Deduplication: Concepts and Techniques

William Jannen
jannen@cs.williams.edu

April 25, 2021

1 Document Overview

This document's goal is to present a survey of the general deduplication design space. Deduplication is a broad concept, and deduplication systems can be implemented using a variety of techniques and design choices. For each core design element, the document presents the high-level idea and a set of considerations that influence an appropriate choice for that design. The document also presents a few representative systems as case studies, hopefully giving a sense for practical parameter choices and their implications. If there are parts of the text that are unclear, please let me know so that I can improve them! I am not aware of any good surveys/introductory texts on deduplication, so this is my attempt to create one for us to use in this class.

Learning objectives. After reading, you should be able to answer the following questions:

Definition. At a high level, what are the core tasks that deduplication systems perform?

Measurement. How do you quantify the effectiveness of a deduplication system so that we can compare it against other designs?

Timing. When do you perform deduplication? What is the difference between an inline and an offline deduplication system? How do they trade performance and space consumption?

Data structures. What is a chunk? What is a fingerprint? What is the fingerprint index, and how is the fingerprint index used?

Parameters. What knobs can be tuned in a typical deduplication system design, and how do different choices affect performance, deduplication ratio, metadata overhead, and fragmentation?

Use cases. In what scenarios are deduplication techniques beneficial? What characteristics of an environment present promising opportunities or difficult challenges for a deduplication system to manage?

2 Introduction

Data deduplication is a form of compression. The name itself, *deduplication*, hints at a deduplication system's overarching goal, which is to **identify duplicate data**, and **eliminate repeated instances** of that data. Successfully identifying and eliminating redundant data can drastically decrease the storage requirements for systems that manage large volumes of data. This is especially true for systems which maintain multiple copies of identical files—such as a system that performs nightly backups—or for systems that store many files with large, overlapping regions—such as boilerplate or configuration files that are part of an OS distribution. Even when storage capacity is not an issue, there are other reasons that deduplication may be valuable; for example, when sending data from a client to a server, a client that knows that the server already has a copy of the data it wishes to send can avoid transferring unnecessary data, avoiding network overheads.

Completely eliminating *all* duplicate data is an admirable goal, but deduplication systems may tolerate some replication for a variety of reasons. For example, allowing some redundant data may be necessary in order to complete important tasks in a reasonable amount of time (identifying all duplicates may be expensive!), or tolerate system failures (if we lose the only copy of a file system superblock, our whole system is hosed). Thus, we would like some measure that describes a deduplication system's effective "data savings". For that, we turn to the **deduplication ratio**.

Concretely, if N bytes of data are presented to a storage system S, which unambiguously represents that data using B bytes, then the deduplication ratio of D is defined to be:

$$D_S = \frac{N}{B} \qquad (1)$$

A more efficient deduplication system will have a higher deduplication ratio (i.e., it represents more data using fewer bytes).

D_S is not the only measure that we care about. Latency, throughput, scalability, and resiliency are all system design concerns, and the right balance of performance and deduplication efficiency is going to be context dependent.

2.1 Deduplication in a nutshell

Deduplication system designs differ along many dimensions, but at a high level, they all follow the same basic formula. The system receives a data stream as input, and it breaks that stream into discrete parts called **chunks**. For each chunk, the system calculates a collision-resistant hash (e.g., SHA-1, MD5). This hash value is commonly referred to as the chunk's **fingerprint** because the fingerprint uniquely identifies the chunk's contents: the same data always produces the same fingerprint, and two chunks that differ by even a single byte should have independent fingerprints. The set of unique chunk hashes (fingerprints) and the locations of the data that those fingerprints describe (chunk pointers) are stored in a data structure that can be searched and updated when new data is written. Regardless of which data structure is used, this repository of ⟨`fingerprint, chunk pointer`⟩ mappings is called the system's **fingerprint index**.

From this basic outline, we can start to see a rough deduplication workflow. As the system processes a data stream, it breaks the stream into successive data chunks. For each chunk, the system calculates the chunk's unique fingerprint and checks the fingerprint index to see if that chunk is new. If the fingerprint is missing (i.e., the chunk is new), the system writes the chunk and adds a new ⟨`fingerprint, chunk pointer`⟩ pair to the index for future data sharing. If the fingerprint is found in the index (i.e., the chunk is a duplicate), then the system does not write the chunk; instead, the system stores a logical pointer to the existing chunk, which is now shared.

A deduplicated data stream can also be restored from a list of fingerprints in a straightforward manner. To reassemble the original data stream, the stream's sequence of fingerprints are iteratively found in the fingerprint index, and the data referred to by the chunk pointers are added to the stream.

2.2 Design Choices

Although a given deduplication system can be large and complex, there are three primary design decisions within the basic deduplication framework:

- when to perform deduplication,
- where to perform deduplication, and
- what to deduplicate

This section breaks down these choices and explains the considerations that might lead system designers to make particular choices.

2.2.1 When?

The first choice in the design of any deduplication system is *when* to perform deduplication. The two main classes of deduplication systems are **inline** and **post-process** deduplication [7].

Inline. In an inline deduplication system, the data stream is chunked and hashed *before* it is written to disk [6]. The main advantage of the inline approach is that duplicate chunks are identified before they are written, saving I/O. However, inline deduplication introduces chunking, hashing, and index queries—expensive computations—onto the critical path of each potential write. As a result, inline deduplication systems generally suffer increased latency.

Offline. Post-process (or offline) deduplication eliminates data redundancy *after* the data has been completely committed to disk. This deferred processing keeps system responsiveness high, since duplicate elimination is done as a background process—possibly even on a remote system (imagine a system where a nightly deduplication process scans the system and copies all new *unique* data to a backup server). Unfortunately, duplicate data temporarily consumes space in the time between being written and the time that the data is identified as a duplicate and replaced with a reference to the common data.

Thus, an offline deduplication system may store more data than is strictly necessary, but with some extra work in the background, it can restore "good performance." In some ways, this trade-off is similar to the way that garbage collection in a log-structured file system or in a NAND flash SSD pushes the work of reclaiming space off of the critical path. However, there may be a point where the background work becomes a bottleneck, so sufficient resources must be available for this approach to be successful.

2.2.2 What?

A second choice for deduplication systems is *what* to deduplicate. Deduplication can be performed at the granularity of entire files [2], fixed-size blocks [11, 15], or variable-sized chunks [3, 10]. (Hybrid schemes are also possible, where the chunking strategy is dynamic. Hybrid schemes often leverage additional information, such as file type, to form heuristics and make decisions on a case-by-case basis about which chunking scheme is most appropriate [9]. For example, an `.mp3` file might be deduplicated at whole-file granularity, but a `.c` file may use variable-sized chunking. The rest of this section gives some intuition for why that might be a good design.)

Whole file. In whole-file deduplication, the system calculates one fingerprint for each file, regardless of the file's size. Files are deduplicated only if their entire contents are identical to an existing file. The downside of this coarse file-level granularity is that there are fewer sharing opportunities. For example, consider two large files which are identical up to the last byte; with whole-file deduplication, no data is shared whatsoever.

File granularity is inappropriate for deduplicating boilerplate code, configuration files, and files that are iteratively modified. In these contexts, no amount of data will be shared despite large regions of commonality.

Yet in some domains, whole file granularity is preferable. Compressed files and media files exhibit the property that even small modifications to the underlying data completely transform the final representation. As a consequence, either the whole file is identical or none of it is [9].

Fixed-size chunking. Dividing data into fixed chunk sizes, which are typically on the order of a few kilobytes, enables more fine-grained sharing than whole-file deduplication. Chunk boundaries are set at predetermined offsets in the data (often fixed-size chunks are set at 4KiB to match common page/block sizes). Using fixed offsets makes the task of dividing the data fast and computation-free, and fixed offsets lend themselves nicely to block-oriented storage hardware where I/Os must be aligned. System rules governing disk layout are much simpler as a result, and an appropriate block size can be chosen to manage internal fragmentation and index size.

Unfortunately, inserting or deleting data at the middle of a stream is difficult for a fixed-size-chunking system to handle—all data that comes after the insertion/deletion is shifted, affecting the boundaries of all subsequent chunks. The rest of the file must be rehashed, creating many "unique blocks" even though the data has not changed. Despite this so-called *boundary shifting problem*, fixed-size chunking still typically has better deduplication ratios than whole-file deduplication.

The next strategy, variable-sized chunking, gracefully handles both insertions and deletions.

Variable-sized chunking. A system that uses variable-sized chunking divides its data into potentially many, potentially small chunks in an attempt to maximize the sharing opportunities. The locations of chunk boundaries are determined by a computation over the data itself, which is why variable-sized chunking is also referred to as **content defined chunking**.

Content-defined chunking often yields the highest deduplication ratios. Like fixed-size chunking, variable-sized chunking creates many small segments, so there are many opportunities to share data among the logical objects in the system. Content-defined chunking also gracefully handles sharing in the case where new data is inserted into a stream; only the modified chunk (and potentially its neighbors) must be re-chunked and re-hashed in the common case [10]. (For more details, a variable-sized chunking example is shown in the LBFS case study.)

Despite its higher deduplication ratios, content-defined chunking introduces several costs into the end-to-end deduplication workflow. First, work must be done to identify the chunk boundaries. One popular strategy is to compute Rabin fingerprints [12] over a sliding window of the data. When the computation in a given window matches a target value, a chunk boundary is created. By changing the window size, we can change the target size of our chunks, giving us more control over the system's behavior.

The overheads that content-defined chunking adds to the cost of storing and querying the fingerprint index are more subtle. The more chunks there are in the index, the larger the indexing data structure becomes; for even modest data sets, the index may exceed the size of RAM and spill onto disk. This problem is compounded by the fact that fingerprint index queries have almost no locality of reference—a collision resistant hash is independently and uniformly distributed, so data that has locality in the workload does not have locality in the index. As index sizes grow, each look-up requires one or more disk seeks, introducing the chunk index **disk bottleneck** [16].

2.2.3 Where?

A third design decision is *where* to perform deduplication.

Source deduplication describes systems where an index is maintained or queried locally, and data is examined for duplicates before being sent to a remote server for storage. Data transfer is minimized because duplicate data is never transmitted across a network in source deduplication.

Destination deduplication systems remove the deduplication process from the client; all deduplication is done remotely. Destination deduplication minimizes client computation at the cost of network bandwidth.

In some systems [14], a lightweight index is used to check for existence at the source, but location data is maintained at the destination. The choice of source or destination deduplication determines where system resources must be allocated (should we have one very powerful system that is used as the destination, or should each local system dedicate a small amount of resources for its own work?). Location may also be determined by system constraints; general purpose workstations may not be well-equipped to meet deduplication requirements.

3 Case Studies

In this section, we discuss deduplication use cases and a few representative systems that target those use cases. Hopefully, the discussion highlights the properties of those workloads that lend themselves to different design decisions.

Table 1 provides a classification of a handful of systems. For each system, the system is categorized along the dimensions we've described above: where it performs deduplication (source or destination), what it deduplicates (chunking strategy), and the structure of its fingerprint index. Table 1 shows the diversity of design choices that systems make.

3.1 Archival

Perhaps the most frequent (and important) use case for data deduplication is data backup. In many industries, companies are required to maintain comprehensive records of old data, and keep that data for many years, in order to comply with government regulations. The characteristics of these large-scale data backups are drastically different from the workloads seen on single-user workstations in several important ways:

- a backup operation is a large streaming write, with no random accesses or reads
- a backup must complete in its entirety in a given window, often as a nightly process
- once written, the data in a backup will never change (it is immutable)
- data *may* be read in the future, but rarely (data is "cold storage"). Backups are often accessed only in emergencies or in "offline settings", like an audit, where latency is not the primary performance concern.
- data must always be accessible in its original form
- the aggregate volume of data will only ever grow—the system's capacity should be incrementally scalable

These archival workloads often prioritize deduplication ratio, favor throughput over latency, require resilience to data loss, and must scale extraordinarily well.

3.1.1 Case Study: Venti

Venti [11] is an early implementation of a content addressable archival repository, shareable among multiple clients. Venti implements a **write once** policy — once written, data cannot be modified or deleted by a user or administrator. The decision to archive data is permanent.

Venti writes data to an append-only log, divided into fixed-size, logical containers termed *arenas*. Data blocks may be variable sized, but the append only nature of arenas prevents physical fragmentation. Each data block is stored with an associated header. The header lists, among other information, whether or not the data was compressed and with what algorithm. A list of the headers for all data the arena contains is replicated at the end of each arena. When an arena is filled, it is sealed — never to be modified again.

An on-disk hash table is used as the fingerprint index, and it is stored separately from the log. Hash buckets, which are used to resolve collisions, occupy an entire disk block, with any excess fingerprints written to subsequent blocks. Thus, every index query requires at least one disk seek, but often only one seek.

Separating the index from the block store allows Venti to maintain flexible storage policies. The block store is kept on a RAID, providing fault tolerance through parity.

Venti is not a full backup solution—it is merely a backend block store that can be used as one component of a complete system. Mappings from files to their logical chunks must be maintained externally. We observe this pattern in many deduplication systems. A storage backend is optimized for expected access patterns, and overlaid with client file system structures.

3.2 Minimizing data transfer

Persistent storage is not always the most constrained system resource. This subsection studies the application of deduplication techniques to data written over the network, as opposed to a rotating disk. In this scenario, systems may choose to trade CPU and memory for bandwidth savings.

3.2.1 Case Study: LBFS

The low bandwidth file system (LBFS) [10] is a network file system. Clients cache some data on their local machines, but the authoritative versions of the system's data are kept on a centralized server.

LBFS minimizes network transmission at all costs. LBFS prioritizes deduplication ratio and relies heavily on client-side caching in order to achieve these goals.

LBFS was the first system to propose variable-sized chunking, although they did not use that name at the time. In the paper, the authors referred to variable-sized chunking as the **sliding window method** (SW), which is shown in Algorithm 1. SW has two parameters: window size w, and target pattern size t. SW computes a fingerprint over

System	**Deduplication Location**	**Chunking**	**Index**	**Keywords**
Venti [11]	destination	N/A (variable)	on-disk hash table	write-once policy, append-only arena, data compression
Deep Store [15]	*destination*	variable	local hash structure, distributed hash table	rich metadata, delta encoding, compression
Hydrastor [4]	destination	variable (Rabin)	distributed hash table	resiliency classes, erasure codes, continuous operation
LBFS [10]	hybrid	variable (Rabin)	legacy DB	content defined chunking, modified NFS, resource trade-off
Data Domain [16]	destination	variable (Rabin)	tiered: Bloom filter, locality preserving cache, legacy DB	locality-preserving cache, stream informed segment layout, disk bottleneck
Sparse Indexing [8]	destination (hybrid)	variable (two thresholds, two divisors)	in-memory, sampled	sampling, sparse index, chunk locality
PRUNE [9]	destination	variable (INC-K)	partitioned index (tablets)	INC-K, tablet, partitioned index
HydraFS [13]	destination	variable (Rabin)	distributed hash table	familiar file system API over Hydrastor backend
SIS (Windows 2000) [2]	source	N/A (whole-file)	database	file links with copy semantics, copy-on-close

Table 1: A categorization of example deduplication file systems.

all overlapping w-width byte ranges (hence, the name sliding window: the first window spans from bytes [0, w], the next window from bytes [1, w+1], and so on). When the value of the fingerprint's t low-order bits are equal to 0, SW defines a *break point* (i.e., a chunk boundary) at the last byte in w. SW then shifts the window right by w bytes, and repeats the process until reaching the end of the file.

Algorithm 1 - Sliding window method

```
1:  param NUMERIC $w, t$
2:  param FILE $f$
3:  INT $i \leftarrow 0$
4:  while $(i+w) \leq |f|$ do
5:      $t \leftarrow$ FINGERPRINT($f[i, \ldots, i+w]$)
6:      if $t = 0$ then
7:          define chunk boundary at $f[i+w]$
8:          $i \leftarrow i+w$
9:      else
10:         $i \leftarrow i+1$
11:     end if
12: end while
```

A negative binomial distribution allowing r failures has a mean of $(pr)/(1-p)$, so the sliding window method yields an expected chunk size of $2^t - 1 + w$. LBFS selects $t = 13, w = 48$, for an expected chunk size of $\approx$ 8K. Thus, the calculation of a single chunk boundary requires $\approx$ 8K individual fingerprint calculations.

Fortunately, Rabin fingerprints [12] are a reasonably efficient choice for chunking calculations because their calculation is incremental. Large parts of a Rabin fingerprint computation for one window can be reused in the computation for the next overlapping window. Note that Rabin fingerprinting is used to identify chunk boundaries, not to uniquely identify block contents. Deduplication requires that each chunk be separately fingerprinted with a cryptographically strong hash, such as `SHA-1`, in order to prevent collisions.

Variable-sized chunking presents an elegant solution to the problem of data insertion. Recall that fixed-size chunks are defined by byte offsets within a file. A file f', produced by adding a single byte at the front of the file f, might have no fixed-size blocks in common with f because every block boundary is shifted by one. This is called the boundary-shifting problem. In LBFS, this would mean that the entire file would need to be retransmitted over the network, despite the fact that the data exists in its entirety at the server (the file just has one new byte at the front). If the system used variable-sized chunking (which it does!), it is likely that only one block would need to be sent.

To see why, consider the three cases shown in Figure 1 where (1) data is inserted into the middle of a chunk, (2) data is inserted that includes (or produces) a new chunk boundary, and (3) data is inserted into a window that previously contained a chunk boundary. In (1), only the single block containing the new data must be re-chunked. In both (2) and (3), the containing block up to any new boundary is fingerprinted, followed by the successor, until an existing chunk boundary is encountered. LBFS only transmits the new chunks across the network.

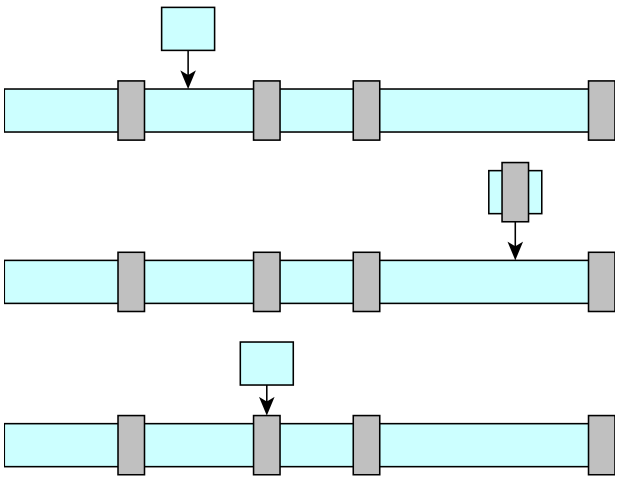
Figure 1: Three cases of a small data insertion into an existing file with variable sized chunks. Windows that represent chunk boundaries are shown in gray. In the first case, the insertion does not introduce any new chunks. A new fingerprint must be taken for the containing chunk, but no boundaries shift. In the second case, the insertion causes an existing chunk to be divided into two. The final example eliminates an existing chunk boundary, causing two previously existing chunks to be merged. LBFS only transmits new chunks across the network, reducing bandwidth.

Two drawbacks to variable-sized chunking are an increased computational burden (calculating the Rabin fingerprints is not free) and a vulnerability bad behavior under certain inputs. On one extreme, there are inputs that would contain a chunk boundary every w bytes. It would be less expensive to transmit the raw data than to transmit the messages that identify these very tiny duplicate blocks. On the other extreme, some data streams could produce no internal chunk boundaries. Notably, a sequence of all 0's has this property: since every window is identical, there would be no boundaries in any of them.

To combat this, LBFS defines a minimum chunk size of 2KiB and a maximum of 64KiB to handle these degenerate cases: the sliding window algorithm starts at an offset of 2KiB instead of 0, and after 64K, an artificial boundary is inserted whether a match is found or not.

4 Improving Dedup Systems

The two primary factors that influence a system's deduplication efficiency are the chunking method and the index management. Although related—the average chunk size affects the index size—the index and chunking algorithm can be optimized separately.

This section presents two case studies that improve upon the basic strategies we've discussed so far. In Subsection 4.1, we introduce the *disk-bottleneck problem* and we discuss the various techniques that the Data Domain deduplication system [16] introduces to manage it. In subsection 4.2, we briefly discuss optimizations to the sliding window algorithm and Rabin fingerprinting.

4.1 The Disk Bottleneck: Efficient Indexing

Each unique chunk that we add to our system requires a corresponding ⟨`hash, location`⟩ entry be inserted to the system's fingerprint index. Even for modest data sets (and we'll use modest as a relative term), the size of the fingerprint index can exceed the size the system's RAM. Let's consider a chunk store with 20TB of unique data: if the fingerprint index *only* stores each chunk's `SHA-1` hash (20B) an average chunk size of 4KB would result in a 100GB index!

In general, caching is the technique we use to improve our performance whenever our data structures exceed the bounds of our memory. Our standard caching techniques typically rely on good locality to be effective (spatial and/or temporal locality). Unfortunately, we've seen that `SHA-1` fingerprints are independently and uniformly distributed, and as a result, fingerprint index queries have no locality of reference. If we naively apply standard caching techniques to our fingerprint index, they will perform poorly, and each lookup will still require an expensive disk seek. This problem is referred to as the disk-index bottleneck problem, and it is the problem that the Data Domain deduplication system [16] sets out to solve.

4.1.1 Case Study: Data Domain

The Data Domain deduplication system introduces a three-tiered system to efficiently manage its fingerprint index queries (the tiers are described below). The overall system goals are to: (1) minimize the number of on-disk index lookups, and (2) make sure that any work that is done when an on-disk lookup *is* necessary also helps with satisfying future queries.

Figure 3 shows the possible paths of a fingerprint lookup through the three tiers, which are described in more detail below.

Tier 1: summary vector. The first level is the **summary vector**, a simple Bloom filter [1] that stores the set of all chunk fingerprints in the system. Looking up a fingerprint in the summary vector isn't perfect, since false positives are possible. But, we do know that any time the Bloom filter says that a fingerprint is *absent*, we are guaranteed that we don't need to look up that fingerprint on our disk. If our Bloom filter returns *present*, we advance to level two.

Tier 2: locality preserving cache. The second tier is the **locality preserving cache** (LPC) which is an in-memory hash table. This hashtable contains full fingerprint index entries, but the caching/eviction policies that move fingerprints into and out of the LPC do so in large groups. If our lookup hits in the LPC, we are done. If our lookup misses in the LPC, the lookup proceeds to the final tier, the actual on-disk index.

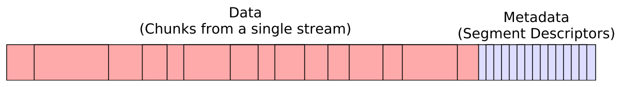

Figure 2: Container abstraction. Data from a single stream is chunked and appended to a until the container is filled. Chunk descriptors are stored contiguously at the end of a container, in order to be efficiently read into to the locality preserving cache. Containers are self-describing.

Tier 3: stream-informed segments. One key observation is that the actual backup data stream has good locality over time. For some intuition, consider a nightly backup of the contents of your laptop. Many files continue to exist between backups (unless the file is new, it was present in the previous backup). And the files that do change will typically have many of the same contents in consecutive versions, with a few new chunks that have the modifications.

Thus, Data Domain's on-disk format divides the data into *containers* that store a group of chunks from the same stream (along with their fingerprints). Data Domain refers to this as the **stream informed segment layout** (SISL). The on-disk container layout is depicted in Figure 2. The metadata for each chunk is kept at the head of the container. The metadata (called segment descriptors) are the fingerprint/chunk pointers mappings that make up the fingerprint index.

When a miss occurs in the LPC, the system must go to the disk to find the fingerprint's container. But instead of just reading that one fingerprint, the system prefetches the fingerprints *for all members of that container* into the LPC. In this way, the disk seek that was done to fetch one entry is used to preload many candidates that will hopefully create cache hits on future lookups.

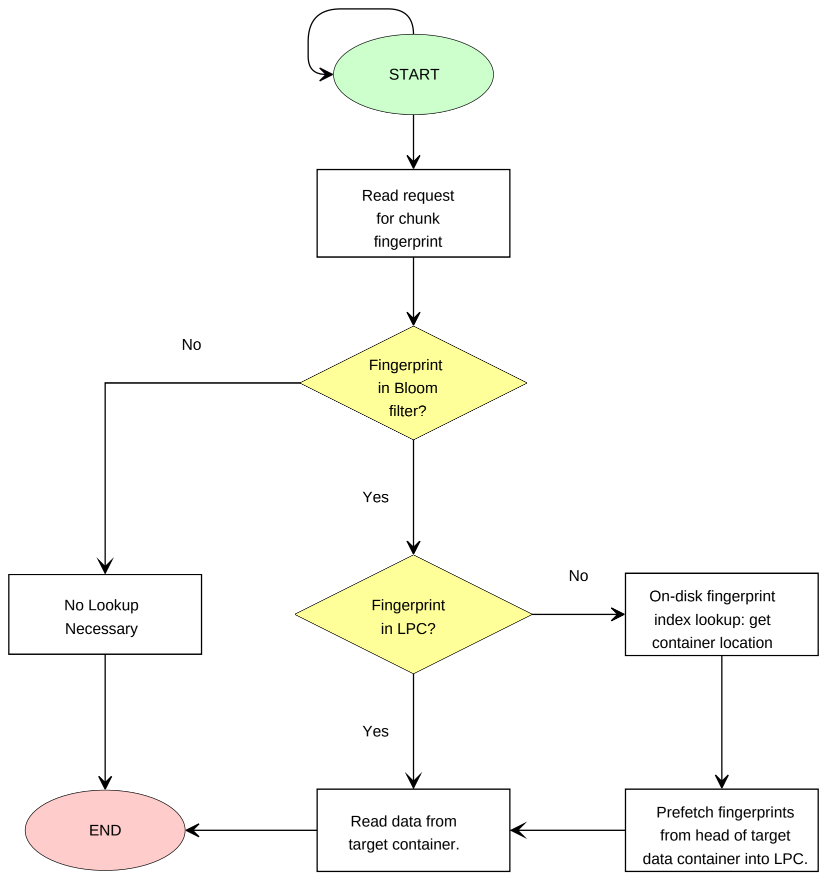
START
Read request for chunk fingerprint
Fingerprint in Bloom filter?
No
Yes
No Lookup Necessary
Fingerprint in LPC?
No
Yes
On-disk fingerprint index lookup: get container location
END
Read data from target container.
Prefetch fingerprints from head of target data container into LPC.

Figure 3: The Data Domain file system mitigates the disk bottleneck with its three-tiered architecture. The first level is a Bloom filter, which provides efficient in-memory approximate membership queries. A positive Bloom filter hit triggers a lookup in the in-memory locality preserving cache. A miss in the locality preserving cache finally results in an index lookup. Instead of reading just the requested fingerprint, all chunk descriptors from the containing segment are read into the locality preserving cache, evicting the oldest set of segment chunk descriptors.

The summary vector, LPC, and SISL work together to reduce aggregate disk seeks for backup workloads. They are effective as a unit because each component fills a different role, the relative importance of which changes over time. The summary vector filters queries for new data blocks, which are common during an initial backup; however, fewer new blocks are seen thereafter. LPC prefetching loads the previously encountered block fingerprints from a single container into the cache. The LPC is most effective for data that has been previously stored and rarely changes. The LPC and SISL rely on chunk locality within the backup stream. Chunk locality is a reasonable assumption for backup workloads where files change incrementally. However, for frequently changing files, chunk locality LPC prefetching populates the cache with unrelated chunk fingerprints.

Together, the three techniques removed up to 99% of disk accesses for index lookups in two real deduplication workloads. That is an incredible result for a real system, and it is one of the reasons why this paper won the USENIX test of time award for being so influential, even today.

4.2 Chunking

Since chunking is the division of a data stream into one or more segments, and the segment is the granularity at which data is stored and redundant data detected, chunking, perhaps more than any other system component, affects the system's deduplication ratio.

There are two dimensions in which to evaluate a chunking algorithm: runtime performance and chunk quality. Section §4.2.1 describes the two thresholds, two divisors (TTTD) algorithm [5]. TTTD reduces chunk size variance, so that most chunks are about the same size. As a result, it lowers the overhead introduced by small modifications.

The comparison of the runtime performances of two chunking methods is difficult. The speed of identifying boundaries can be directly compared, but a system that identifies more shared data will save disk I/O of duplicate writes. TTTD provides more uniformly sized chunks, which lowers the overall overhead of data insertions, but does so at the cost of additional computations.

4.2.1 Case Study: Two Thresholds Two Divisors

The two thresholds, two divisors algorithm (TTTD) is a generalization of the standard sliding window algorithm (SW) introduced by LBFS [10] and described in Section §3.2.1.

SW has two explicit parameters: window size w, target t, in addition to divisor D. LBFS observed that for some inputs, SW produces very small or very large chunks. LBFS enhanced the algorithm with minimum and maximum chunk size thresholds ($T_{\texttt{min}}$ and $T_{\texttt{max}}$) to bound the chunk size range. TTTD again extends the algorithm and adds a second divisor D'.

For window W and target t, we say a *fingerprint match* against divisor D occurs if

$$\textsc{FingerPrint}(W) \mod D \equiv t \tag{2}$$

The TTTD algorithm slides a fixed-length window W across the data stream byte by byte, starting at $T_{\texttt{min}}$. TTTD checks for a fingerprint match against D and D' at each position. A fingerprint match against D immediately halts execution and defines a chunk boundary. A fingerprint match against D' is saved for future use. TTTD continues until reaching $T_{\texttt{max}}$. The most recent fingerprint match against D', if any, defines the chunk boundary. Otherwise, a boundary is inserted at $T_{\texttt{max}}$.

Adding a second divisor reduces chunk size variance: fewer boundaries are defined by $T_{\texttt{max}}$. Chunks defined by $T_{\texttt{max}}$ are fixed-size chunks, and therefore suffer from the boundary shifting problem. TTTD reduces the overhead of data that is replicated not because it is redundant, but because a chunk modification disassembled its previous chunk. An appropriate choice of D' reduces the overhead of the second check.

5 Conclusion

This survey presented a broad overview of the deduplication system design space. It talked about key parameters that are common to many deduplication systems, and it used several case studies as a way to present the common problems that deduplication systems must solve. There are many other areas of interest that this survey did not cover but that deduplication researchers continue to explore. Restore throughput, sketching, data migration, and even alternative strategies for removing redundancy (e.g., generalized deduplication or delta compression) are all active areas of research that are of interest to academics and industry alike.

References

[1] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, jul 1970. 7

[2] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in windows 2000. In *WSS*, pages 13–24, 2000. 2, 5

[3] O. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: making backup cheap and easy. In *OSDI*, pages 285–298, 2002. 2

[4] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. HYDRAstor: a scalable secondary storage. In *FAST*, pages 197–210, 2009. 5

[5] K. Eshghi and H. K. Tang. A framework for analyzing and improving content-based chunking algorithms. HP Laboratories Palo Alto HPL-2005-30R1, 2005. 8

[6] L. Freeman. Looking beyond the hype: Evaluating data deduplication solutions. Netapp White Paper, September 2007. 2

[7] D. Geer. Reducing the storage burden via data deduplication. *Computer*, 41(12):15–17, Dec. 2008. 2

[8] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *FAST*, pages 111–123, 2009. 5

[9] J. Min, D. Yoon, and Y. Won. Efficient deduplication techniques for modern backup operation. *TC*, 60(6):824–840, june 2011. 2, 3, 5

[10] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP*, pages 174–187, 2001. 2, 3, 4, 5, 9

[11] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *FAST*, pages 89–101, 2002. 2, 4, 5

[12] M. O. Rabin. Fingerprinting by random polynomials. Harvard Aiken Computational Laboratory TR-15-81, 1981. 3, 6

[13] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: A high-throughput file system for the HYDRAstor content-addressable storage system. In *FAST*, pages 225–238, 2010. 5

[14] Y. Won, J. Ban, J. Min, J. Hur, S. Oh, and J. Lee. Efficient index lookup for de-duplication backup system. In *MASCOTS*, pages 1–3, sept. 2008. 3

[15] L. You, K. Pollack, and D. D. E. Long. Deep Store: An archival storage system architecture. In *IDCE*, pages 804–815, 2005. 2, 5

[16] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST*, pages 269–282, 2008. 3, 5, 7