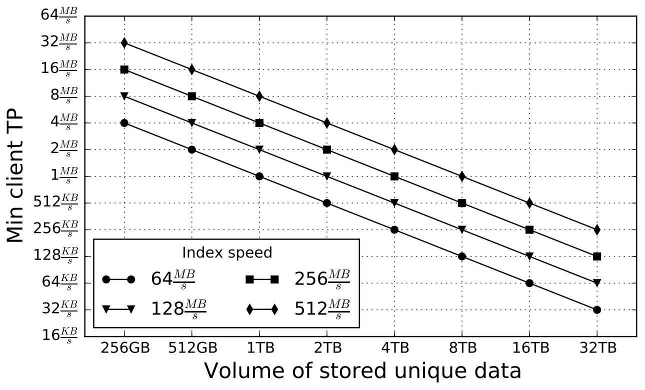


Figure 4. Minimum transmission throughput of clients.

896 MB) at the speed of a common hard disk (128 MB/s). The threshold decreases for bigger chunk indexes. For 32 TB of stored unique chunks, the clients must send at rate of 64 KB/s, which is small enough for most clients. Note that the minimum throughput also decreases for smaller backups.

C. Inter-stream Redundancy

It can happen that two or more clients send the same chunk fingerprint during the same index run. This case requires a special handling because the system must ensure fault tolerance, i.e. the chunk must be stored as long as there is at least one non-crashing client. Therefore, the server always signals each client that the chunk is new (second message in Figure 2) if there is no entry in the chunk index. This lets each client send the chunk and ensures that the chunk arrives at the server as long as there is at least one non-crashing client.

The server could simply store the respective chunks multiple times, even if this rare event relaxes the exact deduplication constraint. Nevertheless, exact deduplication can be maintained if the server keeps an in-memory index to manage in-flight chunks. When a new chunk arrives, it adds a side index entry to lock it, stores the chunk raw data, and updates the chunk index, side index, and the recipe with the storage location. The server then can just discard the chunk and update the recipe if this chunk data arrives a second time.

Each side index entry additionally contains a counter and a timestamp. The counter is set to the number of remaining in-flight versions of the chunk while the timestamp represents a timeout. In the common case, no client crashes and the server removes the entry when receiving the last incoming chunk. If a client crashes, the server removes the entry based on the timestamp during a regularly cleanup of the side index.

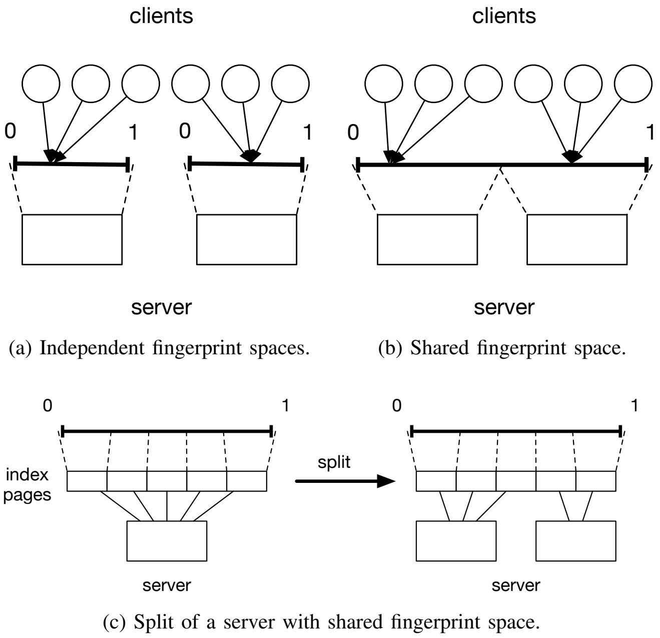
D. Restore Speed Improvements

The SCI approach presented in this paper is compatible with restore speed improvements introduced by previous work [10], [11]. Most of the approaches improve restore speed by fixing chunk fragmentation, i.e., the effect that the caught locality of the data structures holding raw chunk information (*containers*) diverges from the actual locality of the backup streams over time. While processing a backup stream, they allow the system to fill new containers with old chunks that occur at the current position in the data stream.

This technique requires two properties: First, the system must fill containers with chunks of the same backup stream. Second, the system must know the history of the backup stream, i.e., a list of container IDs sorted by the position of their according chunks in their original order. Each container ID may occur multiple times in the list as each according chunk can occur multiple times in the stream. In our system, the former property is guaranteed as mentioned above. The latter is also guaranteed because the clients send the recipes in their original order and each recipe must hold its list of fingerprints in their original order to allow the system to restore the described data. In addition, a recipe holds the container ID of each fingerprint to save a chunk index lookup in the restore case. Therefore, the server receives the container ID history while receiving the recipes from the clients.

E. Distributed Mode

The system is designed to handle thousands of streams with a single server. However, a single-server setup is limited in its scalability. To distribute the workload to multiple servers, two approaches are possible which are depicted in Figure 5. The first approach (Figure 5a) divides the set of



(a) Independent fingerprint spaces. (b) Shared fingerprint space.

(c) Split of a server with shared fingerprint space.

Figure 5. Different distribution modes.

clients into disjoint subsets and maintains a chunk index for each subset on separate servers. The mapping of clients to servers/indices is trivial and must only ensure that the same clients are mapped to the same servers for each backup.

The disadvantage of this approach is that the system performs approximate deduplication, i.e., it can happen that a chunk is stored more than once. This happens when two clients are in disjoint sets and send the same chunk fingerprint to different servers. In this case, both indices would declare the chunk as new because the servers work independently of each other. The impact of approximate deduplication depends on the data: in our test data sets, for example, less than 1% of all chunks are shared among two or more streams.

Another disadvantage is that a split of an overloaded server is complex. In this case, the system splits the set of a server's clients into two subsets C_{stay} and C_{move} and moves index entries to a new server. To split the chunk index, the system performs a full index run and checks for each entry whether the chunk was stored by any client in C_{move}. This in turn requires a new reverse index holding fingerprint$\rightarrow\{$clients having the chunk$\}$ mappings which requires additional resources.

The second approach is shown in Figure 5b, where the system divides the fingerprint space instead of dividing the clients. Here, each server is assigned a subrange of the [0,1) interval and only keeps chunk index entries in its subrange. Each server performs own index runs on its own subrange. Compared to the first approach, this system has the advantages that it performs exact deduplication and that the splitting of an overloaded server is simpler. If, for example, a server splits its range [0, 0.5) into the subranges [0, 0.25) and [0.25, 0.5), it only has to transfer the chunk index pages that belong to the range of the new server. This is a simpler and more straightforward method because each index page covers its own contiguous range in the [0,1) interval (see Figure 5c). If the system splits the old ranges at index page boundaries, it only needs to transfer complete index pages without splitting any existing ones. Each transferred page can then directly be used in the new server without modifications.

The shared space approach, however, complicates the client handling. The system can now choose between two options: The clients can either send their fingerprints as before, but are stopped at each index boundary until a new index run is started for the next range; or they sort their fingerprints subject to the split of the servers and send them to the subranges in parallel. Which of the options is better depends on the interconnect between clients and servers: A client with a slow interconnect should use the first technique, while a client with a fast interconnect can use the second one to utilize the parallel processing on the server side.

F. Recipes and Container IDs

SCI holds the container IDs as part of the recipes. This saves random accesses to the chunk index during a restore, as the recipe holds enough information to load a given chunk. However, these savings are traded for a higher complexity: Backups will be deleted over time and old containers start to store less active chunks. The deduplication system should therefore merge containers to save storage capacity.

The new container holds all old, but active chunks and also has a new ID. Without an additional mechanism, the merge invalidates several recipes as the container IDs of several chunks have changed. Therefore, all affected recipes must be found and updated, which is an expensive operation. To mitigate this effect, SCI can maintain a layer of indirection for container IDs and maintain a container index which maps each container ID to its position on disk. During a merge, SCI then updates the entries of the old containers with the position of the new one. The new index is small enough to be held in memory as, in the simplest case, a single entry consists of two 8 B entries. Therefore, 1 GB of memory can hold about 268 million entries.

Alternatively, the recipes could not contain container IDs so that the chunk index must be queried during a restore. Note that this case has only a minor impact on the communication pattern. The second communication step in Figure 2 is still mandatory as it tells the clients which raw chunk data to send in the next step. Instead of the container IDs, the server only would send a bitmap, therefore reducing the volume of the messages.

G. Limitations

The described system is designed to concurrently handle many streams. However, it is not suitable for every scenario because the index runs can cause a considerable write latency. SHA1 creates a nearly uniform fingerprint distribution, so that even small writes involving only few chunks require lookups in pages across the whole fingerprint range. An extreme case would, e.g., be a backup that consists of only two chunks whose fingerprints are mapped to the first and last index page. After the first lookup, the respective client would not be able to directly jump to the last page because the other clients would most likely have chunks in between and force this client to wait.

III. Evaluation

Our system evaluation focuses on the server side of SCI and compares SCI with DDFS and Sparse Indexing. We will first describe the evaluation method and the used data sets, then we will determine a suitable page size for the LSM tree. Afterwards, using this page size, we will investigate how many streams can be processed in parallel and at which backup size SCI starts to outperform DDFS and Sparse Indexing. Finally, we will compare the I/O access patterns of

Table I
DATA SET OVERVIEW.

	HPC_HOME	MS
#different Streams	597	140
#backup Generations	61	33
Avg. #Streams/Gen.	68	27
Avg. Chunk Size	8 KB	8 KB
Backup Volume total	8 TB	48.7 TB
Backup Volume unique	3.7 TB	7.8 TB
Avg. Backup Vol./Gen.	131 GB	1.5 TB
Avg. unique Vol./Gen.	60 GB	235 GB

the three approaches as well as their performance in relation to the memory usage.

A. Data Sets and Methodology

We have used two data sets, HPC_HOME and MS, for the comparison with DDFS and SI: HPC_HOME consists of one full and 60 daily incremental backups of up to 260 home directories on our university's HPC cluster. MS is a subset of a data set collected from desktop computers at Microsoft [12]; for this, 140 nodes were randomly selected and their incremental backups extracted. The chunks were generated using content-defined chunking with an expected chunk size of 8 KB [13]. Table I summarizes the core statistics.

We have performed the comparison based on a reimplementation of Meister et al.'s simulator [14]–[16]. DDFS and SI have been configured in a way that they use the same amount of main memory. Based on the size of the data sets, we have modeled a hardware setup that can store up to 16 TB of unique chunk data and provides 8 GB of main memory for all data structures necessary to identify incoming chunks. 8 GB has been chosen to keep a reasonable ratio between the cache size and the backup size. Bigger memory sizes would lead to systems, where the complete chunk index could be kept in main memory. A full index for the HPC_HOME data set consumes, e.g., 13 GB of main memory assuming 28 B per chunk entry. Different memory settings are discussed in Section III-F.

The available memory has been distributed as follows:

We have configured the DDFS's Bloom filter to lead to an expected false positive rate of 1% with the optimal number of hash functions. This gives a fixed memory consumption of 2.4 GB for 8 KB chunks. The remaining memory has been used for the container cache. We have simulated DDFS using a container size of 4 MB with an assumed compression ratio of 50%. Thus, each container can 1024 chunks on average.

SI identifies similarities based on segments and the sparse index is the main data structure for identifying similar segments. In addition, it allows SI to detect additional already stored chunks by loading and caching the referenced segment's manifests. We have determine in our implementation the hook chunks stored in the sparse index by sampling based on the k most significant bits of the chunks' fingerprints. Therefore, we have maximized the index size first by maximizing k. We have then filled the remaining memory with the manifest cache. For 8 KB chunks, this leads to a sampling based on the 4 most significant bits and a cache capacity of 27,396 manifests. In other words, the system only adds on average every 16th unique chunk to the sparse index.

SCI's simulated memory consumption is 1 GB for the first level of the LSM tree plus the size of one page.

Our main interest is on the number of I/Os which each approach requires to identify a given chunk as duplicate or new. This number is still the most important influence factor for system performance. We do not count I/Os for writing chunk raw data to disk as a system can perform them asynchronously. We also do not count the I/Os of the first backup generation because we are interested in the steady state of the approaches. For comparability between the data sets, we show the average number of generated I/Os per 1000 chunks ($\frac{\text{I/Os}}{\text{1K chunks}}$) and average this value over all backup generations.

We furthermore assume that if an approach generates less disk accesses than another one, a concrete implementation would also perform better, especially if a similar hardware setup is used and the same performance tuning is possible. However, we refrain from estimating the throughput based on this information. Measuring the throughput is only possible in a real implementation.

B. LSM Tree Page Size

The page size of the LSM tree is an important parameter of a sorted-streams system. On the one hand, it determines the total number of pages, which is an upper bound for the number of I/Os during an index run because each page is read at most once. On the other hand, it influences the probability of the pages to be hit by a fingerprint and loaded by the system. For this reason, it is interesting to find out whether small page sizes allow the system to skip pages and, by doing so, speed up index processing.

The aim of this section is to show that, for typical system parameters and a reasonable number of parallel streams, all pages will be hit and skipping pages can therefore not help to increase the performance.

We set the page size to 1 MB as this is the smallest size which is still reasonable considering the cost of disk seek operations. A quick evaluation on our hardware has shown no performance difference between reading a page and skipping the next and reading both pages for page sizes up to 1 MB. For the total backup size, we assume a minimum of 2^{24} distinct chunks per backup run, which is easily required in the presence of thousands of clients. To make the calculations more accessible, we increase the page entry size from 28 B to 32 B, the next power of two. This is permissible because a larger entry size only reduces

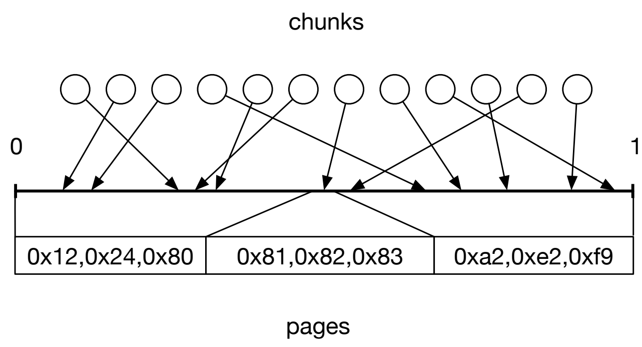


Figure 6. Logically small pages problem. The middle page covers a small fraction of the $[0, 1]$ interval and, therefore, is hit by the chunk fingerprints with a low probability.

the pages' probability for being hit. The number of entries per page is $\frac{1\text{MB}}{32\text{B}} = 2^{15}$. We assume the commonly used average chunk size of 8 KB and a matured deduplication system, which is filled with 16 TB of unique chunks. Hence, the number of chunks already being stored in the system is $\frac{16\text{TB}}{8\text{KB}} = 2^{31}$, and the number of pages is $\frac{2^{31}}{2^{15}} = 2^{16}$.

For the analysis, we model the problem as a balls-into-bins game. It differs from the standard game, however, because the probabilities of the pages (bins) to be hit by a fingerprint (ball) are proportional to the page's fingerprint range and therefore not uniform[1]. Figure 6 shows an example for three pages, each having a capacity of three entries. As the middle page covers a smaller interval, it has a lower probability to be used.

Hence, the key element of our analysis is to show in Lemma 1 that the fingerprint ranges of all pages have a sufficient minimal length with high probability and to show in Lemma 2 that, in expectation, all pages will be hit.

Lemma 1. *Let $m, s, \ell \in \mathbb{N}$, $d, n \in \mathbb{R}$ and $S = \{0, 1, 2, ..., s\}$ where $n = s/d \geq 4 \cdot m \geq 2^{16}$ and $n \cdot (\log(n))^2 \leq \ell \leq n \cdot (\log(n))^3$. Assume that ℓ distinct points are randomly picked from S and that S is divided into m subintervals $b_1, ..., b_m$ such that each b_i contains exactly ℓ/m consecutive points. Then all subintervals have length at least d with probability at least $1 - n^{-3}$.*

Proof: In this lemma, m represents the number of pages and ℓ the number of unique chunks out of the $|S|$ possible fingerprints, where each page has to store ℓ/m fingerprints. We are interested in the minimal range of fingerprints over all pages, which is denoted by d and which will be checked later.

The proof is based in a first step on the idea that the number of fingerprints assigned to each of n helper bins which are each responsible for an interval of length $n = s/d$ can be bounded by standard balls-into-bins theorems:

Partition the interval $[0, s]$ into $n = s/d$ subintervals $J_1, ..., J_n$ of size d. Applying Theorem 1 of [17], the maximum number of points in any such interval is upper bounded by

$$\ell/n + 2 \cdot \sqrt{2 \cdot \log(n) \cdot \ell/n}$$

with probability at least $1 - n^{-3}$.

We will now use this density for the n bins to build a relationship between our sought interval length d and the number of fingerprints m/ℓ stored in each page. Therefore consider any arbitrary interval $I_{x,d} = [x, x+d]$, $0 \leq x \leq s-d$, and denote the number of points in this interval by $r(I_{x,d})$. Since $I_{x,d}$ intersects with at most two of the subintervals from $\{J_1, ..., J_n\}$, we obtain

$$\begin{aligned} r(I_{x,d}) &\leq 2 \cdot \left(\ell/n + \sqrt{8 \cdot \log(n) \cdot \ell/n}\right) \\ &\leq 2 \cdot (\ell/n + \ell/n) = 4 \cdot \ell/n \leq \ell/m \end{aligned}$$

using that $\ell \geq n \cdot (\log(n))^2 > 8 \cdot n \cdot \log(n)$ with $\log n > 8$ for $n > 2^{16}$. It follows that every interval b_i, $i \in 1, ..., m$, has length at least d with probability at least $1 - n^{-3}$. ■

Lemma 2. *Consider a balls-into-bins game in which the bins b_i have different probabilities p_i. For some $n \geq 25$, let $B = \{b_i \mid p_i \geq 1/n\}$ be the set of all bins b_i which have a probability of at least $1/n$. Let M be the number of balls necessary to allocate at least one ball to every $b_i \in B$. Then it holds that*

$$E[M] \leq n \cdot (\ln(n) + 1.6).$$

Proof: The result follows directly from the *Coupon Collector's Problem* (see, e.g., [18]). Assume that there are $n \geq 25$ coupons and that in every round one coupon is randomly selected. If every coupon has the same probability $1/n$ to be selected, the number of rounds T necessary to collect all coupons is known to be

$$E[T] = n \cdot H_n \leq n \cdot (\ln(n) + 1.6)$$

where H_n is the nth *harmonic number* [19].

This problem can be rephrased as a balls-into-bins game with n bins where, in every round, each bin has the same probability to receive the ball. Here, T is the number of balls necessary until every bin has received at least one ball.

Now consider the balls-into-bins game defined in the statement of the lemma. Since in every round the probability for every bin $b_i \in B$ to receive the ball is at least $1/n$, we can bound the expected number of balls by

$$E[M] \leq E[T] \leq n \cdot (\ln(n) + 1.6).$$

■

[1]In the worst case, a page could be filled with $\frac{1\,MB}{32\,B} = 2^{15}$ *consecutive* hashes. The probability of a chunk hitting such a page would be as small as $\frac{2^{15}}{2^{160}} = \frac{1}{2^{145}}$.

In the following, we will combine the two Lemmas. First we apply Lemma 1: $S = \{0, 1, ..., s-1\}$ is the set of all fingerprints ($s = |S| = 2^{160}$), $m = 2^{16}$ the number of pages, $\ell = 2^{31}$ the number of chunks stored in the system, and we set the minimal length of any fingerprint range to $d = 2^{140}$.

It is easy to check that these values actually fulfill the conditions, namely $n = s/d \geq 4 \cdot m \geq 2^{16}$ and $n \cdot (\log(n))^2 \leq \ell \leq n \cdot (\log(n))^3$, and that the way the interval $[0, s]$ is divided into m subintervals (or ranges) b_i conforms to the procedure described in Section II-B1. The result is that all subintervals b_i have a length of at least $d = 2^{140}$ with probability at least $1 - n^{-3} = 1 - 2^{-60}$.

Hence, the probability of each subinterval b_i to receive a chunk hash is $p_i \geq d/s = 2^{-20}$. Now let M count the number of chunk hashes necessary to allocate at least one hash to each subinterval. Applying Lemma 2, we get:

$$E[M] \leq \frac{s}{d} \cdot \left(\ln\left(\frac{s}{d}\right) + 1.6\right) = 2^{20} \cdot (\ln(2^{20}) + 1.6) < 2^{24}.$$

Hence, in expectation less than $2^{24} \cdot 8\,\text{KB} = 128\,\text{GB}$ different chunks are required to use every page in our setting.

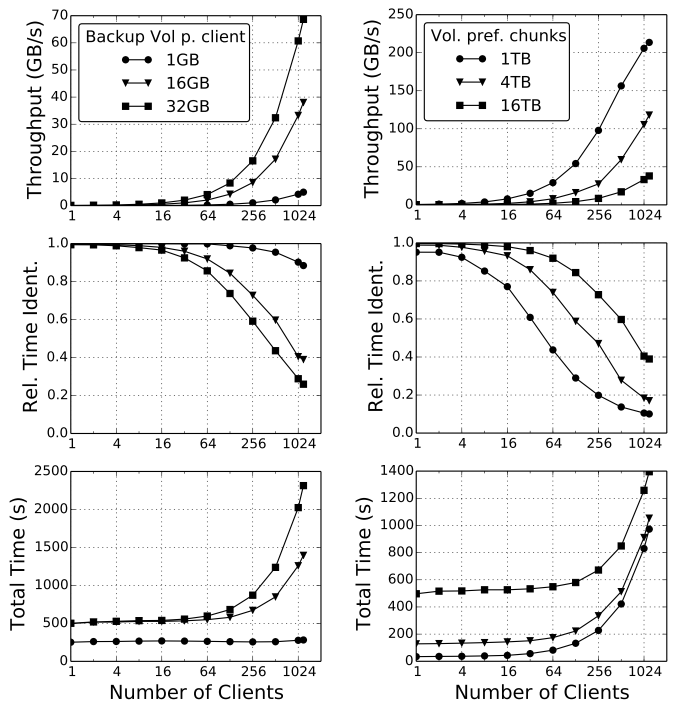
Besides the more theoretical analysis, we have also experimentally measured the share of used pages. For this, the simulated system has been populated with a number of backup generations before the subsequent generation has been simulated with a varying page size and number of clients. In the case of HPC_HOME, the system has been filled with 3.5 TB of unique raw chunk data from the first 37 backup generations, and the 38th generation has been analyzed. For MS, we have populated the system with 7.1 TB from the first 22 generations, and the 23rd has been analyzed. For both data sets, the system has used all pages for 11 clients.

We conclude that page skipping is only possible if the number of clients and backup sizes are small, and that sorted-streams systems can use bigger pages. For the following experiments, we choose a page size of 128 MB. The number of I/Os for other sizes can be easily computed due to the fact that every page is used with high probability.

C. Server prototype

We have implemented a server prototype and have evaluated it for a varying number of clients. The clients simulate backups of equal size by randomly generating chunk fingerprints. The server receives the fingerprints, returns the storage locations and finally receives the recipes and the raw data of the new chunks (cf. Section II-B2 and Figure 2); though, due to hardware limitations, the raw data is not stored.

The server runs on a node equipped with an Intel Xeon CPU with 4 cores@3.3 GHz (8 hardware threads), 16 GB RAM, a 10 GBit ethernet network, and a single HDD. For all experiments we generate random input data while assuming an average chunk size of 8 KB, a deduplication ratio of 90%



(a) Volume of prefilled system fixed to 16 TB

(b) Backup Volume per client fixed to 16 GB

Figure 7. Server performance for varying number of clients. Left: varying backup sizes for a system prefilled with 16 TB of unique chunks. Right: fixed backup size per client for different volumes of prefilled chunks.

per client, and a compression rate of 50% for the chunk raw data when it is sent over the network. Each run uses a randomly pre-generated chunk index to simulate an aged system. We do not use our real world data sets since they only include up to 597 clients.

Figure 7 shows different performance metrics for the prototype. Figure 7a shows results for a varying number of clients and different backup volumes, while the experiments always start for a deduplication system storing 16 TB of unique chunks. For Figure 7b, we have fixed the backup volume per client to 16 GB, while varying the number of clients and the initially stored volume of data. This volume influences the size of the chunk index as the index has to store an entry for each unique chunk. This, in turn, increases the number of index pages the system processes during a single run.

The first thing we are interested in is how fast SCI can identify incoming chunk fingerprints. Thus, we consider the first two steps of the communication (cf. Figure 2), but ignore the transmission of recipes and raw data. The upper figures show the server's throughput, which is defined as the added (logical) volume of all client backups divided by the time necessary to identify all chunk fingerprints.

The processing time for one page consists of waiting for the page load (T_{load}) and waiting for the system to process all fingerprints for the page (T_{fp}). During a run, the system can prefetch the next page in parallel to processing the current one, therefore the processing time for a single page is $max(T_{load}, T_{fp})$. T_{fp} depends on the CPU only and is bigger than T_{load} if many fingerprints fall into the range of a single index page. This is the case for small indices or big backup volumes. In the experiments, the CPU has not been the performance bottleneck, therefore the identification throughput doubled for each doubling of the number of clients.

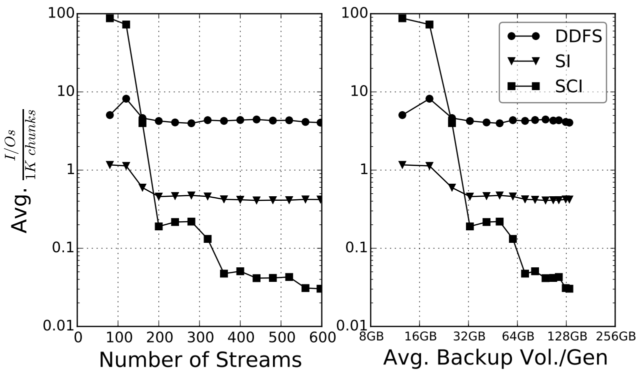
After the chunk identification, the clients send their recipes plus the chunk raw data in the last communication step (see Figure 2). Assuming the same deduplication ratio for each client, doubling the number of clients roughly doubles the volume of new chunks. Based on the high identification throughput, the transmission of the raw chunk data becomes dominant for the total runtime. This is visible in the middle row in Figure 7, which shows the identification's fraction of the total runtime, i.e. all communication steps. The last row shows the total runtime.

D. Comparison with DDFS and SI

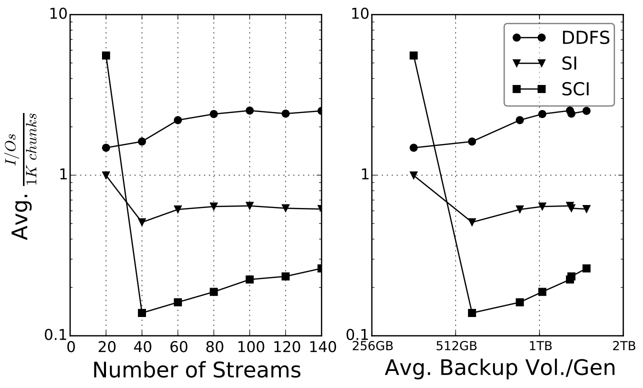
In order to perform better than the established systems DDFS and Sparse Indexing, SCI requires a minimum backup size. If the backup size is small, there is still a high probability that SCI loads most of its pages. In contrast, DDFS and Sparse Indexing will perform only few fetch operations provided that the incoming stream has a high chunk locality. DDFS fetches and uses the metadata sections of its *containers* for chunk identification. Containers are the data structure holding the chunk raw data. The system maintains one open container per stream and stores it on disk when it reaches a certain size. Therefore, a container stores the chunks of a stream in order of their arrival. If the backup stream shows only little variation over consecutive backup generations, the system can identify many chunks with a single container fetch. However, this mechanism only identifies old chunks, so the system additionally maintains a Bloom Filter to avoid I/Os for most of the new chunks.

Sparse Indexing (SI) is an approximate approach proposed by Lillibridge et al. [2]. Here, the chunks are sequentially grouped into segments. Fingerprints of a small subset of chunks are chosen as *hooks* and stored in a RAM-based chunk index. For each new segment, the most similar segments are determined by counting the number of shared hooks. These segments are used to determine duplicates in the new segment. The approach caches the segment recipes (*manifests*) to save I/Os. Since the segments are constructed in chunk arrival order, the system can identify many consecutive chunks with a a single manifest fetch.

Both data sets show less interference among the streams. For HPC_HOME, 99% of all chunks appear only in one stream (98% for MS). On the one hand, this increases cache



(a) Data set HPC_HOME



(b) Data set MS

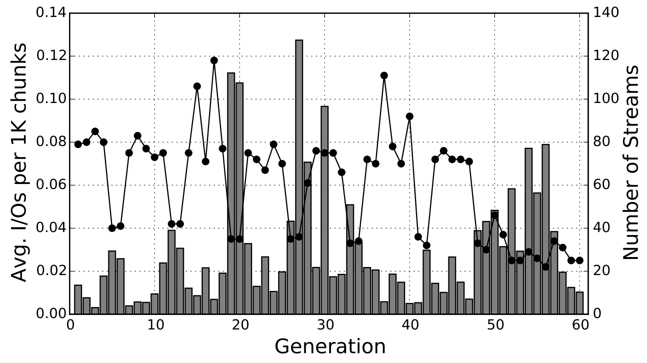
Figure 8. Average $\frac{\text{I/Os}}{\text{1K chunks}}$ for different number of streams. The right side shows the same results as a function of the average backup volume per generation, i.e. the backup volume of all clients before any deduplication, averaged over all generations.

pollution because of low synergy effects. On the other hand, this increases chunk locality since DDFS and SI fill the containers or segments with chunk information of the same stream.

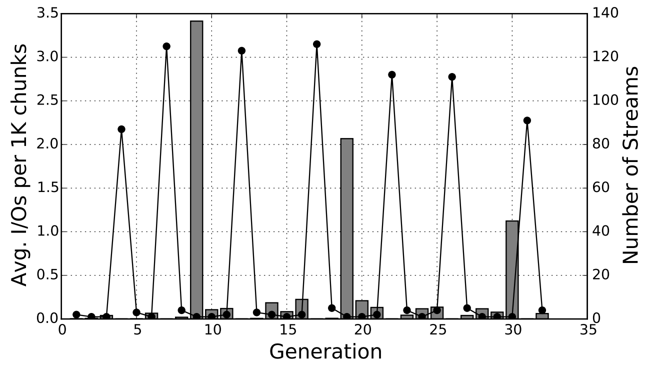
However, we show that SCI generates less I/O after reaching the threshold of a certain backup size because its number of generated I/Os only grows with the volume of unique chunks while the I/Os of the other approaches are also based on the following factors:

- The I/Os of the other approaches grow with the volume of the **logical backup size**. With typical deduplication ratios, this benefits SCI over DDFS and SI after a certain size of logical backup data per backup generation.
- Even if the individual backups are small, the total volume still grows with the **number of streams**.
- **Aging** decreases the locality fetched by containers and segments. For DDFS this leads to more I/O operations. SCI is more robust against these effects as shown in Section II-A.

In the following, we evaluate the number of generated I/Os for all systems. For SCI, we count each page load as



(a) Data set HPC_HOME (597 different streams)



(b) Data set MS (140 different streams)

Figure 9. Average $\frac{\text{I/Os}}{\text{1K chunks}}$ (bars) and the number of streams (lines) for each generation.

a single I/O operation. We generate smaller data sets by sampling the number of backup streams in HPC_HOME and MS. We have chosen the streams randomly, but ensured that the streams in a smaller set are included in the bigger ones. It is important to notice that not all streams participate in every backup generation (see later explanations within this subsection).

The left part of Figure 8 shows the average number of $\frac{\text{I/Os}}{\text{1K chunks}}$ for different numbers of streams. The right side shows for the same experiments the average number of $\frac{\text{I/Os}}{\text{1K chunks}}$ depending on the average backup volume per generation. This volume includes all logical backup data, i.e. the data that a client would write to DDFS or SI. For comparability, we ignore the client side deduplication of SCI since we do not assume that the other approaches have the same feature.

For HPC_HOME, SCI generates 87 $\frac{\text{I/Os}}{\text{1K chunks}}$ for 80 streams, where the 80 stream correspond to an average added capacity of 12 GB per generation. This values decrease to 0.19 $\frac{\text{I/Os}}{\text{1K chunks}}$ for 200 streams (32 GB/gen) and, finally, to 0.03 for 597 streams (131 GB/gen).

The reason for the high number of $\frac{\text{I/Os}}{\text{1K chunks}}$ for a small number of streams is the limited number of backup generations and the corresponding deviation of the backup sizes. For example, there is one generation for the 80 streams experiments which consists of only three chunks, which cause three page load operations and therefore $1000 \frac{\text{I/Os}}{\text{1K chunks}}$. Such generations have a huge impact on the average, even if the system behaves nearly optimal for them. These small generations become less frequent as more backup streams are added. For 597 streams, Sparse Indexing generates with $0.36 \frac{\text{I/Os}}{\text{1K chunks}}$ already 12 times more I/Os than SCI. DDFS generates even 133 times more accesses than SCI, resulting in $4 \frac{\text{I/Os}}{\text{1K chunks}}$

For the MS data set, SCI generates 5.6 $\frac{\text{I/Os}}{\text{1K chunks}}$ for the smallest data set with 20 streams and an average backup volume of 353 GB per generation. This value decreases to 0.14 for 40 streams (573 GB) and increases to 0.26 for 140 streams (1.5 TB). Similar to HPC_HOME, there is a small backup generation (219 chunks) which causes a bias of the average $\frac{\text{I/Os}}{\text{1K chunks}}$. The median $\frac{\text{I/Os}}{\text{1K chunks}}$ is 0.03 for the 20 streams data set. DDFS generates 2.5 $\frac{\text{I/Os}}{\text{1K chunks}}$ on average for the biggest data set. Sparse Indexing generates 0.6 $\frac{\text{I/Os}}{\text{1K chunks}}$.

For MS, SCI's average number of $\frac{\text{I/Os}}{\text{1K chunks}}$ increases because of a set of small backup generations. These generations consist of computers that were traced only with few others on the same day. This has two effects: First, it adds further small backup generations as we increase the number of backup streams; second, the size of these generations is nearly constant as we increase the total number of streams. On the other side, the number of pages in the chunk index grows as each additional stream adds unique chunks. Therefore, these generations generate more $\frac{\text{I/Os}}{\text{1K chunks}}$ for a bigger total number of streams because their chunks are distributed over more pages. These outliers are visible in Figure 9b, which shows the average $\frac{\text{I/Os}}{\text{1K chunks}}$ (bars) and the number of streams (line) for each generation. The average $\frac{\text{I/Os}}{\text{1K chunks}}$ becomes more stable if more streams participate in a backup generation/backup run since each additional stream generates a more uniform distribution of chunk hashes to the LSM tree pages.

SCI generates in summary between 9.6 and 113 times less I/Os than DDFS on average for the biggest data sets. Compared to the approximate Sparse Indexing, it generates between 2.3 and 12 times less I/Os while still performing exact deduplication.

E. I/O Pattern

Another advantage of SCI over DDFS and SI is its simple sequential access pattern. For each approach, we have analyzed the pattern of how the system accesses the containers (DDFS), segments (SI) or pages (SCI) in which the chunks are stored.

In the experiment, the first 17 backup generations of the MS data set filled the system with 6.8 TB of unique chunks. Figure 10 shows the access pattern of the subsequent 18th generation for all three approaches, in which 126 streams wrote 4.5 TB to the system. Every dot in each of

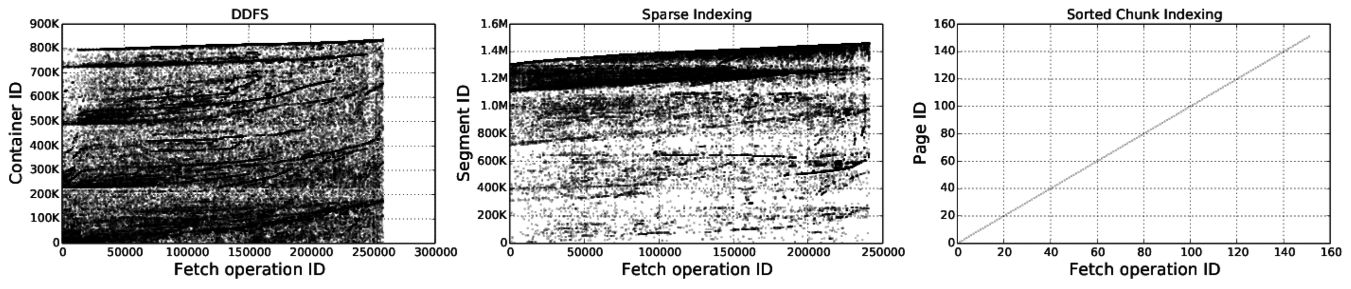


Figure 10. Generated I/O Pattern of DDFS, SI, and SCI.

the subfigures represents an access. It marks the ID of the operation and the ID of the container, segment or page from which the chunk was fetched. For DDFS and SI, the plots are restricted to 100,000 randomly sampled data points, and in the case of DDFS, we also disregard the random accesses to the disk-based chunk index.

DDFS and SI nevertheless generated a large number of accesses. Every stream has frequently fetched units (i.e., containers or segments) of its own past generations and also units of other streams. And although there is a range of units that were frequently accessed, the internal caches were not able to prevent the fetches. Sorted Chunk Indexing, on the other hand, generated a single series of sequential accesses to the pages.

The difference in the number of fetch operations is caused by the big backup generation size. As mentioned in the last section, DDFS and SI depend on the backup volume while SCI only depends on the volume of the previously stored unique chunks.

F. Memory Usage

In our experiments, we have set the available main memory to 8 GB to keep a reasonable cache size to backup size ratio. However, deduplication servers would contain a multiple of this amount, which could favor DDFS and SI. In the following, we investigate how different memory sizes affect the I/O generation.

For DDFS and SI, we use the available memory as described in Section III-A. For SCI, we adjust the flushing threshold of the memory-based first level of the LSM tree. Figure 11 shows the average $\frac{\text{I/Os}}{\text{1K chunks}}$ for the full MS data set and different memory sizes. For the approximate Sparse Indexing, it also shows the relative amount of undetected duplicates (dashed line, second y-axis) since different memory amounts also affect the detection rate.

For DDFS, there are no values smaller than 4 GB because the assumed Bloom filter incurs a minimum size. The approach benefits from high main memory situations until the cache can hold the metadata of all generated containers. After this point, its main source of I/Os is the false positive ratio of its Bloom filter. Each misspredicted new unique chunk generates a lookup in the disk-based chunk index.

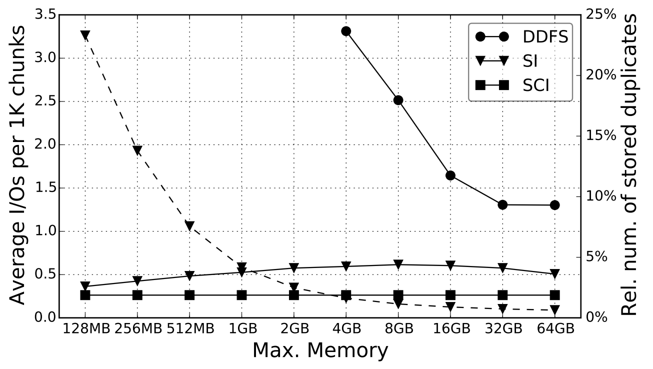


Figure 11. Average $\frac{\text{I/Os}}{\text{1K chunks}}$ for different memory consumption for the MS data set.

Sparse Indexing generates between 0.36 and 0.64 $\frac{\text{I/Os}}{\text{1K chunks}}$ on average. The average relative number of undetected duplicates decreases exponentially from 23.3% to 0.06%. For a memory size of 64 GB, the system holds its sparse index without sampling. However, it still generates I/Os because it identifies duplicates based on its manifests which must be loaded from disk. SI does not detect all duplicates because it loads and considers only a limited number of manifests.

SCI generates less $\frac{\text{I/Os}}{\text{1K chunks}}$ than the other approaches. Its values are almost the same because the simulation flushes the memory part of the LSM tree after every backup generation and there are few generations that are big enough to trigger a flush during their run. Without the flush after each generation, the values would be even smaller since the disk part of the LSM tree receives pages at a later point.

IV. Related Work

Fingerprint-based data deduplication has been a very active research topic over the recent years. A major issue has always been the *chunk-lookup disk bottleneck* [2], which occurs when a system uses a central disk-based index to identify chunks as duplicates. The approaches to overcome this challenge can be distinguished into two main concepts: exact (e. g. used in DDFS [1]) and approximate (e. g. Sparse Indexing [2]). There are many systems following these approaches. Exact means that every chunk is stored exactly once [20], [21] while a chunk can be stored redundantly in systems following the approximate approach [3], [5], [22], [23]. All exact and all approximate deduplication approaches directly exploit the given locality in the data streams. Our system, on the other hand, replaces the given locality with a new locality by sorting fingerprints.

Tan et al. present a backup system for clouds called *CAB-dedupe* [24]. The authors use a causality-based deduplication model to reduce the data transferred from clients to the deduplication server. For this, the clients require information about previous backup runs. This is not necessary in our system. Instead we reduce the data transferred by only sending the new data. This is achieved by sending the fingerprints first so that the server can inform the client which chunks are new.

Clements et al. describe a distributed deduplication system that is embedded in a SAN cluster file system [25]. Similar to SCI, DeDe aims for efficient chunk index accesses by exploiting the sorted order of chunk fingerprints. However, DeDe uses the order for efficient index updates. Updates to the index first are accumulated in logs. Regularly, DeDe sorts the updates by the hash value and performs a merge operation of the updates and the index.

Beaverson et al. describe in a patent application a system that stores fingerprints in an index structure which consists of index structure portions [26]. Each index structure portion is associated to a range of token values. The token values are stored in sorted order in a B-tree. The system receives the chunks and their corresponding values from multiple sources. The received data is first sorted and stored in the deduplication system. At a later point, the system checks the data for uniqueness using the index portion structures. The system is not intended for online-deduplication environments and its description lacks crucial details and allows interpretations, as it is common in patents. For example, the authors do not provide information how clients can be handled in parallel nor an evaluation of their system.

In cloud environments, cross-user and source-based deduplication can affect the data privacy and security. Harnik et al. [27] observe high redundancies among different backup sources and demonstrate their potential for malicious attacks. They argued for moving the complete deduplication process to the servers and advise clients to avoid deduplication by encrypting their data. In contrast, the aim of our online backup system is not security, but efficiency, and we involve the clients to relieve the servers.

Several systems exploit the similarity of different backup data [3], [5], [28], [29] using Broder's theorem [30] and comparing the smallest fingerprints of two sets of fingerprints. Douglis et al. [28] extend this idea; they sort both sets and compare the n smallest fingerprints of each set. In our system we also sort fingerprints, but for the purpose of avoiding I/O operations on the server.

V. Conclusion

In this paper we have described a new high-performance deduplication system for the parallel processing of multiple backup streams. It sorts the chunks of all streams and coordinates their accesses to the fingerprint index. The main benefits of this approach are:

- It forces all concurrent fingerprint identification processes to use the same part of the index while it resides in memory.
- It ensures that all concurrent chunk identification processes access the same region in the index only once. In combination with the first property, this is especially helpful if the number of incoming data streams is big because it reduces the number of I/Os for each index "region" to one, independently of the number of streams.
- It ensures a disk-friendly access pattern by iterating over the chunk index in the order of the sorted fingerprints.

As a proof of concept, we have implemented a prototype of the server. It has shown that it can identify the chunk fingerprints at a rate of up to 222 GB/s, i.e. it identifies the chunks of 222 GB worth of backup data as old or new per second. In addition, we have compared *Sorted Chunk Indexing* (SCI) with the *Data Domain deduplication file system* of Zhu et al. and *Sparse Indexing*, the approximate deduplication approach by Lillibridge et al. The experiments revealed that SCI generates up to 113 times less I/Os than DDFS and up to 12 times less than SI while consuming less memory, generating a disk-friendly access pattern, and still performing exact deduplication.

In future work, we plan to implement the client and evaluate the system in its complete setup. Furthermore, we plan to implement the distributed mode.

Acknowledgment

We thank the anonymous reviewers for their helpful suggestions and our shepherd, Philip Shilane, for the excellent support. This work was partially funded by the German Ministry for Education and Research (BMBF) under project grant 01|H13004 (FAST).

References

[1] B. Zhu, K. Li, and R. H. Patterson, "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. 269–282.

[2] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble, "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.

[3] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge, “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.

[4] S. Quinlan and S. Dorward, “Venti: a new approach to archival storage,” in *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 2002, pp. 89 – 101.

[5] W. Xia, H. Jiang, D. Feng, and Y. Hua, “SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with low RAM Overhead and High Throughput,” in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011.

[6] J. Kaiser, D. Meister, A. Brinkmann, and T. Süß, “Deriving and Comparing Deduplication Techniques Using a Model-Based Classification,” in *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*. ACM, 2015.

[7] G. Amvrosiadis and M. Bhadkamkar, “Identifying trends in enterprise data protection systems,” in *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX, 2015, pp. 151–164.

[8] P. E. O’Neil, E. Cheng, D. Gawlick, and E. J. O’Neil, “The Log-structured Merge-Tree (LSM-tree),” *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[9] EMC Coperation, “Data domain boost software,” http://www.emc.com/collateral/software/data-sheet/h7034-datadomain-boost-sw-ds.pdf, 2012.

[10] M. Lillibridge, K. Eshghi, and D. Bhagwat, “Improving restore speed for backup systems that use inline chunk-based deduplication,” in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013, pp. 183–198.

[11] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, and Q. Liu, “Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information,” in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2014, pp. 181–192.

[12] D. T. Meyer and W. J. Bolosky, “A study of practical deduplication,” *ACM Transactions on Storage (TOS)*, vol. 7, no. 4, p. 14, 2012.

[13] M. O. Rabin, “Fingerprinting by random polynomials,” TR-15-81, Center for Research in Computing Technology, Harvard University, Tech. Rep., 1981.

[14] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, “A study on data deduplication in HPC storage systems,” in *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, November 2012, pp. 7:1–7:11.

[15] D. Meister, J. Kaiser, and A. Brinkmann, “Block locality caching for data deduplication,” in *Proceedings of the 6th International Systems and Storage Conference*. ACM, 2013, p. 15.

[16] D. Meister, A. Brinkmann, and T. Süß, “File recipe compression in data deduplication systems,” in *Proceedings of 11th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 2013, pp. 175–182.

[17] M. Raab and A. Steger, “Balls into bins - a simple and tight analysis,” in *Proceedings of the 2nd International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM’98)*, ser. LNCS, vol. 1518. Springer-Verlag, Oct 1998, pp. 159–170.

[18] R. Motwani and P. Raghavan, *Randomized Algorithms*. New York, NY, USA: Cambridge University Press, 1995.

[19] P. Erdős and A. Rényi, “On a classical problem of probability theory,” *Magyar Tudományos Akadémia Matematikai Kutató Intézetének Közleményei 6: MR 0150807*, p. 215–220, 1961.

[20] J. Wei, H. Jiang, K. Zhou, and D. Feng, “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)*. IEEE, May 2010, pp. 1–14.

[21] S. Rhea, R. Cox, and A. Pesterev, “Fast, inexpensive content-addressed storage in Foundation,” in *Proceedings of the USENIX 2008 Annual Technical Conference (ATC)*. USENIX, 2008.

[22] F. Guo and P. Efstathopoulos, “Building a highperformance deduplication system,” in *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX, 2011, pp. 25–25.

[23] D. N. Simha, M. Lu, and T. Chiueh, “A scalable deduplication and garbage collection engine for incremental backup,” in *Proceedings of the 6th Annual International Systems and Storage Conference (SYSTOR)*, 2013, p. 16.

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

[25] A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li, “Decentralized deduplication in san cluster file systems,” in *Proceedings of the*, ser. USENIX’09. Berkeley, CA, USA: USENIX Association, 2009.

[26] A. Beaverson, B. Yang, and J. Pocas, “Patent No.: US 7,930,559 B1,” EMC Corporation, April 2011.

[27] D. Harnik, B. Pinkas, and A. Shulman-Peleg, “Side channels in cloud services: Deduplication in cloud storage,” *IEEE Security & Privacy*, vol. 8, no. 6, pp. 40–47, 2010.

[28] F. Douglis, D. Bhardwaj, H. Qian, and P. Shilane, “Content-aware load balancing for distributed backup,” in *Proceedings of the 25th Large Installation System Administration Conference (LISA)*, 2011.

[29] D. Teodosiu, N. Bjorner, Y. Gurevich, M. Manasse, and J. Porkka, “Optimizing file replication over limited bandwidth networks using remote differential compression,” *Microsoft Research TR-2006-157*, 2006.

[30] A. Z. Broder, “On the resemblance and containment of documents,” in *Proceedings of the Conference on Compression and Complexity of Sequences*. IEEE, 1997, pp. 21–29.