Deduplication: Overview & Case Studies

CSCI 333
Williams College
Lecture Outline

Background

Content Addressable Storage (CAS)

Deduplication
  Chunking
  The Index

Other CAS applications
Background

Content Addressable Storage (CAS)

Deduplication
   Chunking
   The Index

Other CAS applications
Content Addressable Storage (CAS)

Deduplication systems often rely on Content Addressable Storage (CAS)

Data is indexed by some content identifier

The content identifier is determined by some function over the data itself
- often a cryptographically strong hash function
CAS

Example: I send a document to be stored remotely on some content addressable storage
Example:

The server receives the document, and calculates a unique identifier called the data's **fingerprint**
CAS

The fingerprint should be:

unique to the data
   - NO collisions

one-way
   - hard to invert
The **fingerprint** should be:

- unique to the data
  - NO collisions
- one-way
  - hard to invert

**SHA-1:**

20 bytes (160 bits)

\[
P(\text{collision}(a,b)) = \left(\frac{1}{2}\right)^{160} \\
coll(N, 2^{160}) = \binom{N}{2} \left(\frac{1}{2}\right)^{160}
\]
CAS

Example:

SHA-1( ) = de9f2c7fd25e1b3a...

Name  de9f2c7fd25e1b3a...

homework.txt  de9f2c7fd25e1b3a... data
Example:

I submit my homework, and my “buddy”
Harold also submits my homework...
Example:

Same contents, same fingerprint.
CAS

Example:

Same contents, same fingerprint.

The data is only stored once!
Background

- Content Addressable Storage (CAS)
- Deduplication
  - Chunking
  - The Index
- Other applications
Example:

Now suppose Harry writes his name at the top of my document.
CAS

Example:

The fingerprints are completely different, despite the (mostly) identical contents.
CAS

Problem Statement:

What is the appropriate granularity to address our data?

What are the tradeoffs associated with this choice?
Background

Content Addressable Storage (CAS)

Deduplication
  Chunking
  The Index

Other applications
Deduplication

Chunking breaks a data stream into segments

\[
\text{SHA1(} \text{DATA} \text{)} \quad \text{becomes} \\
\text{SHA1(} \text{CK1} \text{)} + \text{SHA1(} \text{CK2} \text{)} + \text{SHA1(} \text{CK3} \text{)}
\]

How do we divide a data stream?

How do we reassemble a data stream?
Deduplication

Division.

Option 1: fixed-size blocks
- Every (?)KB, start a new chunk

Option 2: variable-size chunks
- Chunk boundaries dependent on chunk contents
Deduplication

**Division:** fixed-size blocks

```
hw-bill.txt

=  
=  
=  

=  
=  
=  

=  
=  
=  

=  
=  
=  

=  
=  
=  

hw-harold.txt
```
Suppose Harold adds his name to the top of my homework. This is called the boundary shifting problem.
Deduplication

Division.

Option 1: fixed-size blocks
- Every 4KB, start a new chunk

Option 2: variable-size chunks
- Chunk boundaries dependent on chunk contents
Deduplication

**Division:** variable-size chunks

parameters:
- Window of width $w$
- Target pattern $t$

- Slide the window byte by byte across the data, and compute a window fingerprint at each position.

- If the fingerprint matches the target, $t$, then we have a *fingerprint match* at that position.
Deduplication

Division: variable-size chunks

- Slide the window byte by byte across the data, and compute a window fingerprint at each position.
- If the fingerprint matches the target, $t$, then we have a **fingerprint match** at that position.
Deduplication

**Division:** variable-size chunks

hw-wkj.txt  hw-harold.txt
Deduplication

**Division: variable-size chunks**

Suppose Harold adds his name to the top of my homework. Only introduce one new chunk to storage.
Deduplication

**Division**: variable-size chunks

Sliding window properties:
- collisions are OK, but
  - average chunk size should be configurable
- reuse overlapping window calculations

\[ w, t \]
- expect a chunk every \( 2^{t-1}+w \) bytes

**LBFS**: \( w=48, t=13 \)
- expect a chunk every 8KB

\[ \rightarrow \text{Rabin fingerprints} \]
Deduplication

**Division:** variable-size chunks

Rabin fingerprint: preselect divisor $D$, and an irreducible polynomial

\[
R(b_1, b_2, \ldots, b_w) = (b_1 p^{w-1} + b_2 p^{w-2} + \ldots + b_w) \mod D
\]

\[
R(b_{i}, \ldots, b_{i+w-1}) = ((R(b_{i-1}, \ldots, b_{i+w-2}) - b_{i-1} p^{w-1}) p + b_{i+w-1}) \mod D
\]

- Arbitrary window of width $w$
- Previous window calculation
- Previous first term
Deduplication

Recap:

Chunking breaks a data stream into smaller segments

→ What do we gain from chunking?
→ What are the tradeoffs?

+ Finer granularity of sharing
+ Finer granularity of addressing
- Fingerprinting is an expensive operation
- Not suitable for all data patterns
- Index overhead
Deduplication

Reassembling chunks:

Recipes provide directions for reconstructing files from chunks
Deduplication

Reassembling chunks:

Recipes provide directions for reconstructing files from chunks
CAS

Example:

Name: homework.txt

de9f2c7fd25e1b3a...

Metadata:

<SHA1>
<SHA1>
<SHA1>
...

de9f2c7fd25e1b3a...

recipe/data

???
Deduplication

Background

Content Addressable Storage (CAS)

Deduplication

  Chunking

  The Index

Other applications
Deduplication

The Index:

SHA-1 fingerprint uniquely identifies data, but the index translates fingerprints to chunks.

<chunk_i> = {location, size?, refcount?, compressed?, ...}
Deduplication

The Index:

For small chunk stores:
- database, hash table, tree

For a large index, legacy data structures won't fit in main memory
- each index query requires a disk seek

- why?
  SHA-1 fingerprints independent and randomly distributed
  - no locality

Known as the **index disk bottleneck**
Deduplication

The Index:

Back of the envelope:

Average chunk size: 4KB
Fingerprint: 20B

20TB unique data = 100GB SHA-1 fingerprints
Deduplication

Disk bottleneck:

Data Domain strategy:
- filter unnecessary lookups
- piggyback useful work onto the disk lookups that are necessary

Diagram:
- Locality Preserving Cache
- Summary Vector
- Stream Informed Segment Layout (Containers)
Deduplication

Disk bottleneck:

Summary vector

- Bloom filter (any AMQ data structure works)

Filter properties:
- No false negatives
  - if an FP is in the index, it is in summary vector
- Tuneable false positive rate
  - We can trade memory for accuracy

Note: on a false positive, we are no worse off
- We just do the disk seek we would have done anyway
Deduplication

Disk bottleneck:

Data Domain strategy:
- filter unnecessary lookups
- piggyback useful work onto the disk lookups that are necessary

Locality Preserving Cache

Bloom Filter

Summary Vector

Disk

Stream Informed Segment Layout (Containers)
Deduplication

Disk bottleneck:

Stream informed segment layout (SISL)
- variable sized chunks written to fixed size containers
- chunk descriptors are stored in a list at the head
  \( \rightarrow \) “temporal locality” for hashes within a container

Principle:
- backup workloads exhibit chunk locality
Deduplication

Disk bottleneck:

Data Domain strategy:
- filter unnecessary lookups
- piggyback useful work onto the disk lookups that are necessary

Locality Preserving Cache

Bloom Filter

Summary Vector

Group Fingerprints: Temporal Locality

Stream Informed Segment Layout (Containers)
Deduplication

Disk bottleneck:

Locality Preserving Cache (LPC)
- LRU cache of candidate fingerprint groups

Principle:
- if you must go to disk, make it worth your while
Deduplication

Disk bottleneck:

START

Read request for chunk fingerprint

Fingerprint in Bloom filter?

Yes

Fingerprint in LPC?

Yes

No

No Lookup Necessary

END

No

On-disk fingerprint index lookup: get container location

No

Read data from target container.

Yes

Prefetch fingerprints from head of target data container.
Deduplication

Summary: Dedup and the 4 W's

Dedup Goal: eliminate repeat instances of identical data

What (granularity) to dedup?

Where to dedup?

When to dedup?

Why dedup?
Summary: Dedup and the 4 W's

**What** (granularity) to dedup?

<table>
<thead>
<tr>
<th></th>
<th>Whole-file</th>
<th>Fixed-size</th>
<th>Content-defined</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Chunking overheads</strong></td>
<td>N/A</td>
<td>offsets</td>
<td>Sliding window fingerprinting</td>
</tr>
<tr>
<td><strong>Dedup Ratio</strong></td>
<td>All-or-nothing</td>
<td>Boundary shifting problem</td>
<td>Best</td>
</tr>
<tr>
<td><strong>Other notes</strong></td>
<td>Low index overhead, compressed/ encrypted/ media</td>
<td>(Whole-file)+ Ease of implementation, selective caching, synchronization</td>
<td>Latency, CPU intensive</td>
</tr>
</tbody>
</table>
Deduplication

Summary: Dedup and the 4 W's

**Where** to dedup?

- Dedup before sending data over the network
  - + save bandwidth
  - - client complexity
  - - trust clients?

- Dedup at storage server
  - + server more powerful
  - - centralized data structures

hybrid

Client index checks membership,
Server index stores location
Deduplication

Summary: Dedup and the 4 W's

**When** to dedup?

- **inline**
  - + never store duplicate data
  - - slower → index lookup per chunk
  - + faster → save I/O for duplicate data

- **post-process**
  - - temporarily wasted storage
  - + faster → stream long writes, reclaim in the background
  - - may create (even more) fragmentation

**hybrid**

→ post-processing faster for initial commits
→ switch to inline to take advantage of I/O savings
Deduplication

Why dedup?

Perhaps you have a looooooot of data...
   - enterprise backups

Or data that is particularly amenable to deduplication...
   - small or incremental changes
   - data that is not encrypted or compressed

Or that changes infrequently.
   - blocks are immutable → no such thing as a “block modify”
   - rate of change determines container chunk locality

Ideal use case: “Cold Storage”
Background

Content Addressable Storage (CAS)

Deduplication
  Chunking
  The Index

Other applications
Other CAS Applications

Data verification

CAS can be used to build tamper evident storage. Suppose that:
- you can't fix a compromised server,
- but you never want be fooled by one

**Insight:** Fingerprints uniquely identify data
- hash before storing data, and save the fp locally
- rehash data and compare fps upon receipt