# Avoiding the Disk Bottleneck in the Data Domain Deduplication File System

Slides about the article:

http://webcourse.cs.technion.ac.il/236834/Spring2017/ho/WCFiles/FAST08-DataDomain.pdf

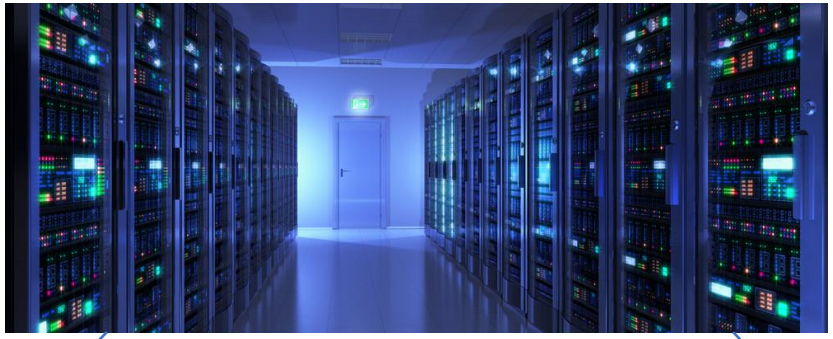
Slides made by Koral chapnik and Yael Amitay

# Introduction

- Disk-based deduplication storage has emerged as the new-generation storage system for enterprise data protection to replace tape libraries.
- Deduplication removes redundant data segments to compress data into a highly compact form and makes it economical to store backups on disk instead of tape.
- This paper describes techniques employed in the production Data Domain deduplication file system to relieve the disk bottleneck.

# Backups

## Data Center



## Weekly Full Backup

Backup **all the data** on the primary storage systems to secondary storage systems.

## Daily Incremental Backup

copy only **the data which has changed** since the last backup

# Secondary Storage

**Requirements:**

- Low cost - so that storing backups and moving copies offsite does not end up costing significantly more than storing the primary data
- High performance - so that backups can complete in a timely fashion.

- **The traditional solution has been to use tape libraries as secondary storage devices and to transfer physical tapes for disaster recovery.**

# What Is A tape And How It Works

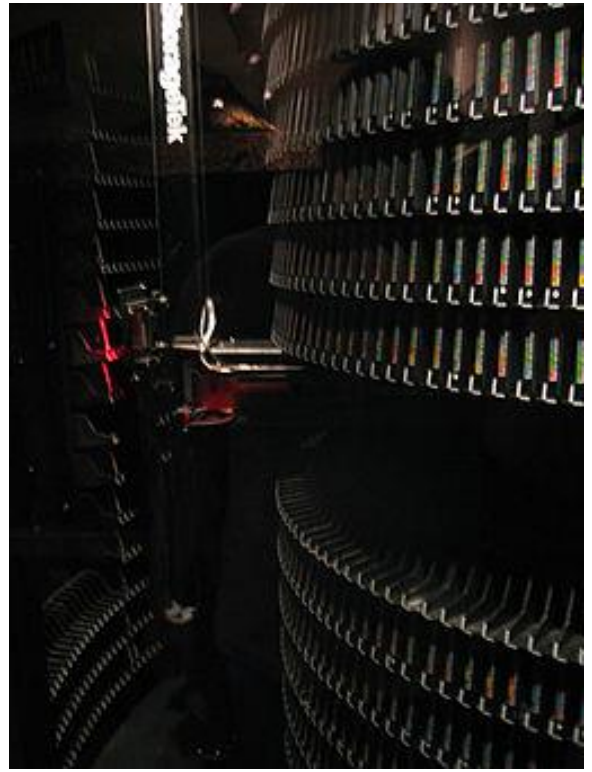
Tape library

contains

Tape drive


reads and
writes data on a

Magnetic tape



# Hard Disk Drive

- A data storage device that uses magnetic storage to store and retrieve digital information using one or more rigid rapidly rotating disks coated with magnetic material.
- The platters are paired with magnetic heads, usually arranged on a moving actuator arm, which read and write data to the platter surfaces.

# Tape VS Disk

## Tape

- Provides sequential access storage
- Tape media generally has a favorable unit cost and a long archival stability.
- a tape drive must physically wind tape between reels to read any one particular piece of data.

## Hard disk

- Provides direct access storage
- A disk drive is more expensive than tape
- A disk drive can move to any position on the disk in a few milliseconds

# Deduplication And Hard Disk Drive

- Deduplication is compressing data by removing duplicate data across files and often across all the data in a storage system
- Deduplication reduces the footprint of versioned data
- Deduplication can make the costs of storage on disk and tape comparable and make replicating data over a WAN to a remote site for disaster recovery practical.

# Challenges

- The key performance challenge is finding duplicate segments

- Given a segment size of 8 KB and a performance target of 100 MB/sec, a deduplication system must process approximately 12,000 segments per second.

# Solution – try #1

An **in-memory index** of all segment fingerprints

But…

the size of the index would limit system size and increase system cost!

# Solution – try #2

An **on-disk index** of segment fingerprints and a cache to accelerate segment index accesses.

But...

- since fingerprint values are random, there is no spatial locality in the segment index accesses.
- because the backup workload streams large data sets through the system, there is very little temporal locality.

# As We Discussed There Are Some Tradeoffs

- Variable vs. Fixed Length Segments
- Segment Size
- Performance-Capacity Balance
- Fingerprint vs. Byte Comparisons

# Deduplication Storage System Architecture – DDFS

At the highest level, DDFS does the following:

- Breaks a file into variable length segments in a content dependent manner
- Computes a fingerprint for each segment

It uses the fingerprints :

- To identify duplicate segments
- As part of a segment descriptor used to reference a segment

# DDFS


NFS
CIFS
VTL
File Services
Content Store
Segment
Store
Container manager

# DDFS - Write

it goes through one of the standard interfaces

**generic File Services layer**

manages the name space and file metadata

forwards write requests to

**Content Store**

manages the data content within a file

breaks a data stream into segments




# DDFS - Write

uses

**Segment Store**

does the actual work of deduplication

Packs and compresses data

Writes results into

Supported by

**Container Manager**

# DDFS - Read


drives the read operation through one of the standard interfaces and the File Services Layer

**Segment Store**

prefetches, decompresses, reads and caches data segments

from

**Container Manager**

**Content Store**

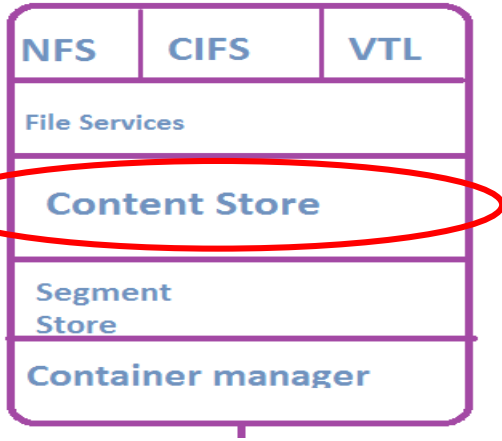
uses the references to deduplicated segments to deliver the data stream to

# Content Store

- Content Store implements byte-range writes and reads for deduplicated data objects
- An object is a linear sequence of client data bytes and has intrinsic and client settable attributes or metadata (for example,  a conventional file, a backup image)

# Content Store: write a range of bytes into an object

# 1. Anchoring




partitions the byte range into variable length segments in a content dependent manner

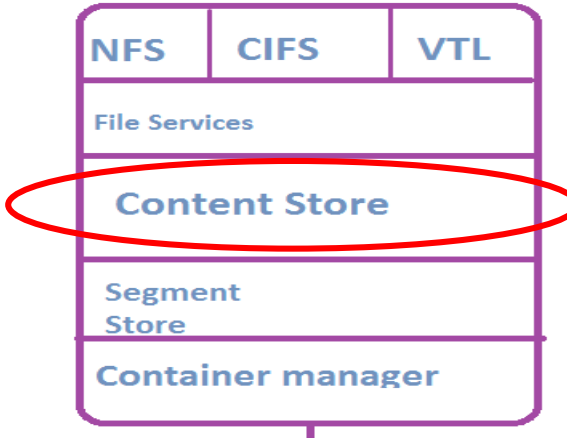
data

segment 1

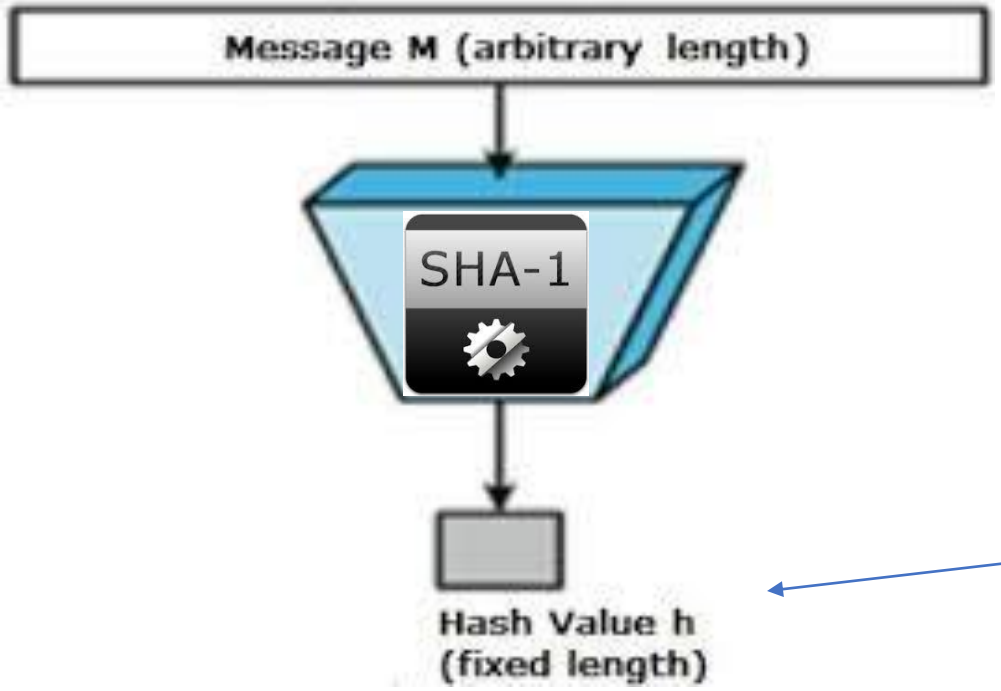
segment 2

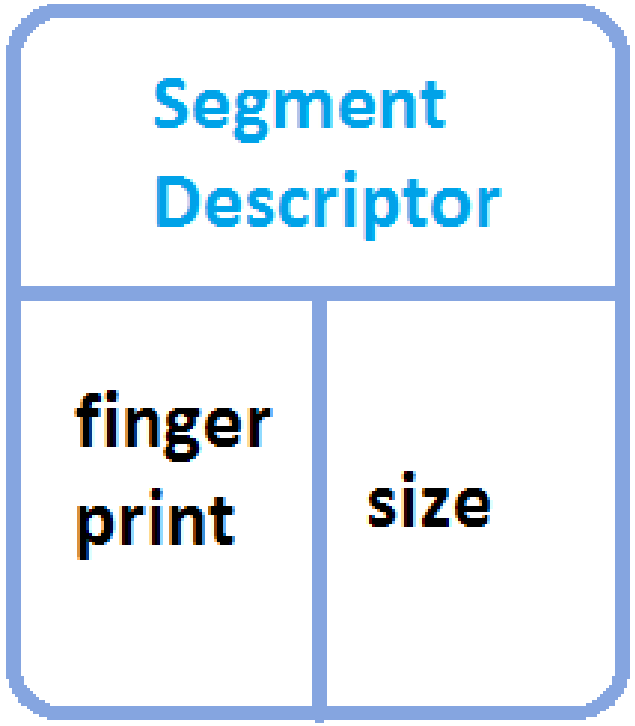
segment 3

# 2. Segment Fingerprinting

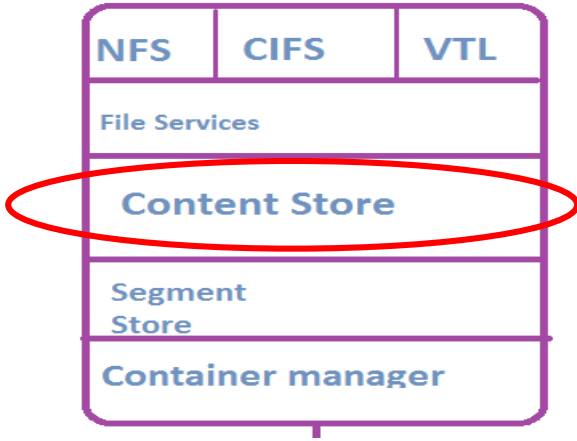



Computes the SHA-1 hash and generates the segment descriptor based on it



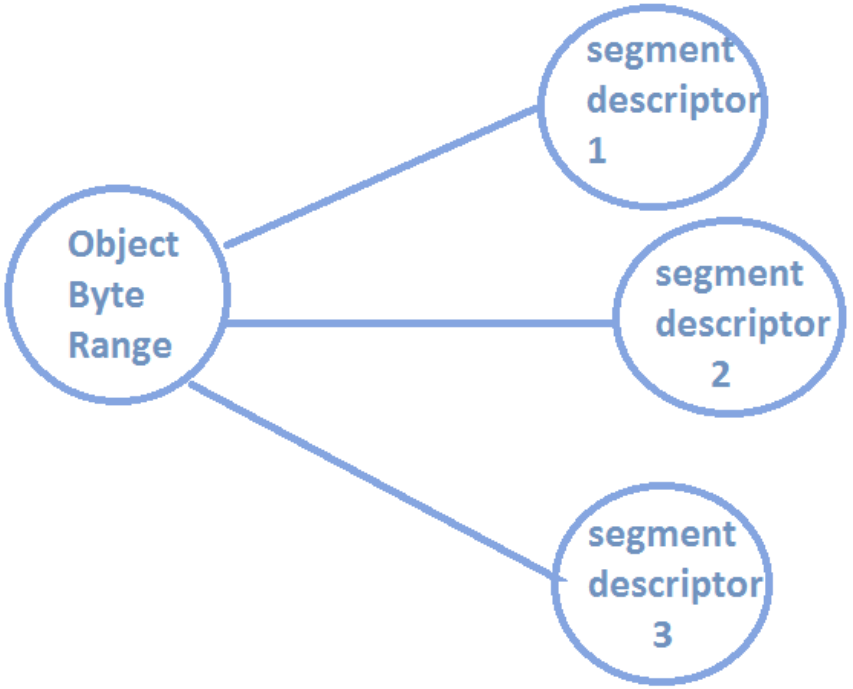



# 3. Segment Mapping



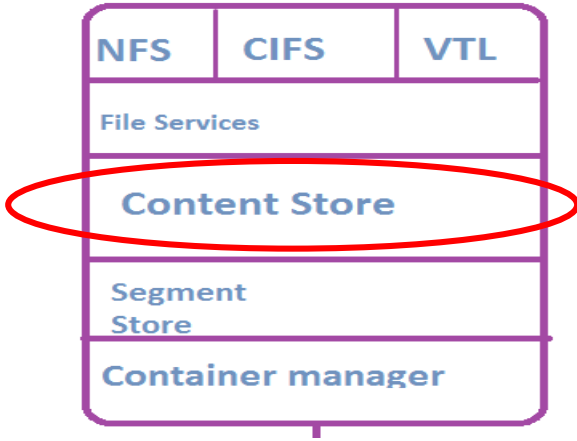

builds the tree of segments that records the mapping between object byte ranges and segment descriptors

The goal is to represent a data object using references to deduplicated segments.



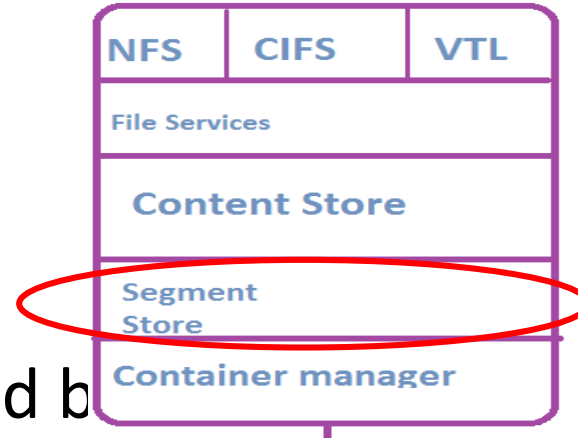
# Content Store: read a range of bytes in an object

# Reading A Range Of Bytes




1. Content store traverses the tree of segments to obtain the segment descriptors for the relevant segments
2. It fetches the segments from Segment Store
3. Returns the requested byte range to the client.

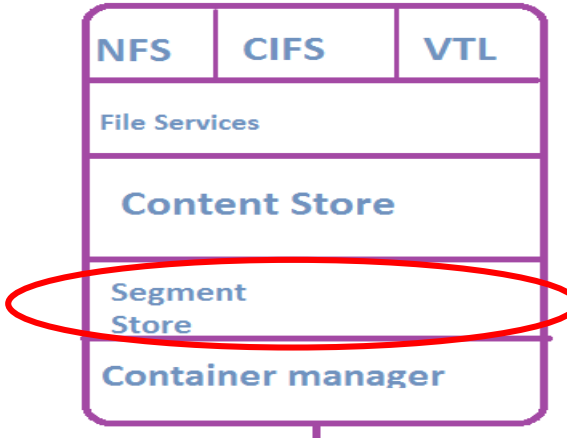
# Segment Store




- Segment Store is essentially a database of segments keyed b segment descriptors
- To support writes, it accepts segments with their segment descriptors and stores them.
- To support reads, it fetches segments designated by their segment descriptors.

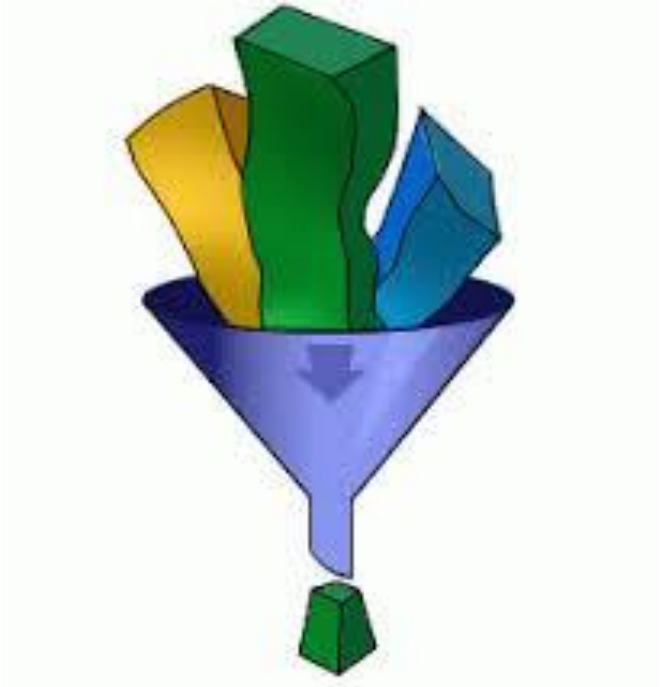
# Segment Store: write a data segment

# 1. Segment Filtering

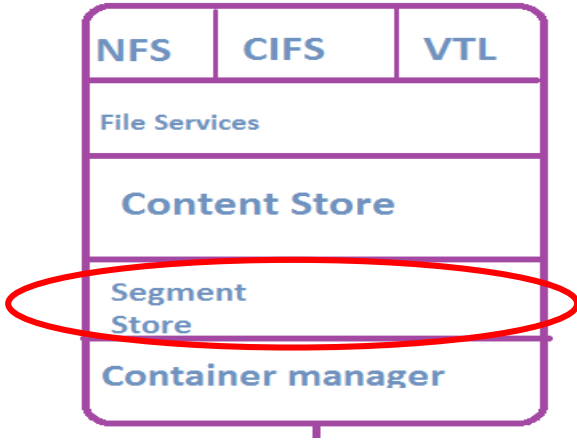



Determines if a segment is a duplicate.

This is the key operation to deduplicate segments and may trigger disk I/Os, (thus its overhead can significantly impact throughput performance. )

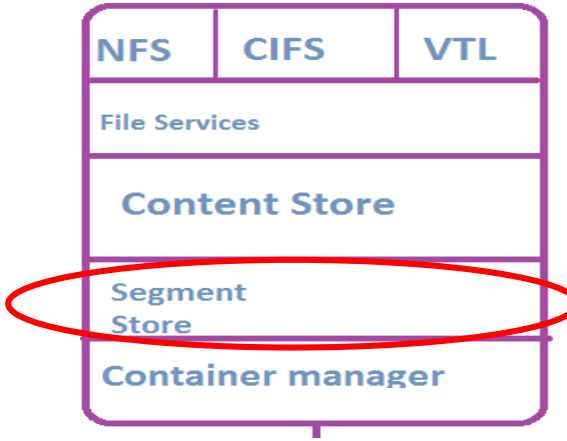


# 2. Container Packing



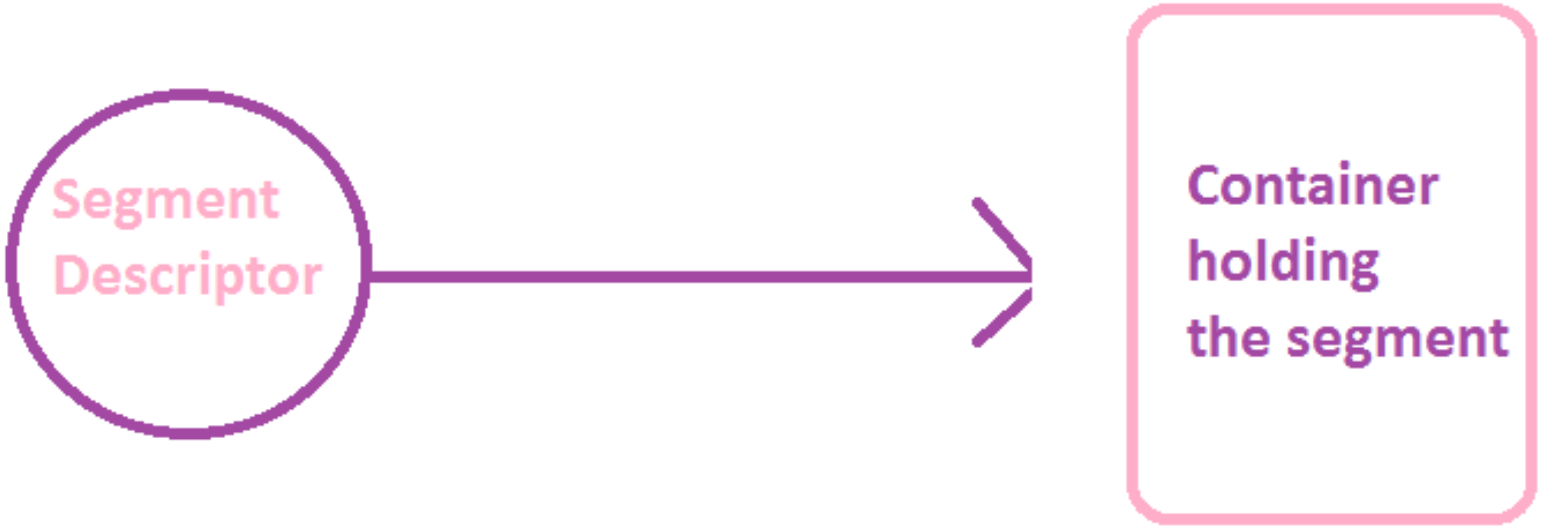

- Adds segments to be stored to a container which is the unit of storage in the system
- Compresses segment data
- A container, when fully packed, is appended to the Container Manager.

# 3. Segment Indexing



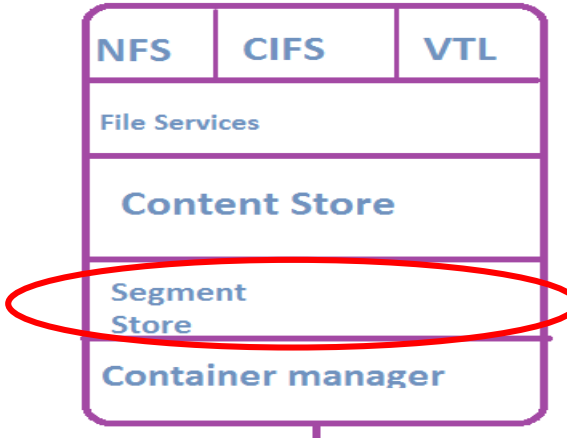

After the container has been appended to the Container Manager.

Segment Store updates the segment index that maps segment descriptors to the container holding the segment



# Segment Store: read a data segment

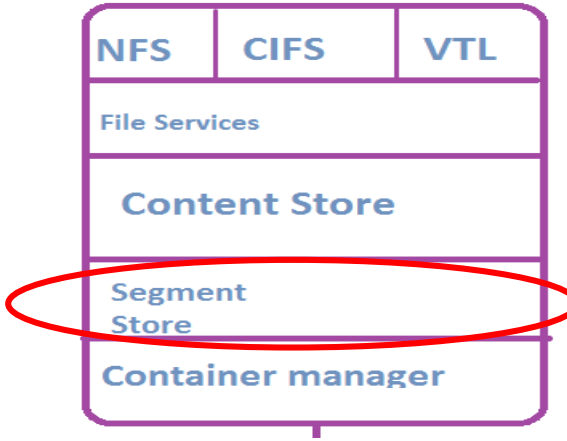
# 1. Segment Lookup



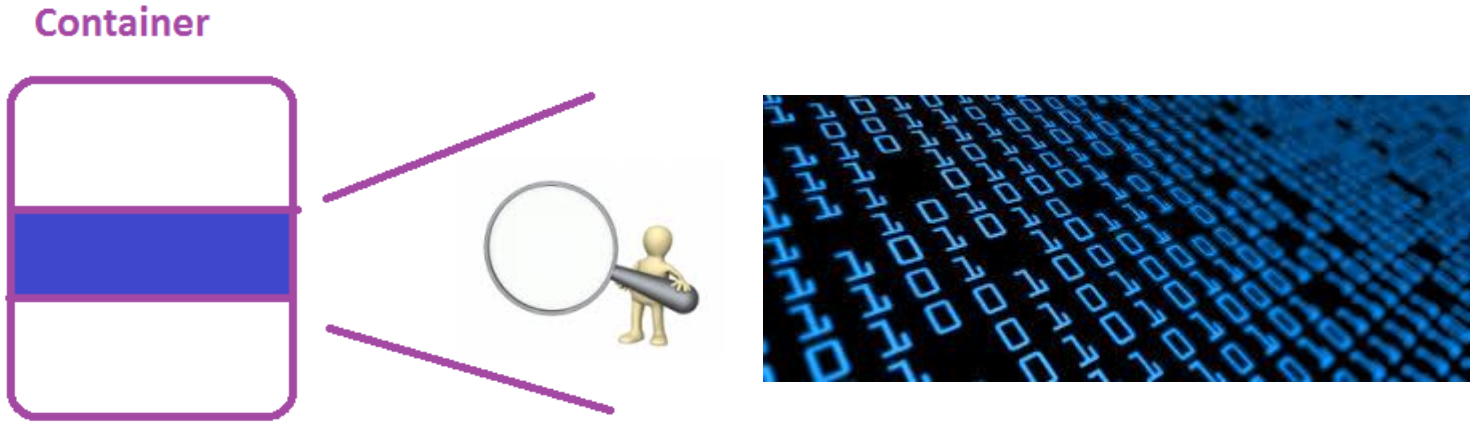

Finds the container storing the requested segment.

This operation may trigger disk I/Os to look in the on-disk index, thus it is throughput sensitive.

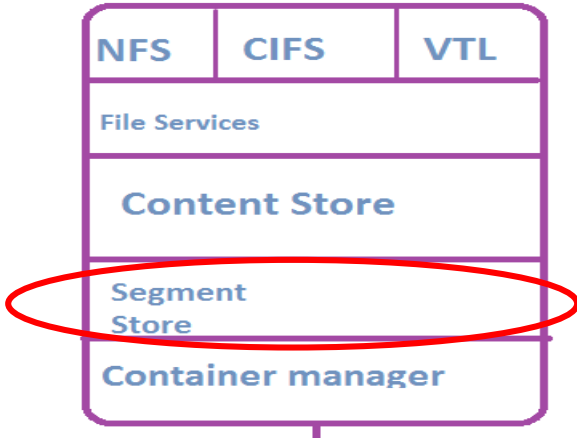
# 2. Container Retrieval




Reads the relevant portion of the indicated container by invoking the Container Manager.



# 3. Container Unpacking

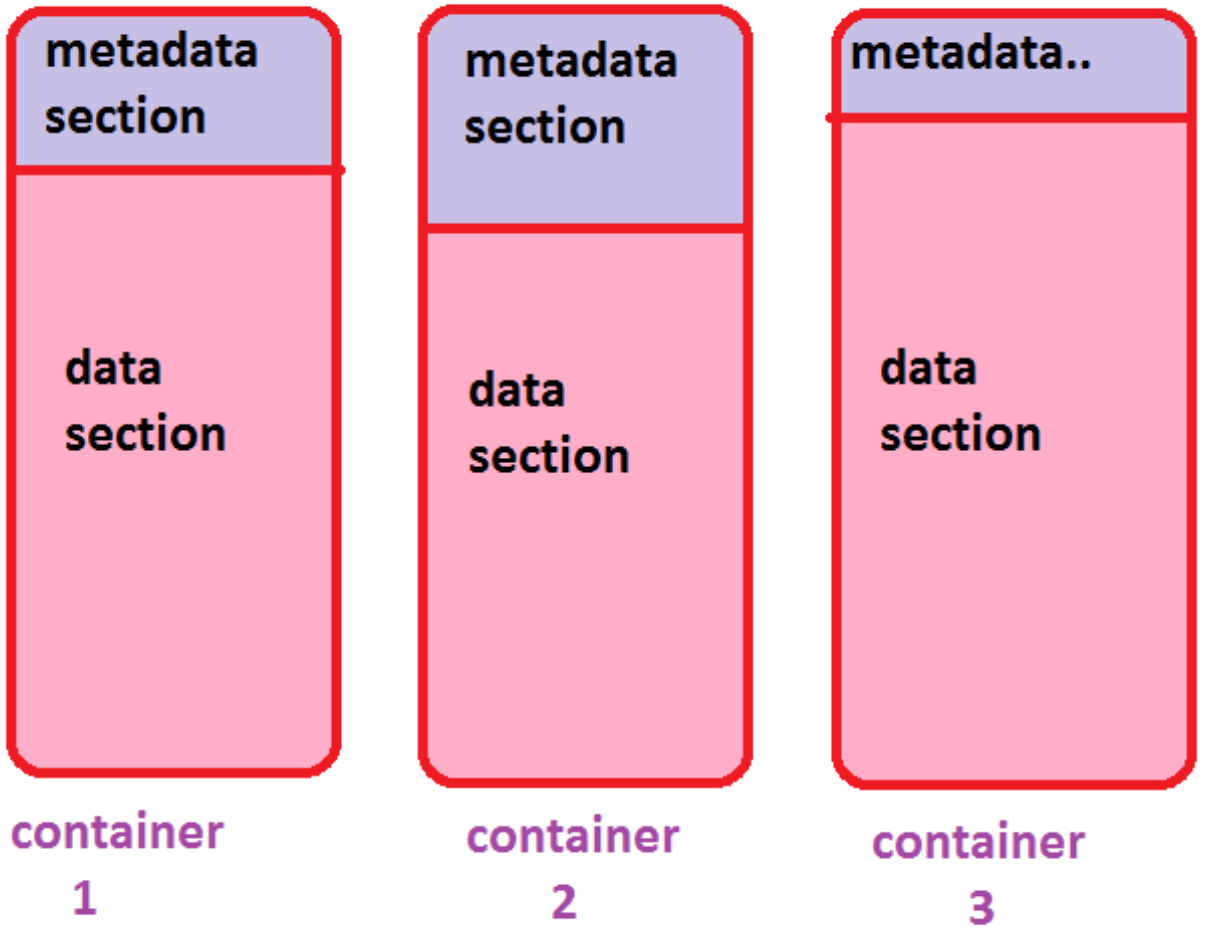



Decompresses the retrieved portion of the container and returns the requested data segment.

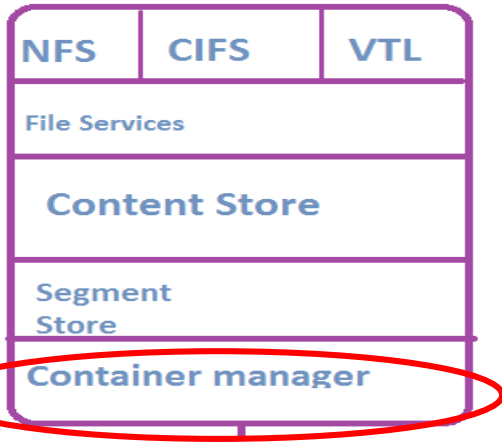
Decompressing

# Containers

- A metadata section includes the segment descriptors for the stored segments
- They are immutable
- Each container has a container ID which is unique over the life of the system

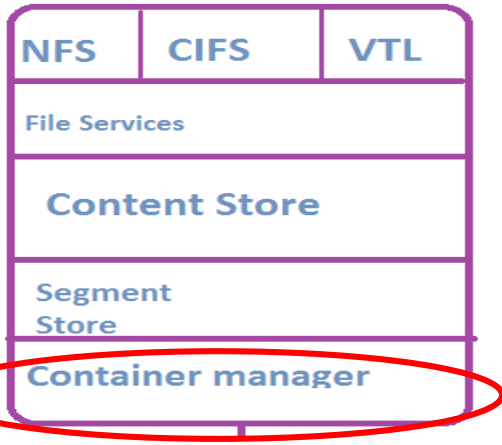


# Container Manager




- The Container Manager is responsible for allocating , deallocating, reading, writing and reliably storing containers.
- It supports reads of the metadata section or a portion of the data section, but it only supports appends of whole containers.
- If a container is not full but needs to be written to disk, it is padded out to its full size

# Container Manager




The container abstraction offers several benefits:

- The fixed container size makes container allocation and deallocation easy.
- The large granularity of a container write achieves high disk throughput utilization.

# Acceleration methods

By now we already know how  the deduplication works

The backup process needs to be fast that’s why we need the deduplication process to be fast.

Next we will discuss 3 methods to accelerate the deduplication process.

# Acceleration Method #1: Summary vector

**The problem**

When we want to check if the some new segment is already in the disk we compare the fingerprint of the new segment with the fingerprints of the segments we already have on the disk.

The fingerprints of the segments we have on the disk form an index.

If our index is large we can't fit it in the RAM.

So, when we want to check if some segment is already in the disk we'll have to look in the disk.

And that's BAD.

# Acceleration Method #1: Summary vector cont'

**The solution:** Bloom filter.

We have k hash functions: $0 \le i \le k, \quad hash_i : \quad fingerprints \to \{0,1,....,m\}$

We have an array of size M named Array.

Insert a new fingerprint to the index:

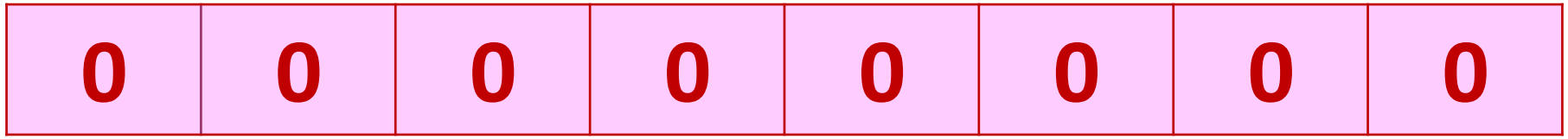
For each hash function: $Array[hash_i(\ fingerprint)] = 1$

Lookup for a fingerprint:

If for all hash function satisfied: $Array[hash_i(\ fingerprint)] = 1$

Then maybe the fingerprint is already in the index. Else the fingerprint is 100% not in the index and we need to add it.

# Acceleration Method #1: Summary vector example



# Acceleration Method #1: Summary vector example

**Insert new fingerprint f:**

$$hash_1(f) = 1, hash_2(f) = 4, hash_3(f) = 2$$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |

# Acceleration Method #1: Summary vector example

**Insert new fingerprint g:**

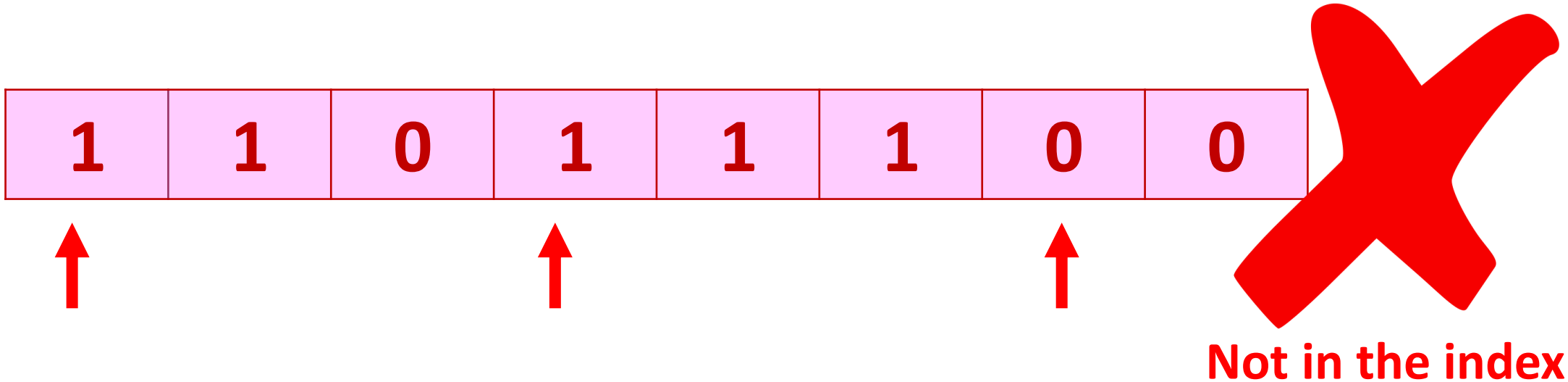
$$hash_1(g) = 2, hash_2(g) = 5, hash_3(g) = 6$$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |

# Acceleration Method #1: Summary vector example

**Lookup fingerprint z :**

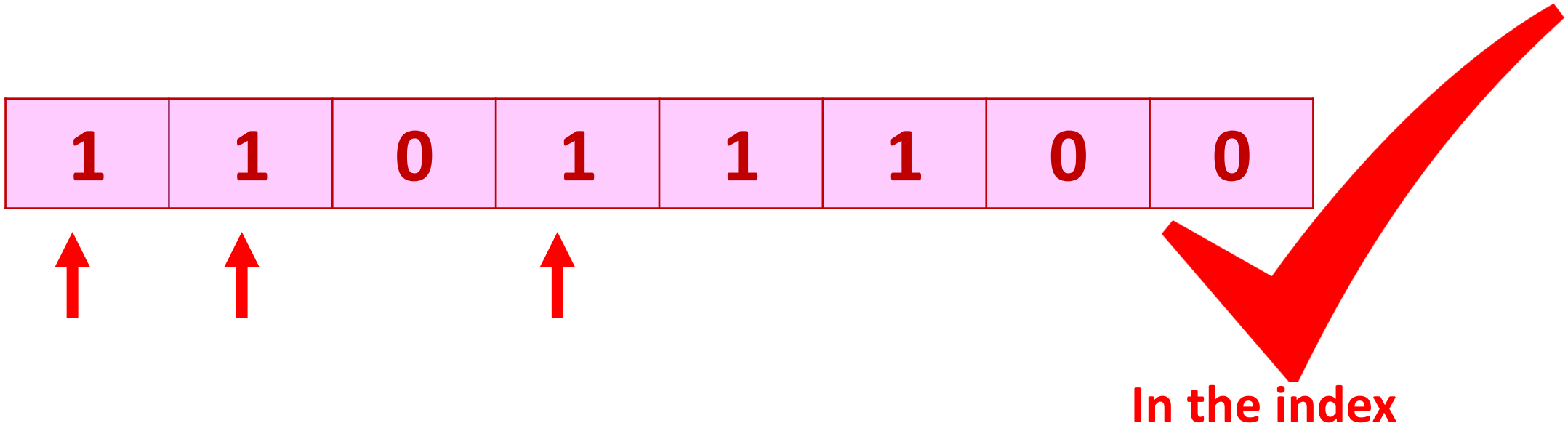
$$hash_1(z) = 1, hash_2(z) = 4, hash_3(z) = 7$$



# Acceleration Method #1: Summary vector example

**Lookup fingerprint f:**

$$hash_1(f) = 1, hash_2(f) = 4, hash_3(f) = 2$$

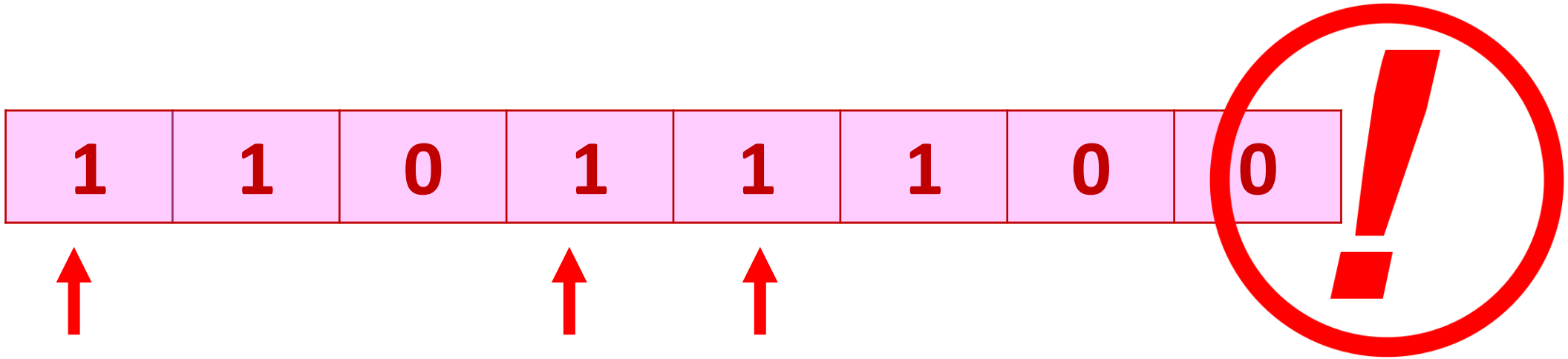


# Acceleration Method #1: Summary vector exam

**Lookup fingerprint t :**

But we didn't add t to the index!
We only add f and g!
That's why we need to check all the index if all the entry of the hash functions are set to 1

$$hash_1(t) = 1, hash_2(t) = 4, hash_3(t) = 5$$




In the index? No!

# Acceleration Method #1: Summary vector False Positive Probability

False Positive Probability – the probability that a fingerprint is not in the index but all the hash functions return 1.

**Note**: if we get 1 from all the hash function then we need to search the index. We are going to calculate the probability that all the hash functions return 1 for a new fingerprint!

# Acceleration Method #1: Summary vector False Positive Probability cont.

**The probability that some specific bit is zero is:**

Assume that we inserted n fingerprints.

The probability that the hash function will return the specific bit is $\frac{1}{m}$

We apply k hash function for each one of the n fingerprints. Meaning we apply k*n hash functions and we want each of them to avoid this specific bit:

$$\left(1-\frac{1}{m}\right)^{kn} = e^{-\frac{kn}{m}}$$

# Acceleration Method #1: Summary vector False Positive Probability cont.

Now we apply k hash function when we lookup for the **new** fingerprint. False positive means that each of the hash function will return bit that is set.

**So, the probability for false positive is**:

$$\left( \underbrace{1 - \underbrace{\left(1 - \frac{1}{m}\right)^{kn}}_{specific\,bit\,is\,zero}}_{the\,value\,of\,the\,bit\ from\,the\,hash\ func\,is\,1} \right)^{k} \approx \left(1 - e^{-\frac{kn}{m}}\right)^{k}$$

# Acceleration Method #1: Summary vector conclude

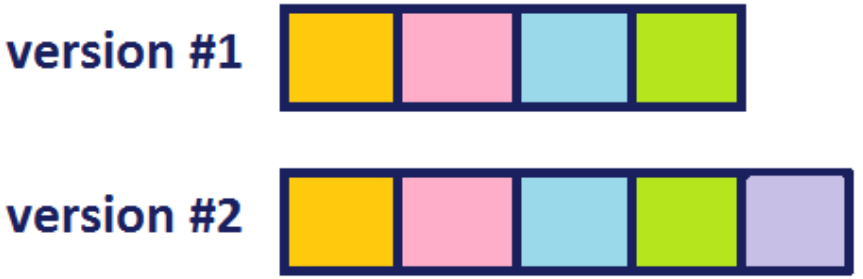
Instead of checking all the fingerprint index to see if some segment exist, we can check the small bloom filter to see if some segment exist.

# Acceleration Method #2: Stream-Informed Segment Layout(SISL)

*Reminder: we split our data to segments. The segments are stored in a container.*

**Main Observation**: In backup application segments tend to appear in the same order. (Or in very similar order.)

For example if we have 2 versions of some file:




The order of the segment remain the same.

We just added some new segment.

# Acceleration Method #2: Stream-Informed Segment Layout(SISL) cont.

So, when we backup some new segment X and we discover that he's a duplicate, there is a high probability that X's neighbors are also a duplicates!

We'll call this property ***"segment duplicate locality"***

# Acceleration Method #2: Stream-Informed Segment Layout(SISL) cont.

Before we'll move on, What is a Stream?

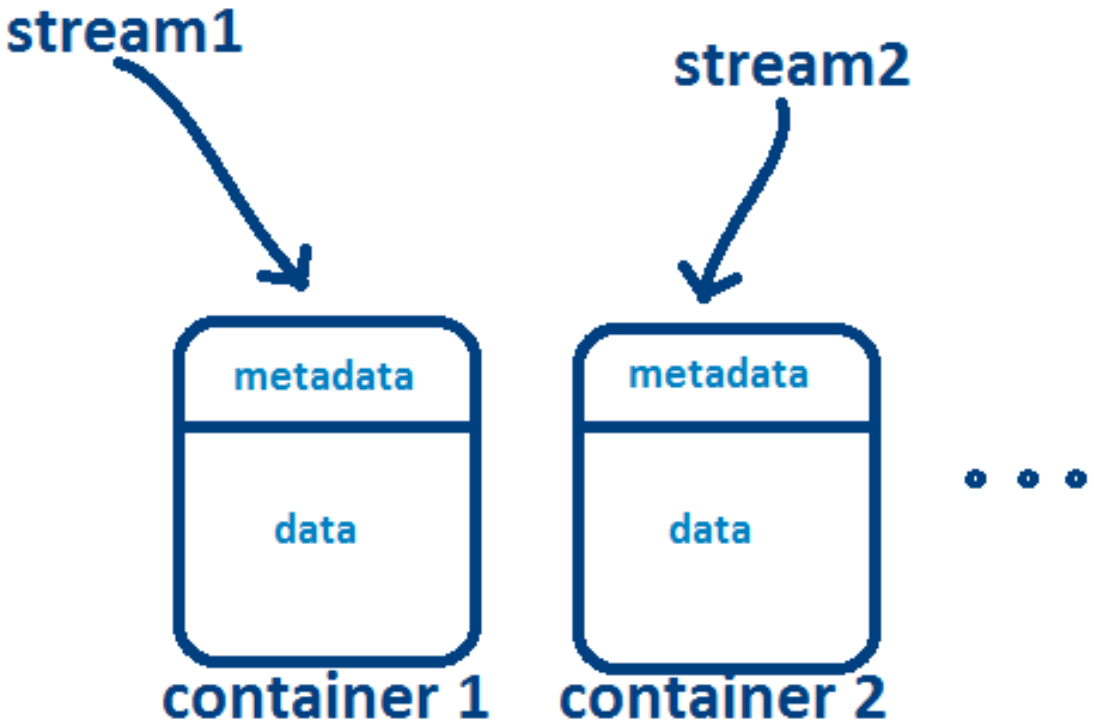
***Stream is a range of bytes that represent set of object that came from specific back up image.***

# Acceleration Method #2: Stream-Informed Segment Layout(SISL) cont.

- In order to preserve ***"segment duplicate locality"*** property we need to keep the locality of each stream.
- When we insert new data to the Content store and the Segment Store we need to separate the segments that came from different stream.
- Each Stream is inserted to a different container.
- We'll  insert 2 (or more) streams in parallel to 2 (or more) different container.
- The result is that segments of the same object store closely to each other. The same goes for their metadata(fingerprints and etc.)
- The metadata  is stored in a separate place but metadata of closely segments stored closely too.

# Acceleration Method #2: Stream-Informed Segment Layout(SISL) example

- We have a few open containers, one for each stream.
- We separate the metadata from the data.
- But metadata from the same stream will be together.
- We can also insert 2 (or more) streams in parallel.



# Acceleration Method #2: Stream-Informed Segment Layout(SISL) cont.

**Benefits:**

☺When multiple segments of the same data stream are written to a container together we need less disk I/Os operations to reconstruct the stream.

☺Descriptors and compressed data of adjacent new segments in the same stream are packed linearly in the metadata and data sections respectively in the same container. This packing captures duplicate locality for future streams resembling this stream, and enables Locality Preserved Caching to work effectively. ( We'll see it in a few slides).

# Acceleration Method #2: Stream-Informed Segment Layout(SISL) cont.

**Benefits continue:**

☺The metadata section is stored separately from the data section, and is generally much smaller than the data section.

The small granularity on container metadata section reads allows Locality Preserved Caching in a highly efficient manner: 1K segments can be cached using a single small disk I/O.

# Acceleration Method #3: Locality Preserved Caching

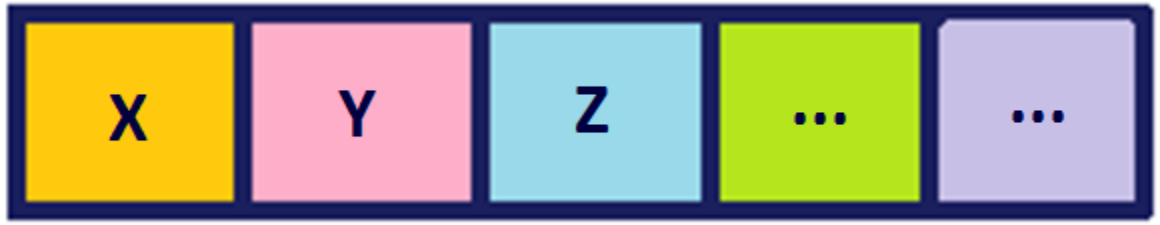
***This method accelerate the process of identifying duplicate segments.***

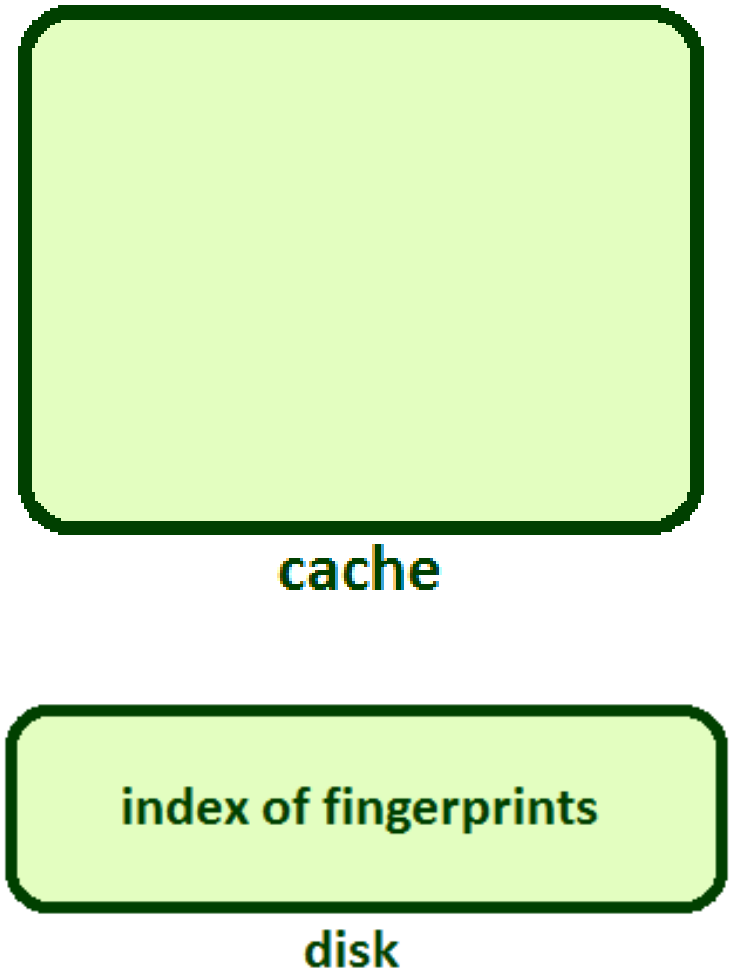
Traditional cache doesn't work good for caching fingerprints of duplicate detection. Why? If we load bunch of random fingerprints from the memory that not necessarily have any connection between them. Then caching them won't help us.

We know that if some segment X is duplicate then X neighbors are probably duplicate as well. So we would like to load their fingerprints with the fingerprint of X to the cache. (take advantage of ***"segment duplicate locality"***)

# Acceleration Method #3: Locality Preserved Caching cont.

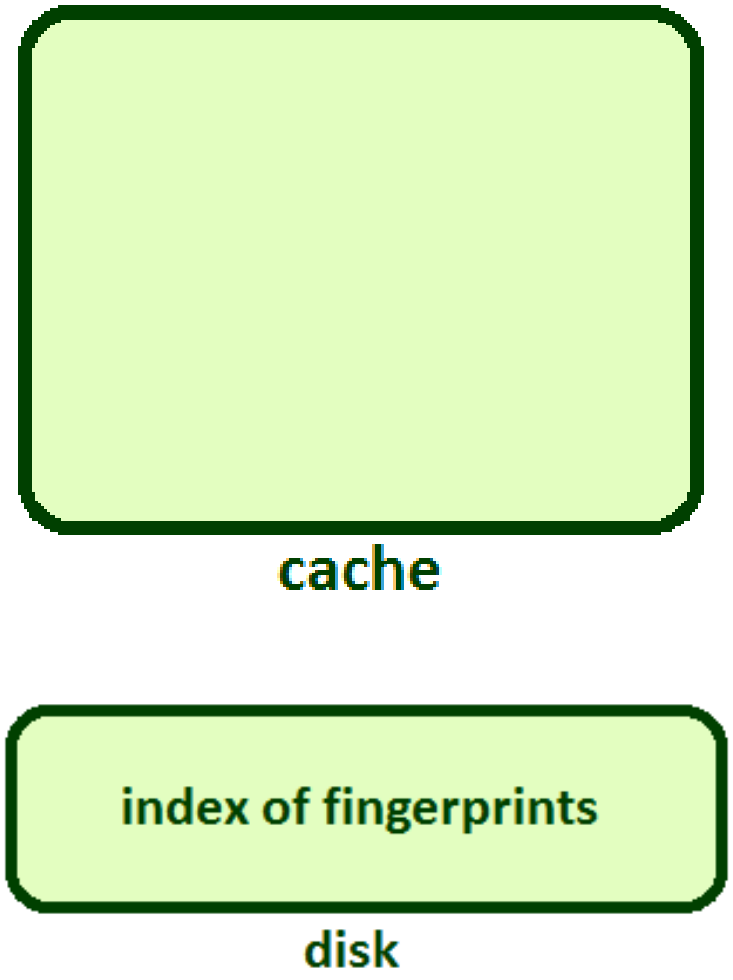
**Let's say we have a file composed of the following segments:**



# Acceleration Method #3: Locality Preserved Caching cont.

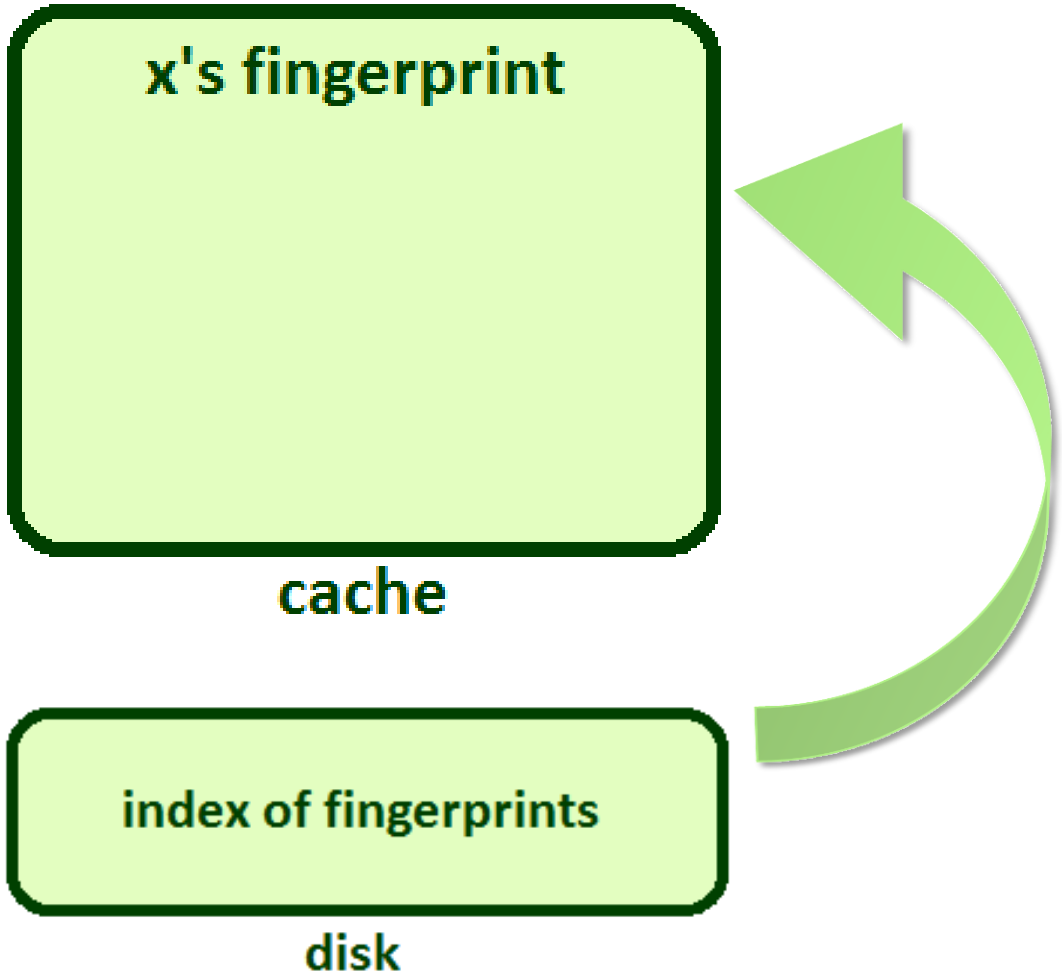
**Check if segment X is duplicate:**
Check if X's fingerprint is in the index. Meaning we need to go to the disk and look for X's fingerprint is in the index.
We also bring some other fingerprints with X's fingerprint to the cache. Who's fingerprint should we bring?



# Acceleration Method #3: Locality Preserved Caching cont.

**Check if segment X is duplicate**

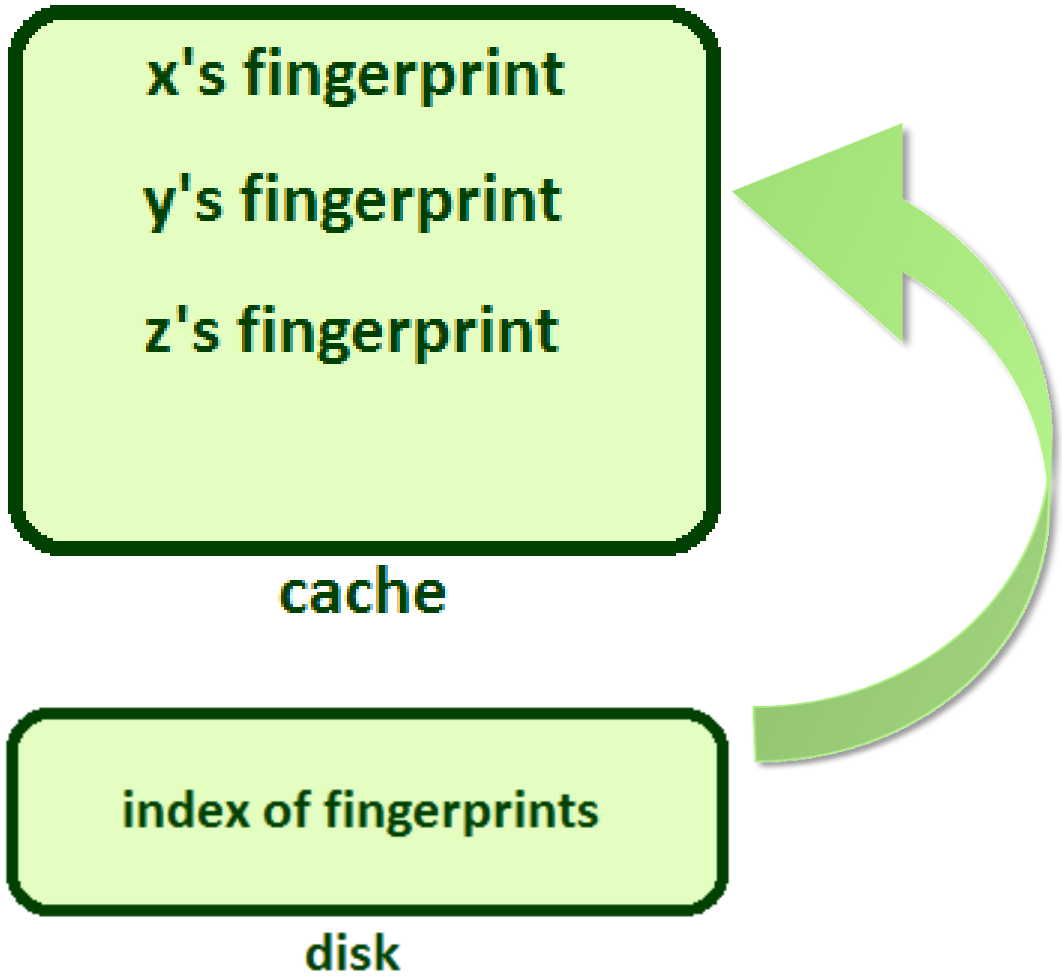
**Load X's fingerprint from disk and bring with him the fingerprints of his neighbors**



# Acceleration Method #3: Locality Preserved Caching cont.

Check if segment X is duplicate

Load X's fingerprint from disk and bring with him the fingerprints of his neighbors



# Acceleration Method #3: Locality Preserved Caching cont.

Now if we want to check if Y is a duplicate we can see in the cache that the Y's fingerprint exist and we don't have to check the disk! The same goes for Z and the other neighbors.

**So we manage to reduce the amount of disk I/O operations.**

# Acceleration Method #3: Locality Preserved Caching cont.

## But how does it work?

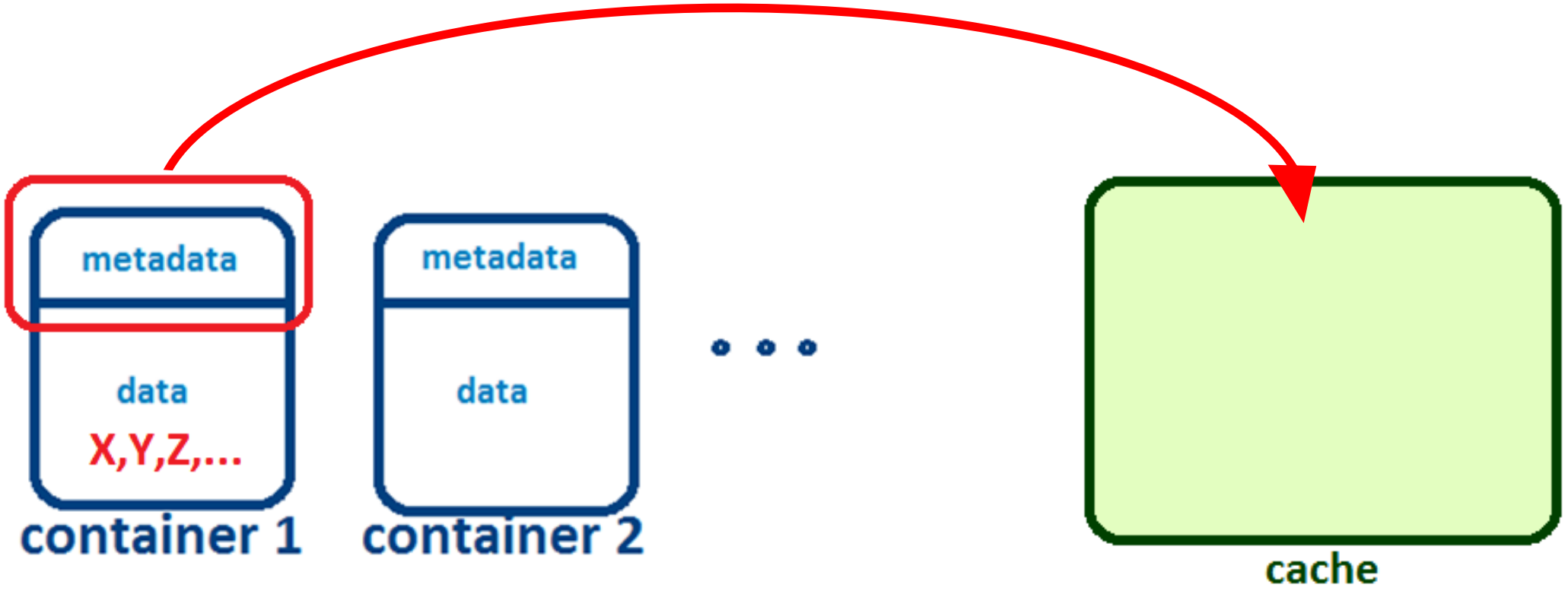
- The cache map segment fingerprint to its corresponding container ID.
- Each segment is kept in some container with other segment that came from the same stream.
- Miss – when we want to check if some segment X is a duplicate we search the cache(Lookup) if the fingerprint is not in the cache then we have a miss and need to check the fingerprint in the index on the disk.

# Acceleration Method #3: Locality Preserved Caching cont.

On a miss we will search for the fingerprint in the index. Simultaneously we fetch all the metadata section from the container the contains X's fingerprint.

# Acceleration Method #3: Locality Preserved Caching cont.

On a miss we will search for the fingerprint in the index. Simultaneously we fetch all the metadata section from the container that contains X's fingerprint to the cache.



# Acceleration Method #3: Locality Preserved Caching cont.

To sum up, these are the operations we have for the cache:

- **Init()** – initialize the segment cache.
- **Insert(container)**-on a miss insert all the metadata section of the container to the cache. With the container ID.(pairs of segment's descriptor and the container ID).
- **Remove(container)**-when there is no room in the cache delete the metadata that belongs to the Least Recent Used container.(LRU)
- **Lookup(fingerprint)**-find the fingerprint in the cache.

# Summery

Let’s connect everything we talked about together:

For an incoming segment for write, the algorithm does the following:

1) Checks to see if it is in the segment cache. If it is in the cache, the incoming segment is a duplicate.


**If X’s fingerprint is in the cache X is a duplicate!**

# Summery

Let’s connect everything we talked about together:

For an incoming segment for write, the algorithm does the following:

1) Checks to see if it is in the segment cache. If it is in the cache, the incoming segment is a duplicate.

2) If it is not in the segment cache, check the Summary Vector. If it is not in the Summary Vector, the segment is new. Write the new segment into the current container.

**X’s fingerprint**



$$Bloom\ \ filter[hash_1(f)]$$
$$Bloom\ \ filter[hash_2(f)]$$
$$Bloom\ \ filter[hash_3(f)]$$

# Summery

Let’s connect everything we talked about together:

For an incoming segment for write, the algorithm does the following:

1) Checks to see if it is in the segment cache. If it is in the cache, the incoming segment is a duplicate.

2) If it is not in the segment cache, check the Summary Vector. If it is not in the Summary Vector, the segment is new. Write the new segment into the current container.

3) If it is in the Summary Vector, lookup the segment index for its container Id.

- If it is in the index, the incoming segment is a duplicate; insert the metadata section of the container into the segment cache. If the segment cache is full, remove the metadata section of the least recently used container first.

# Summery

Let’s connect everything we talked about together:

For an incoming segment for write, the algorithm does the following:

1) Checks to see if it is in the segment cache. If it is in the cache, the incoming segment is a duplicate.

2) If it is not in the segment cache, check the Summary Vector. If it is not in the Summary Vector, the segment is new. Write the new segment into the current container.

3) If it is in the Summary Vector, lookup the segment index for its container Id.

- If it is in the index, the incoming segment is a duplicate; insert the metadata section of the container into the segment cache. If the segment cache is full, remove the metadata section of the least recently used container first.
- If it is not in the segment index, the segment is new. Write the new segment into the current container.

# Results

- The results refer to Real World Data from 2 Data Centers(named A and B)
- Data centers A backup structured data and Data centers B back up a mixture of structured database and unstructured file system data.
- Backup Policy for Data Center A and B: Daily full backup. Each full backup produces over 600 GB.
- We observe 31 days from the initial deployment of a deduplication system for data center A and 48 days for data center B.

# Seeding

*Seed loading is a technology used primarily in remote data backup solutions. It prevents large amounts of backup data being sent over the Internet. Instead, the backup is performed locally on a storage medium (e.g. an external hard disk) which is then shipped to the external storage location, where it is stored in the appropriate account. This method saves the user much time and bandwidth.*

*-Wikipedia*

# Results: physical and logical capacity

**Physical capacity** - the amount of data stored in disk.

**Logical capacity** – the amount of data that all the backups should have take.

For example, at the end of 31st day, the data center A has backed up about 16.9 TB**(Logical capacity),** and the corresponding **physical capacity** is less than 440 GB, reaching a total compression ratio of 38.54 to 1

# Results: physical&logical capacity data center A

Fig 4. shows the logical and Physical capacity of the system over time at data center A.
The ratio between the logical and physical capacity increase.

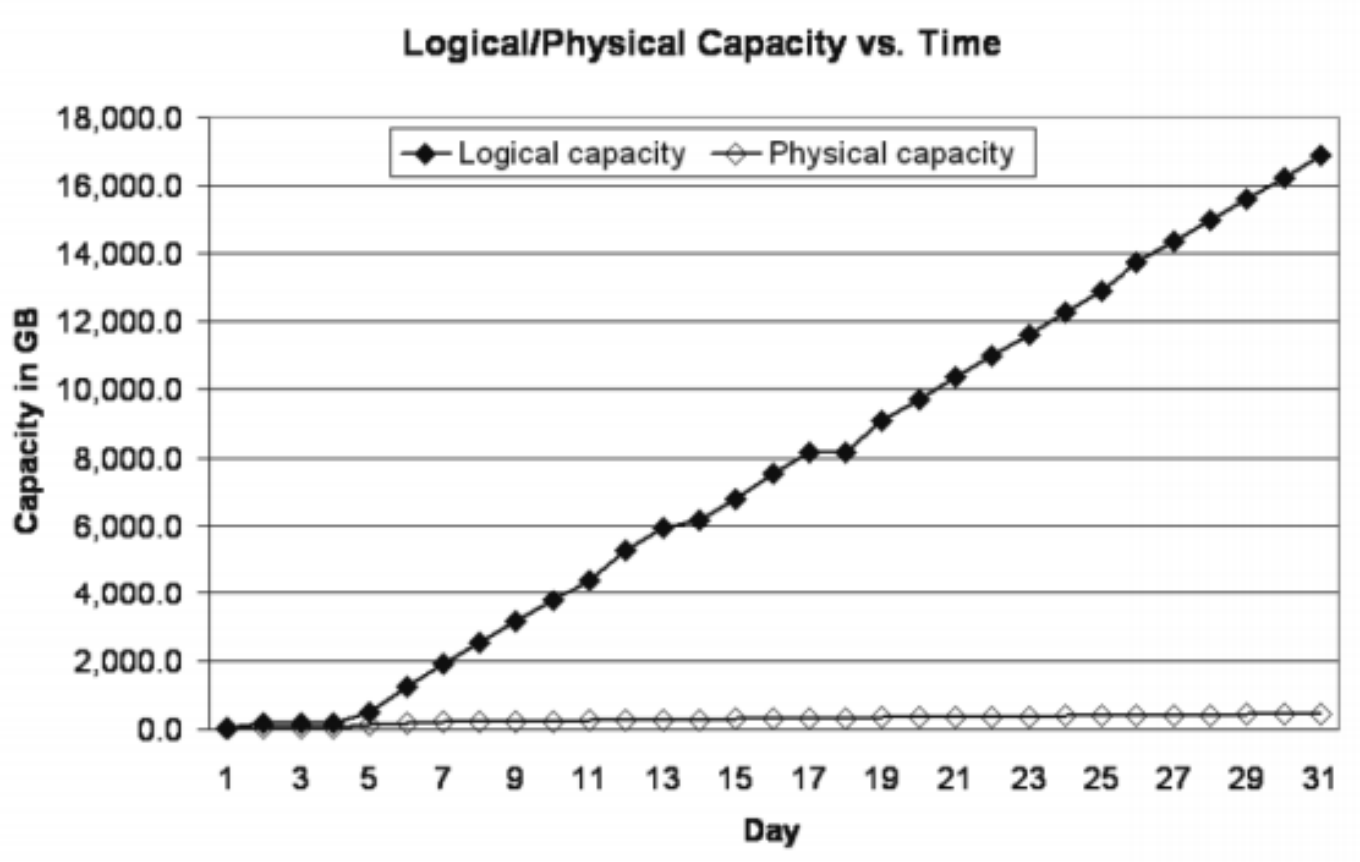



**Figure 4: Logical/Physical Capacities at Data Center A**

# Results: Compression Ratios

- **Daily global compression ratio -** rate of data reduction due to duplicate segment elimination.
- **Daily local compression ratio** - daily rate of data reduction due to Ziv - Lempel compression.
- **cumulative global compression ratio** - the cumulative ratio of data reduction due to duplicate segment elimination.
- **cumulative total compression ratio** - the cumulative ratio of data reduction due to duplicate segment elimination and Ziv-Lempel compression.

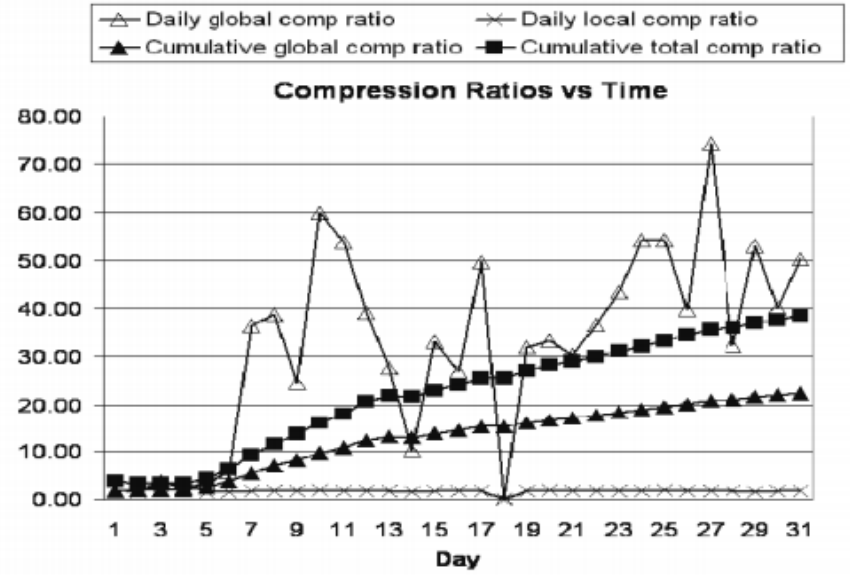



**Figure 5: Compression Ratios at Data Center A**

# Results: Compression Ratios data center A

At the end of 31st day, the cumulative *global* compression ratio reaches **22.53 to 1**, and cumulative *total* compression ratio reaches **38.54 to 1**.

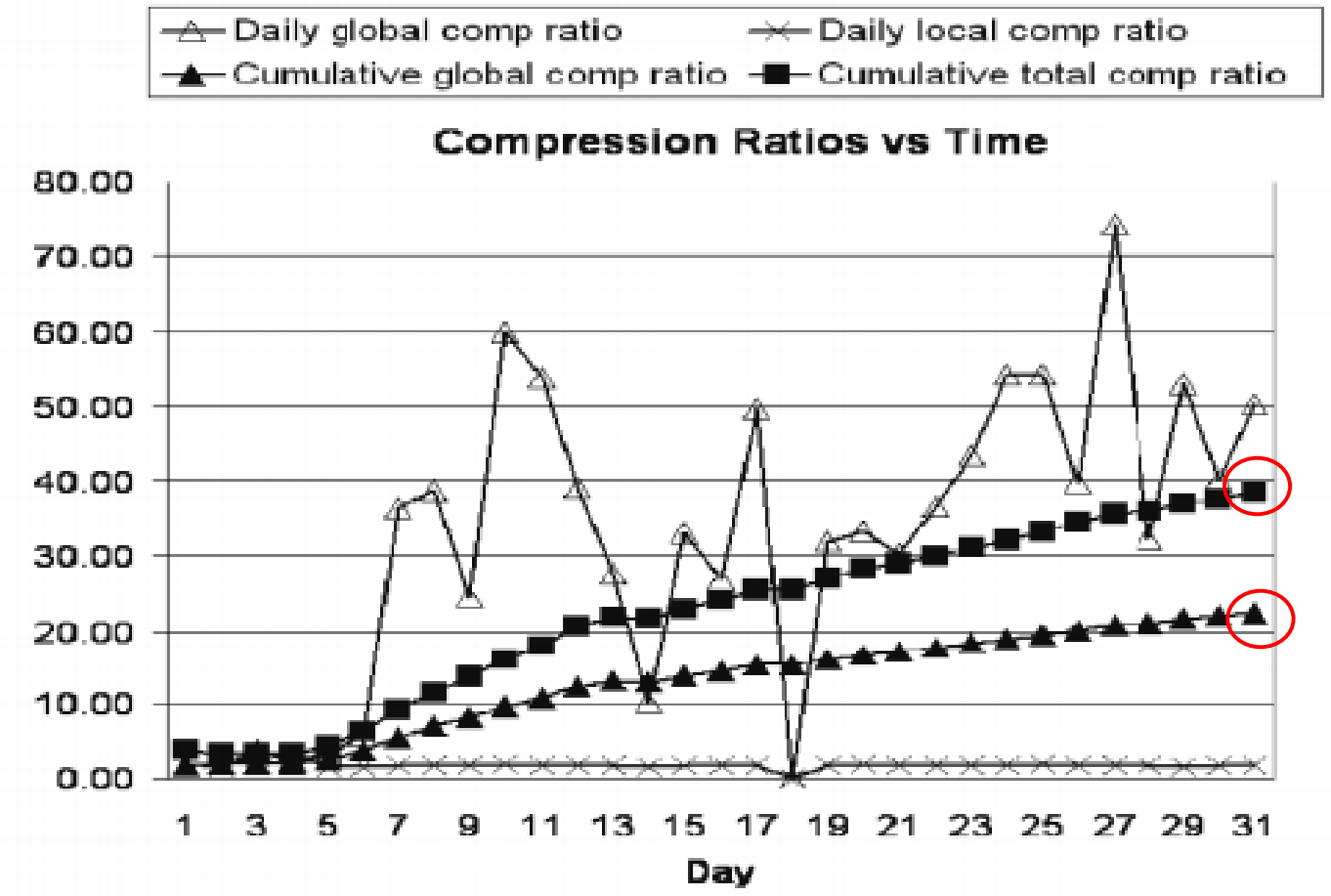



**Figure 5: Compression Ratios at Data Center A**

# Results: Compression Ratios data center A

The daily compression ratios change quite a bit over time, whereas the daily local compression ratios are quite stable.

Table 1 summarizes the minimum, maximum, average, and standard deviation of both daily global and daily local compression ratios, excluding seeding (the first 6) days and no backup on day 18.

| | Min | Max | Average | Standard deviation |
|---|---|---|---|---|
| **Daily global compression** | 10.05 | 74.31 | 40.63 | 13.73 |
| **Daily local compression** | 1.58 | 1.97 | 1.78 | 0.09 |

**Table 1: Statistics on Daily Global and Daily Local Compression Ratios at Data Center A**

# Results: physical&logical capacity data center B

Fig 6. shows the logical and Physical capacity of the system over time at data center B.
The ratio between the logical and physical capacity increase.

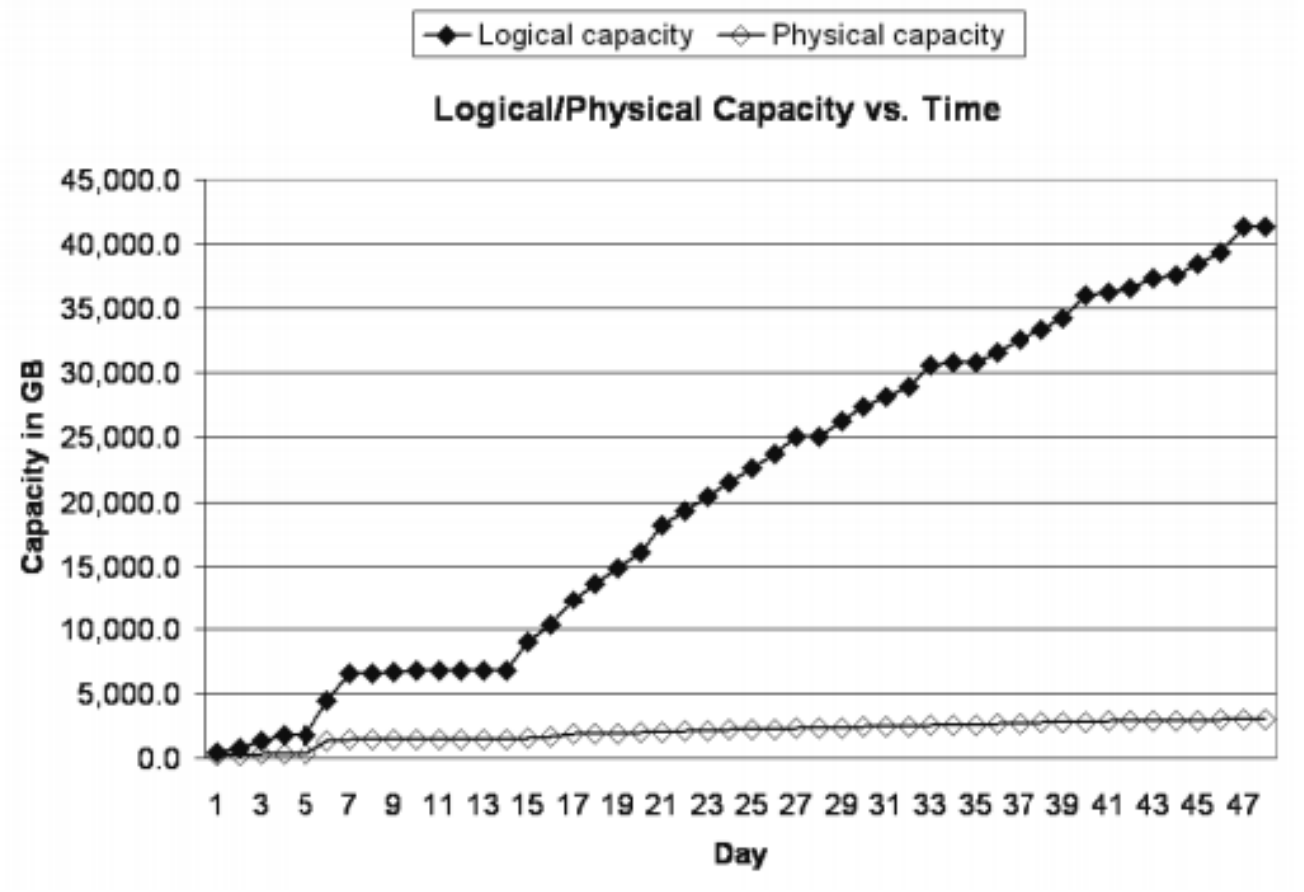



**Figure 6: Logical/Physical Capacities at Data Center B.**

# Results: Compression Ratios data center B

At the end of 48th day, the cumulative *global* compression ratio reaches **6.85 to 1**, and cumulative *total* compression ratio reaches **13.71 to 1**.

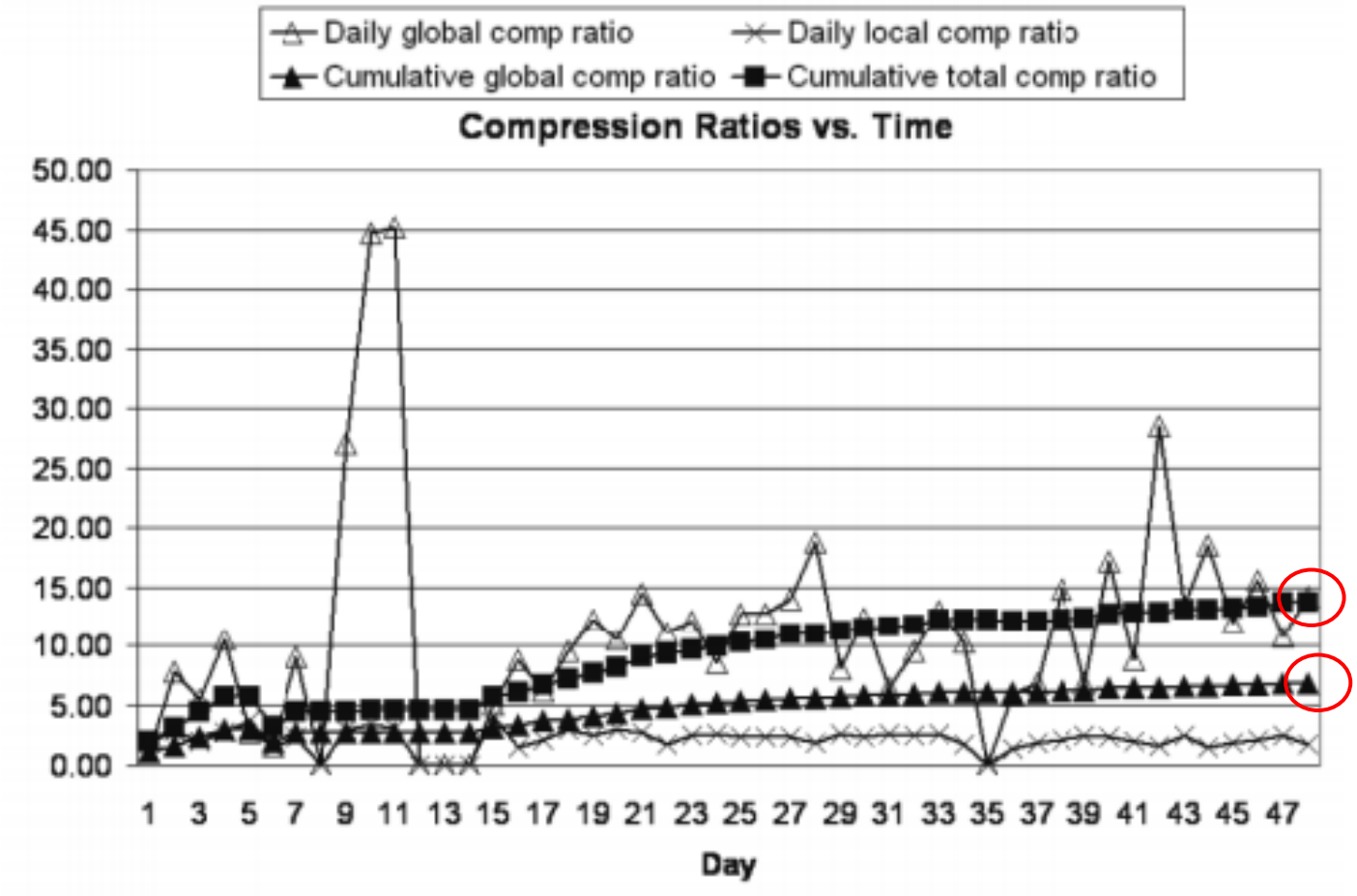



**Figure 7: Compression Ratios at Data Center B.**

# Results: Compression Ratios data center B

The daily compression ratios change quite a bit over time, whereas the daily local compression ratios are quite stable.

Table 2 summarizes the minimum, maximum, average, and standard deviation of both daily global and daily local compression ratios, excluding seeding days and no backup on days.

| | Min | Max | Average | Standard deviation |
|---|---|---|---|---|
| **Daily global compression** | 5.09 | 45.16 | 13.92 | 9.08 |
| **Daily local compression** | 1.40 | 4.13 | 2.33 | 0.57 |

**Table 2: Statistics on Daily Global and Daily Local Compression Ratios at Data Center B**

# Results

The two sets of results show that the deduplication storage system works well with the real world datasets. As expected, both cumulative global and cumulative total compression ratios increase as the system holds more backup data

# Results: Seeding

Note: During seeding, duplicate segment elimination tends to be ineffective, because most segments are new. After seeding duplicate segment elimination becomes extremely effective.

# Results: I/O Savings with Summary Vector and Locality Preserved Caching

To determine the effectiveness of the 3 acceleration methods: (Summary, Vector and Locality  and Preserved Caching), we examine the savings for disk reads to find duplicate segments using a Summary Vector and Locality Preserved Caching.

We examine the results over two internal datasets.

One is a daily full backup of a company-wide Exchange information store over a 135-day period. (we'll named it A)

The other is the weekly full and daily incremental backup of an Engineering department over a 100-day period. (we'll named it B)

**We send the backup data with one stream!**

# Results: I/O Savings with Summary Vector and Locality Preserved Caching

A B

| | Exchange data | Engineering data |
|---|---|---|
| Logical capacity (TB) | 2.76 | 2.54 |
| Physical capacity after deduplicating segments (TB) | 0.49 | 0.50 |
| Global compression | 5.69 | 5.04 |
| Physical capacity after local compression (TB) | 0.22 | 0.261 |
| Local compression | 2.17 | 1.93 |
| Total compression | 12.36 | 9.75 |

**Table 3: Capacities and Compression Ratios on Exchange and Engineering Datasets**

# Results: I/O Savings with Summary Vector and Locality Preserved Caching

we look at the number of disk reads for segment index lookups and locality prefetches needed to find duplicates during write for four cases:

I. with neither Summary Vector nor Locality Preserved Caching;
II. with Summary Vector only
III. with Locality Preserved Caching only;
IV. with both Summary Vector and Locality Preserved Caching.

# Results: I/O Savings with Summary Vector and Locality Preserved Caching

A B

| | Exchange data | | Engineering data | |
|---|---|---|---|---|
| | # disk I/Os | % of total | # disk I/Os | % of total |
| no Summary Vector and no Locality Preserved Caching | 328,613,503 | 100.00% | 318,236,712 | 100.00% |
| Summary Vector only | 274,364,788 | 83.49% | 259,135,171 | 81.43% |
| Locality Preserved Caching only | 57,725,844 | 17.57% | 60,358,875 | 18.97% |
| Summary Vector and Locality Preserved Caching | 3,477,129 | 1.06% | 1,257,316 | 0.40% |

**Table 4: Index and locality reads. This table shows the number disk reads to perform index lookups or fetches from the container metadata for the four combinations: with and without the Summary Vector and with and without Locality Preserved Caching. Without either the Summary Vector or Locality Preserved Caching, there is an index read for every segment. The Summary Vector avoids these reads for most new segments. Locality Preserved Caching avoids index lookups for duplicate segments at the cost an extra read to fetch a group of segment fingerprints from the container metadata for every cache miss for which the segment is found in the index.**

# Results: I/O Savings with Summary Vector and Locality Preserved Caching Summary

In general, the Summary Vector is very effective for new data, and Locality Preserved Caching is highly effective for little or moderately changed data.

For backup data, the first full backup (seeding equivalent) does not have as many duplicate data segments as subsequent full backups. As a result, the Summary Vector is effective to avoid disk I/Os for the index lookups during the first full backup, whereas Locality Preserved Caching is highly beneficial for subsequent full backups.

# Results: Throughput

Determine the throughput of the deduplication storage system.

In this experiment they used one or mote streams.

We insert versions of files to the backup, each successive version (“generation”) is a somewhat modified copy of the preceding generation in the series. The modifications between generation include:

1. data reordering
2. deletion of existing data
3. addition of new data

# Results: Throughput

The **Write** throughput of each generation(=version).
You can see how more streams increase the throughput.
We also achieved our goal to reach 100 MB/s

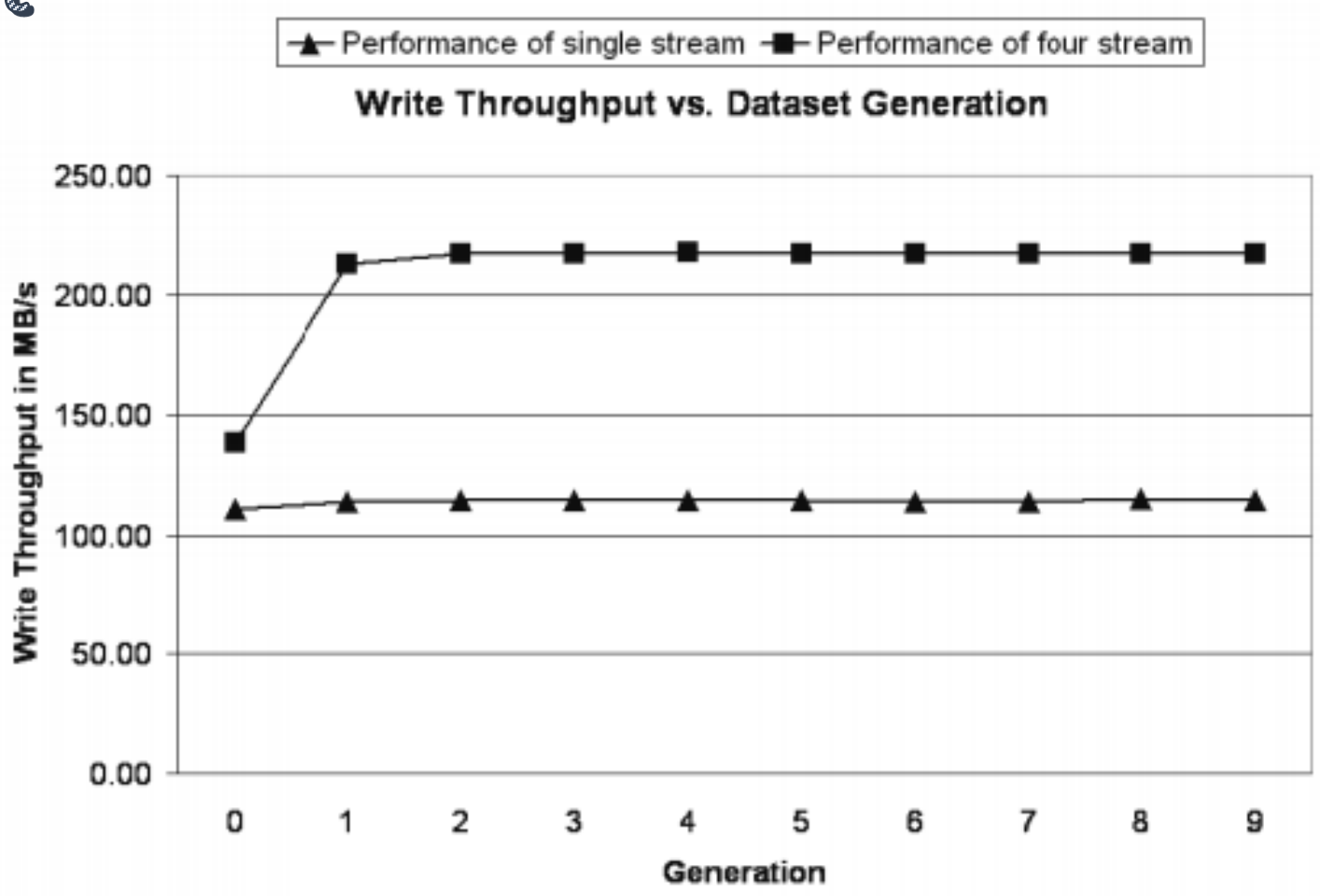



**Figure 8: Write Throughput of Single Backup Client and 4 Backup Clients.**

# Results: Throughput

The **Read** throughput of each generation(=version).
As the generation increase the throughput decrease. ***How could it be?*** Future generations have more duplicate data segments than the first few. However, the read throughput stays at about 140 MB/sec for later generations because of Stream In formed Segment Layout and Locality Preserved Caching.

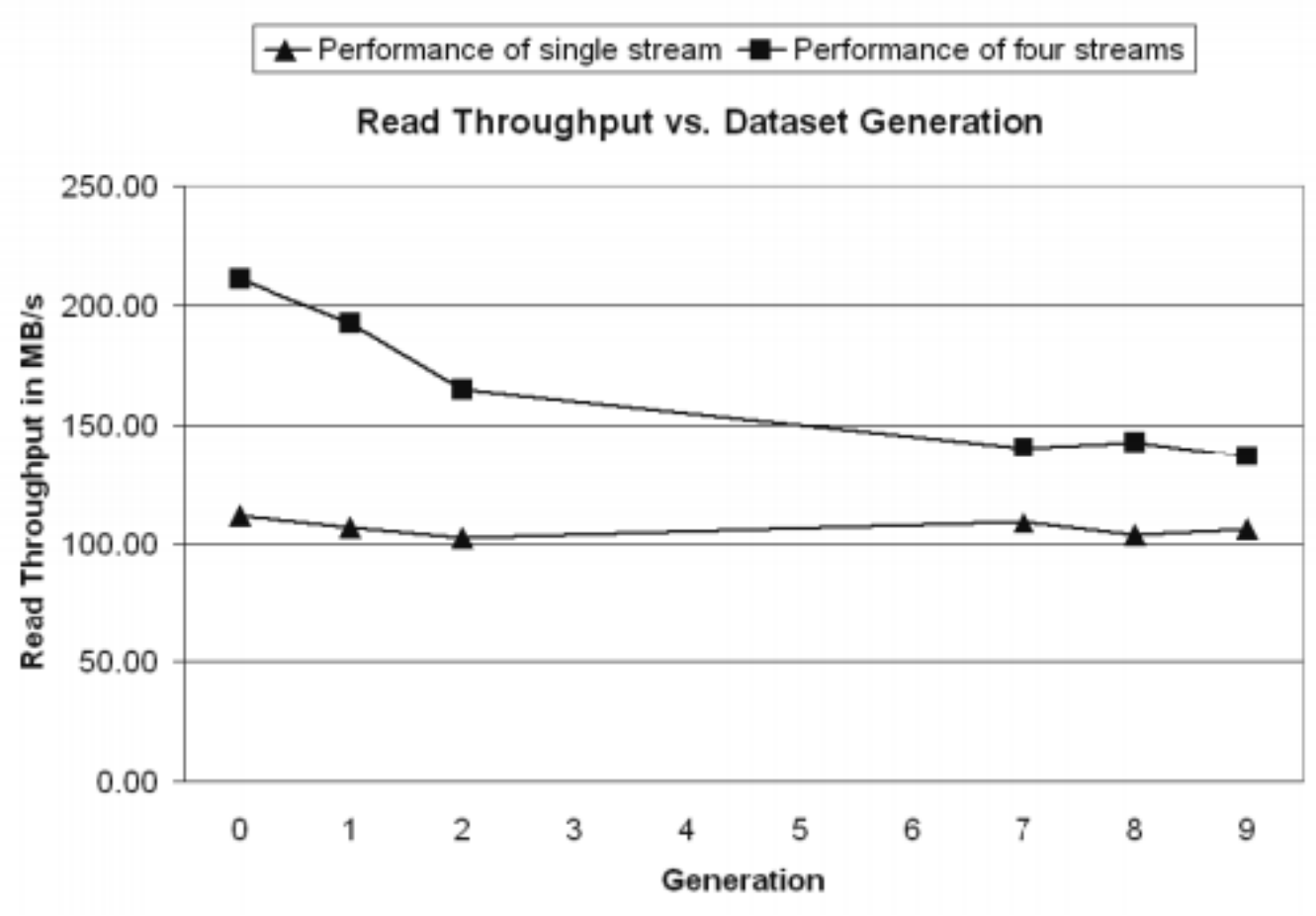



**Figure 9: Read Throughput of Single Backup Client and 4 Backup Clients**

# Results: Throughput

Note that the write throughput is more important to us than the read throughput. Because we need to write the backup within a specific window of time. But restoring(reading) the backup is less important to us. (But still important enough).

# Related Work

- Most of the work on deduplication focused on basic methods and compression ratios, not on high throughput.

- Some previous work only do deduplication at file level, such systems can achieve only limited global compression.

# Related Work - Venti

Venti is a previous DDFS with some changes:

- uses fixed-size data
- It uses a large on-disk index with a straightforward index cache to lookup fingerprints. Since fingerprints have no locality, their index cache is not effective.
- When using 8 disks to lookup fingerprints in parallel, its throughput is still limited to less than 7 MB/sec.
- Venti used a container abstraction to layout data on disks, but did not apply Stream-Informed Segment Layout.

# Related Work - variable-size segments

To tolerate shifted contents, modern deduplication systems remove redundancies at variable-size data blocks divided based on their contents.

Manber described a method to determine anchor points of a large file when certain bits of rolling fingerprints are zeros [Man93] and showed that **Rabin fingerprints [Rab81, Bro93] can be computed efficiently**

# Conclusions - What did we have today?

- ✓ we learned how basic DDFS works.(Segment Store, Content Store, containers and more…)
- ✓ we learned 3 methods that accelerate the Deduplication process.(summary vector, SISL,LPC)
- ✓ We viewed some results( in different conditions)
- ✓ We saw some related works