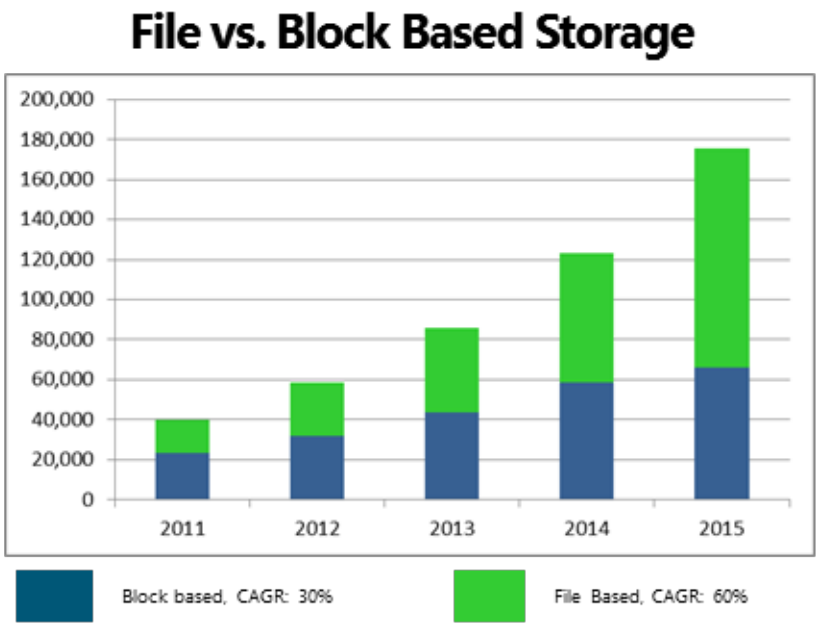
Data Deduplication as Platform for Virtualization and High Scale Storage

Adi Oltean & Sudipta Sengupta
Microsoft Corporation
Redmond, WA, USA

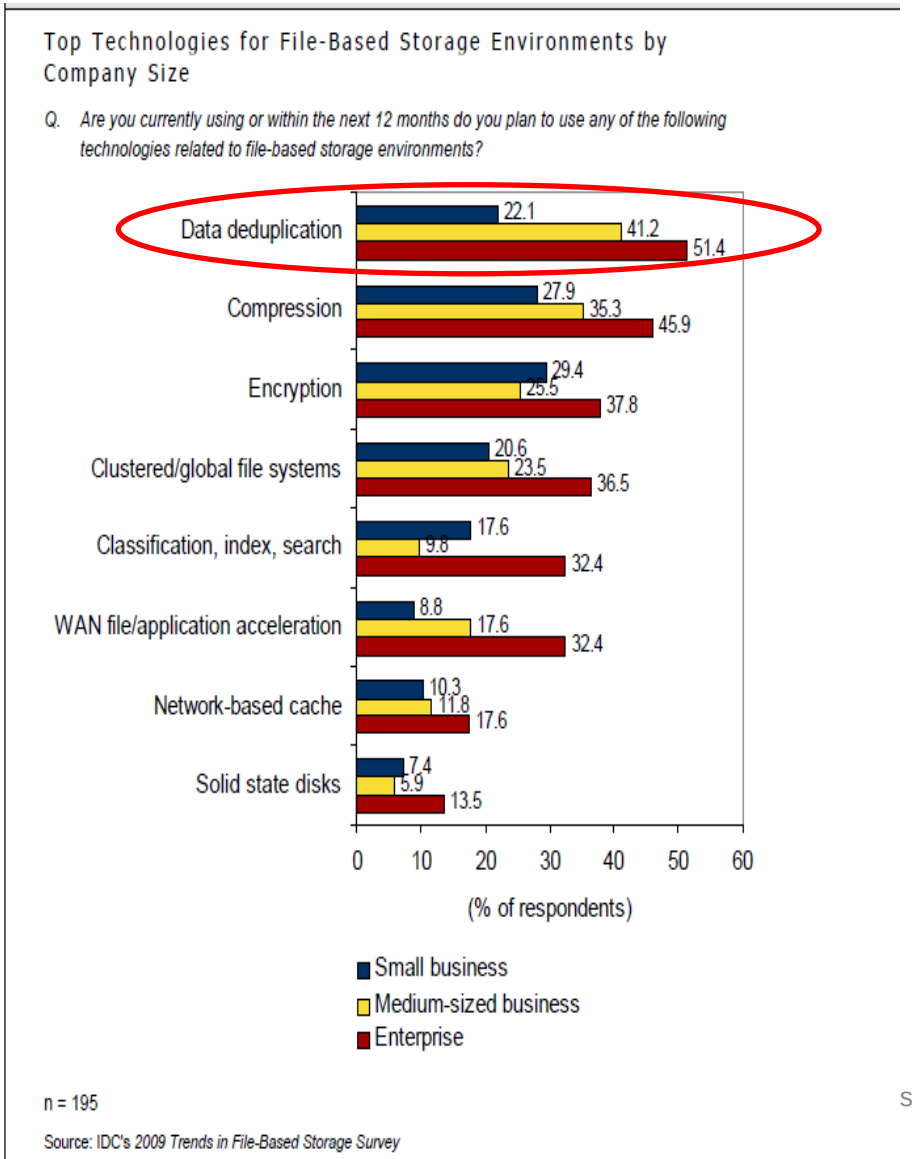
File-based Storage Market

- Growth in file-based data
  - 60% growth 2001-2015 [IDC]
  - Rising storage TCO despite declining disk drive cost
- Increasing access to data over low bandwidth links
  - 60% of IWs working from branch, road, or home
  - Data access increasingly over WAN



Source: IDC Enterprise Disk Storage Consumption Model

Customer/Market response



Major players	
EMC/DataDomain	Backup data dedup $2.1B acquisition
NetApp	Backup and Primary data dedup
Dell/Ocarina	Extensive file format aware optimization $150M acquisition
Permabit	Backup and Primary data dedup
Oracle/ZFS	Native dedup support in ZFS
Linux/SDFS	Open source Linux Dedup project

Trends:

- Package Dedup in a Storage appliance offering
- Move from Backup → Primary → Live data

Deduplication of Storage

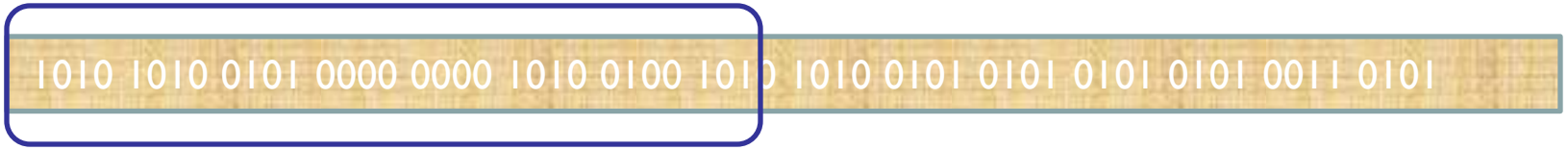
- Detect and remove duplicate data in storage systems
  - Data at-rest: Storage space savings
  - Data on-wire: Network bandwidth and disk I/O savings
- High-level techniques
  - Content based chunking, detect/store unique chunks only
  - Similarity based, differential (delta) encoding

Content based Chunking

- Calculate Rabin fingerprint hash for each sliding window (16 byte)

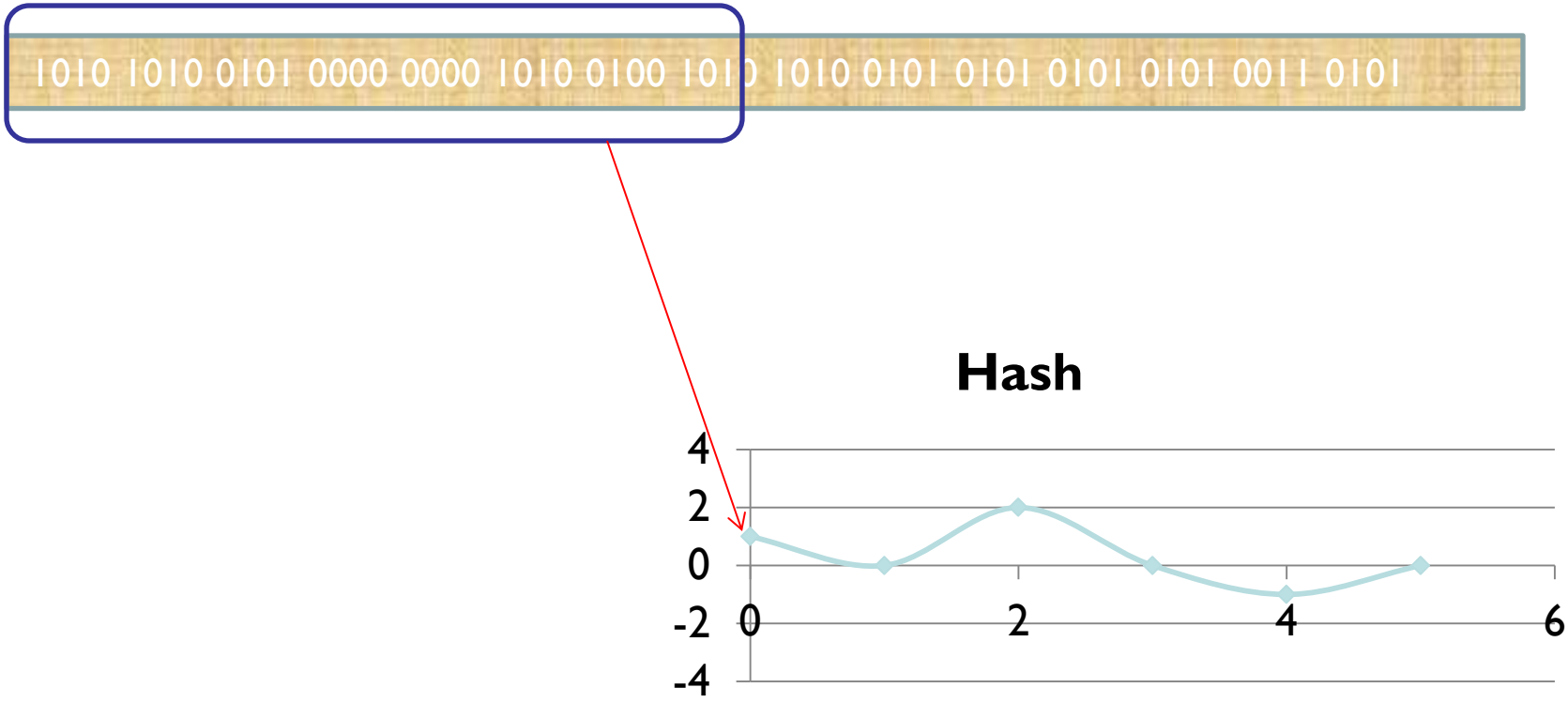
Content based Chunking

- Calculate Rabin fingerprint hash for each sliding window (16 byte)



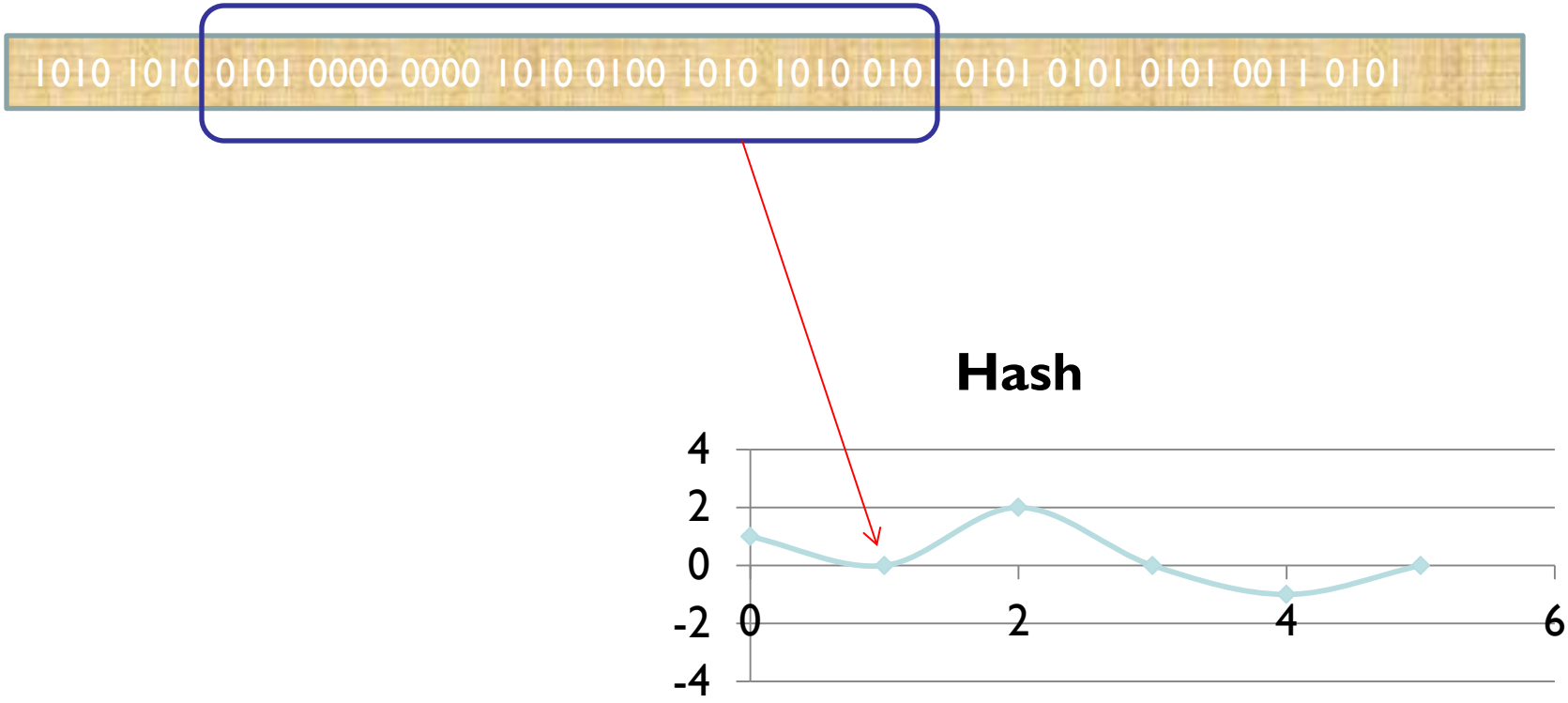
Content based Chunking

- Calculate Rabin fingerprint hash for each sliding window (16 byte)



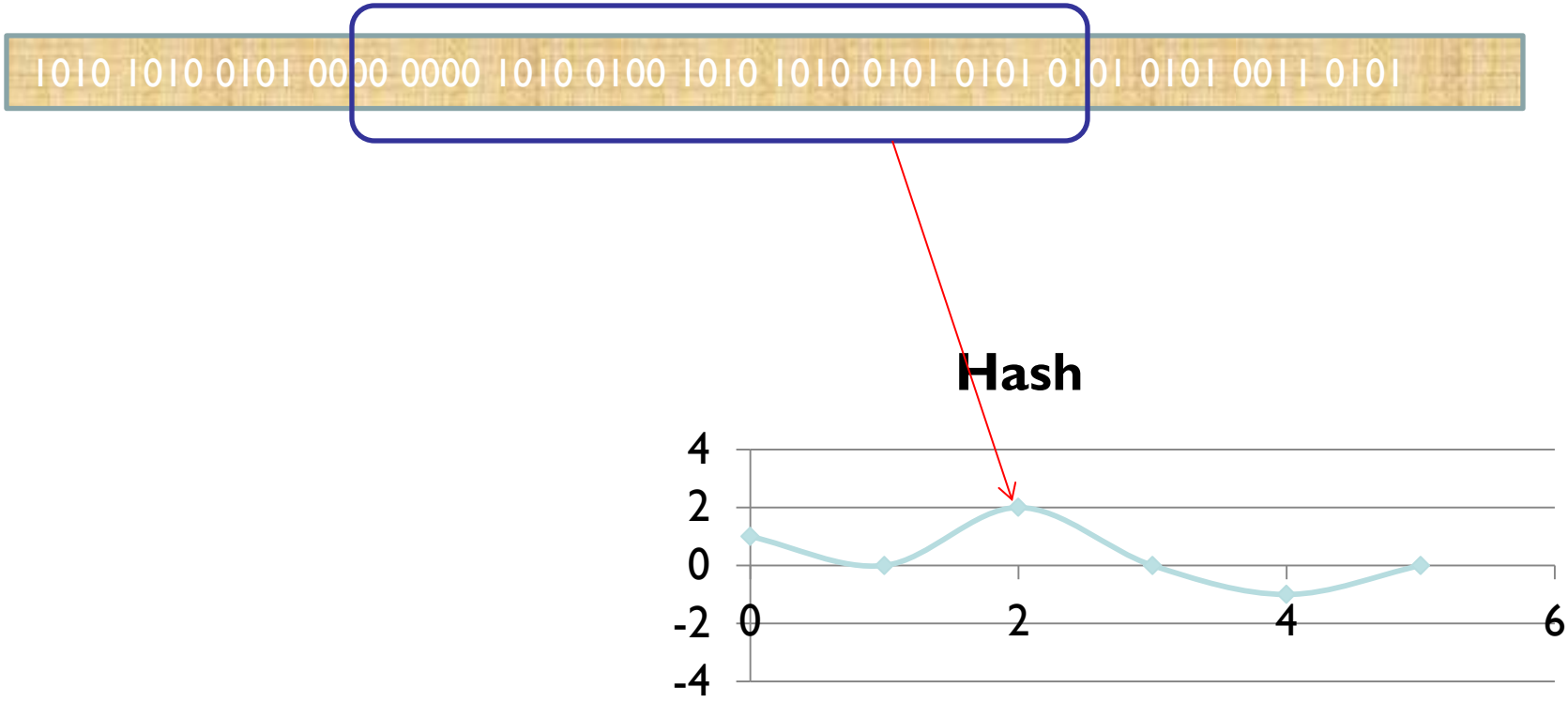
Content based Chunking

- Calculate Rabin fingerprint hash for each sliding window (16 byte)



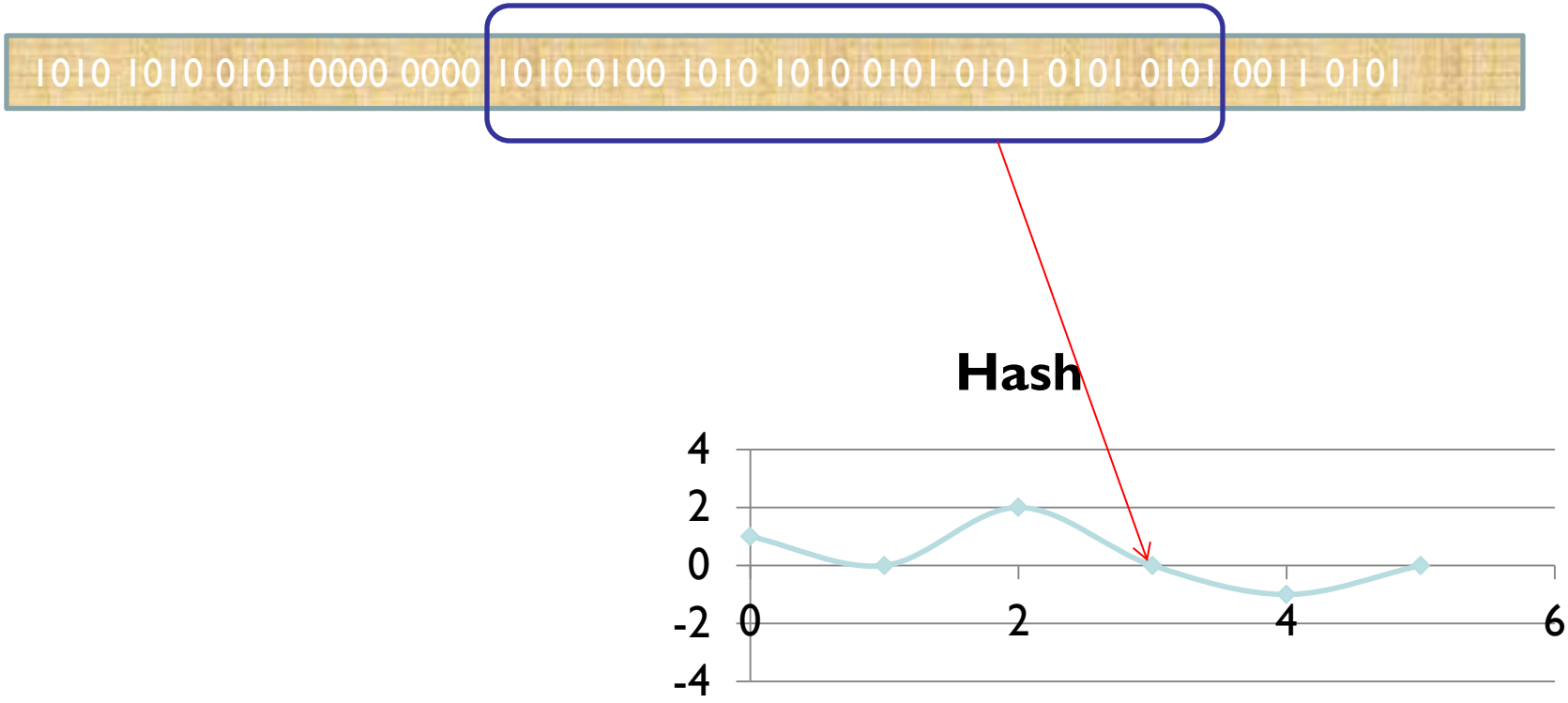
Content based Chunking

- Calculate Rabin fingerprint hash for each sliding window (16 byte)



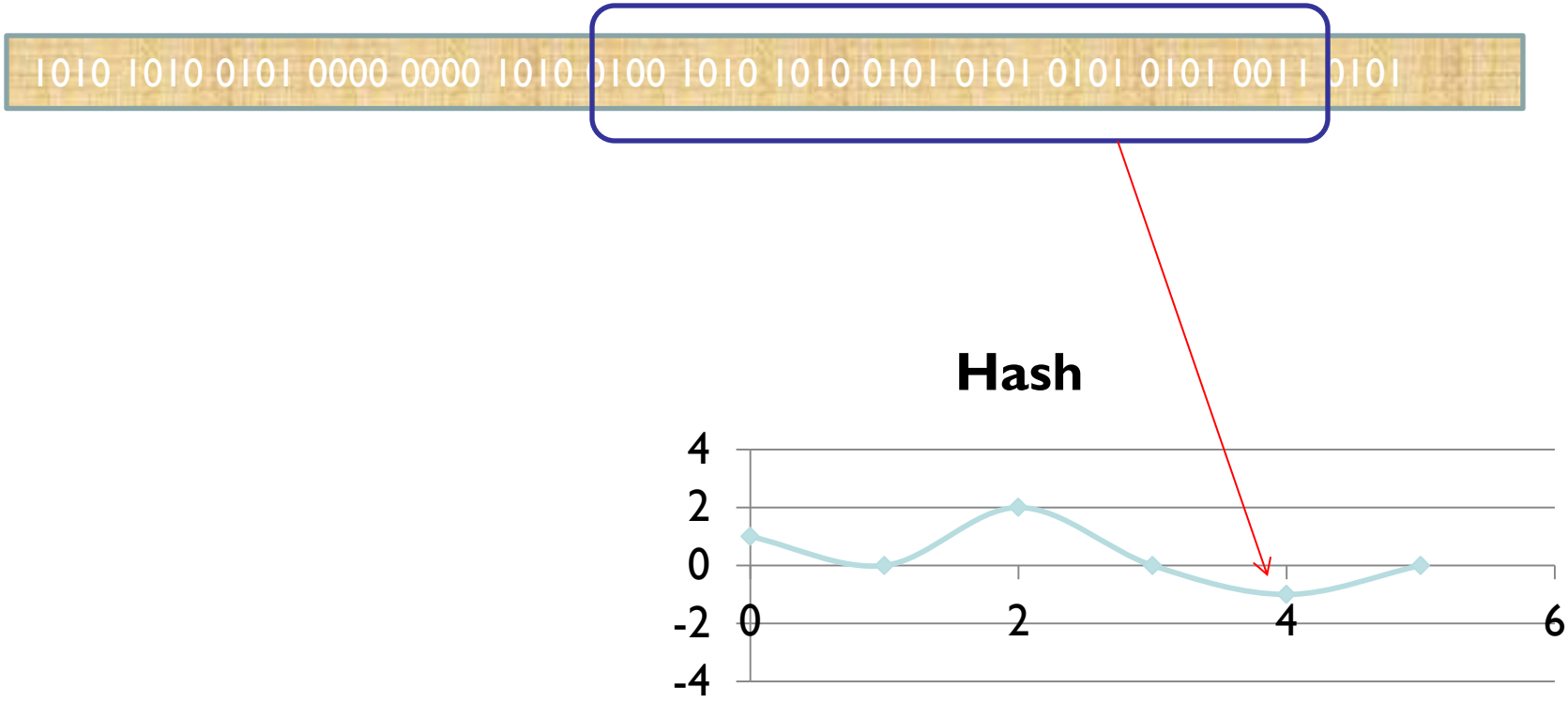
Content based Chunking

- Calculate Rabin fingerprint hash for each sliding window (16 byte)



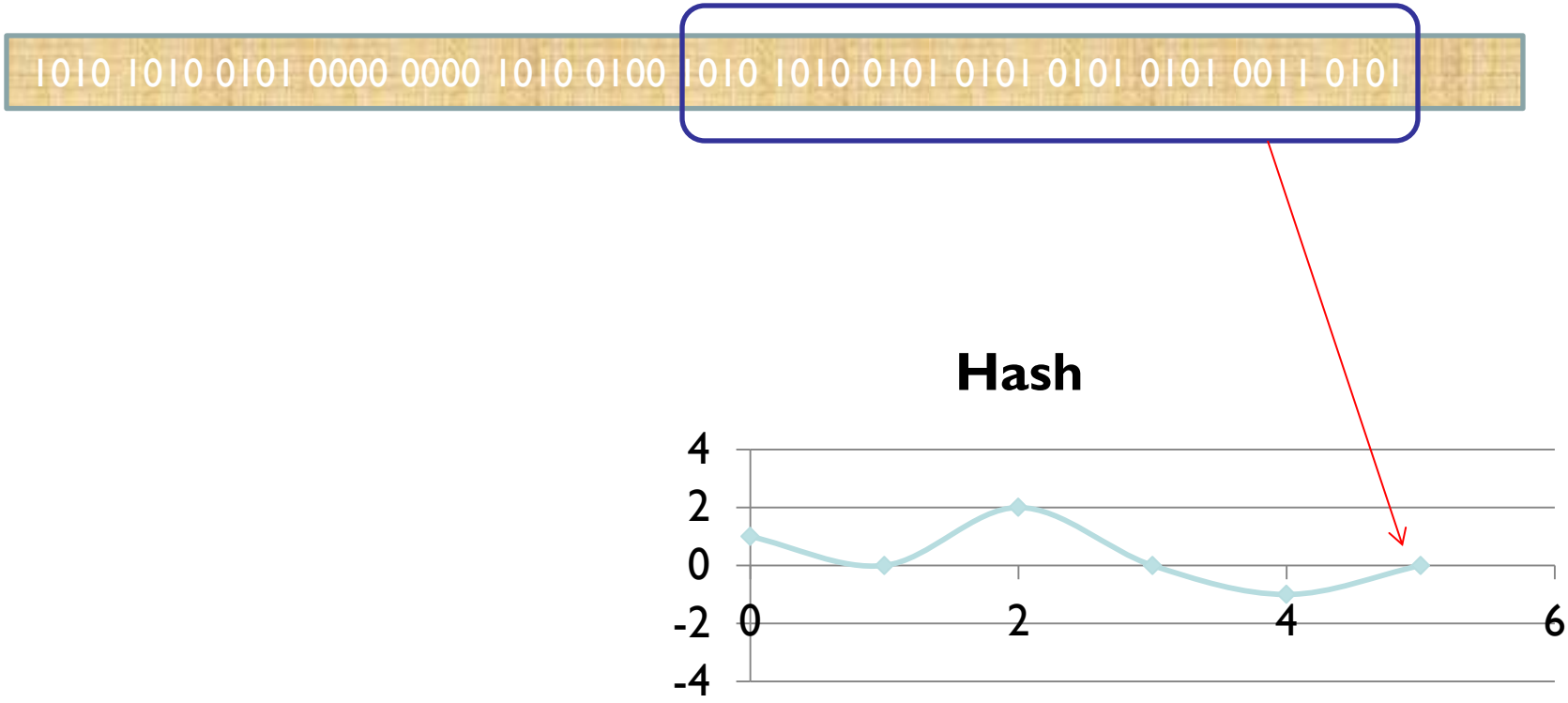
Content based Chunking

- Calculate Rabin fingerprint hash for each sliding window (16 byte)



Content based Chunking

- Calculate Rabin fingerprint hash for each sliding window (16 byte)



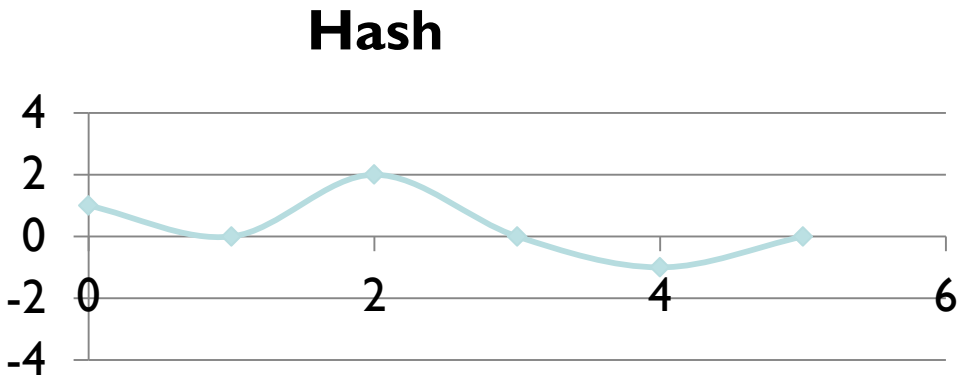
Content based Chunking

- Calculate Rabin fingerprint hash for each sliding window (16 byte)

1010 1010 0101 0000 0000 1010 0100 1010 1010 0101 0101 0101 0101 0011 0101

If Hash matches a particular pattern,

Declare a chunk boundary

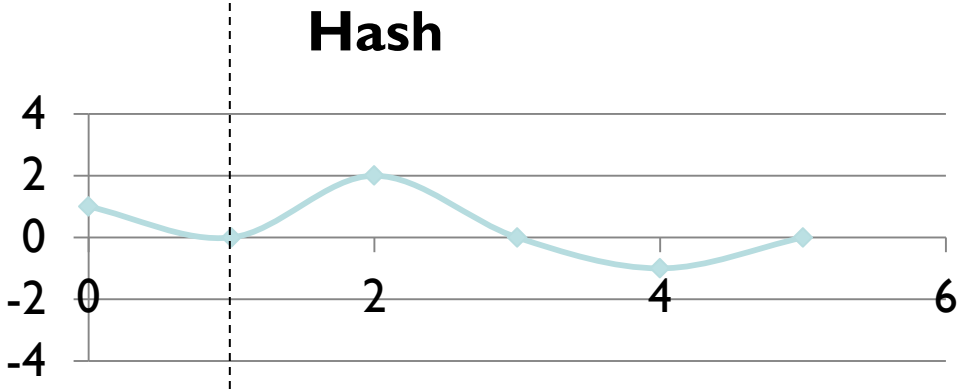


Content based Chunking

- Calculate Rabin fingerprint hash for each sliding window (16 byte)

If Hash matches a particular pattern,

Declare a chunk boundary

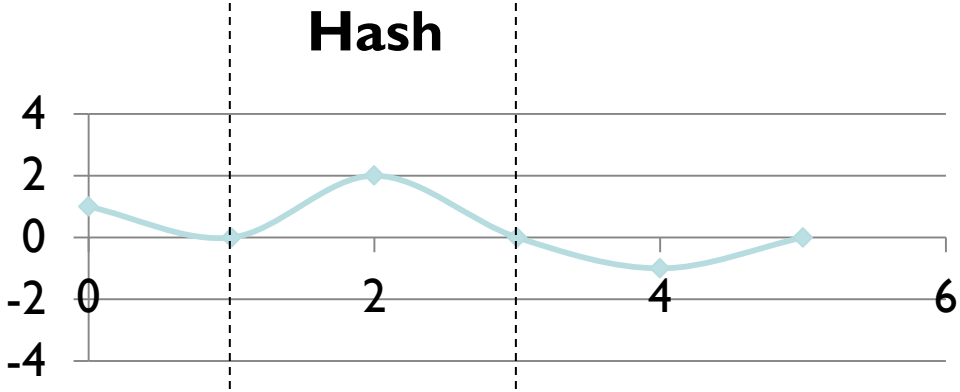


Content based Chunking

- Calculate Rabin fingerprint hash for each sliding window (16 byte)

If Hash matches a particular pattern,

Declare a chunk boundary

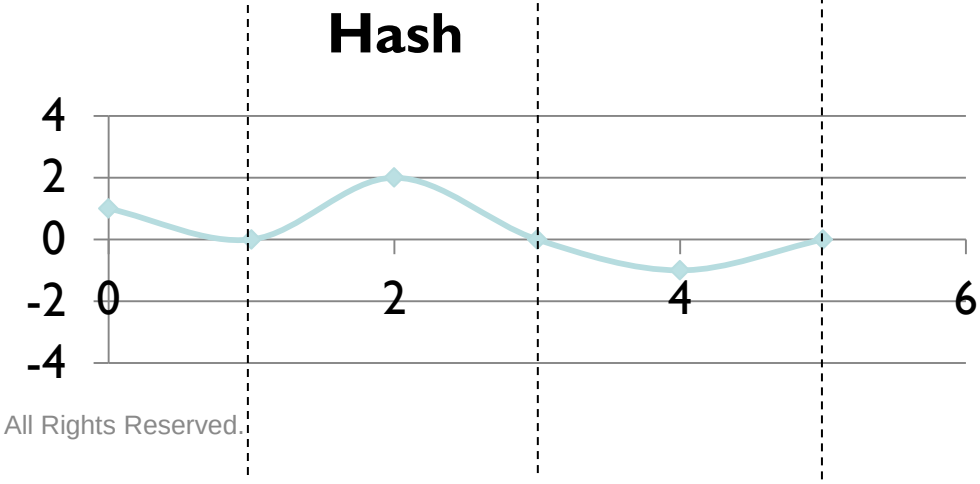


Content based Chunking

- Calculate Rabin fingerprint hash for each sliding window (16 byte)

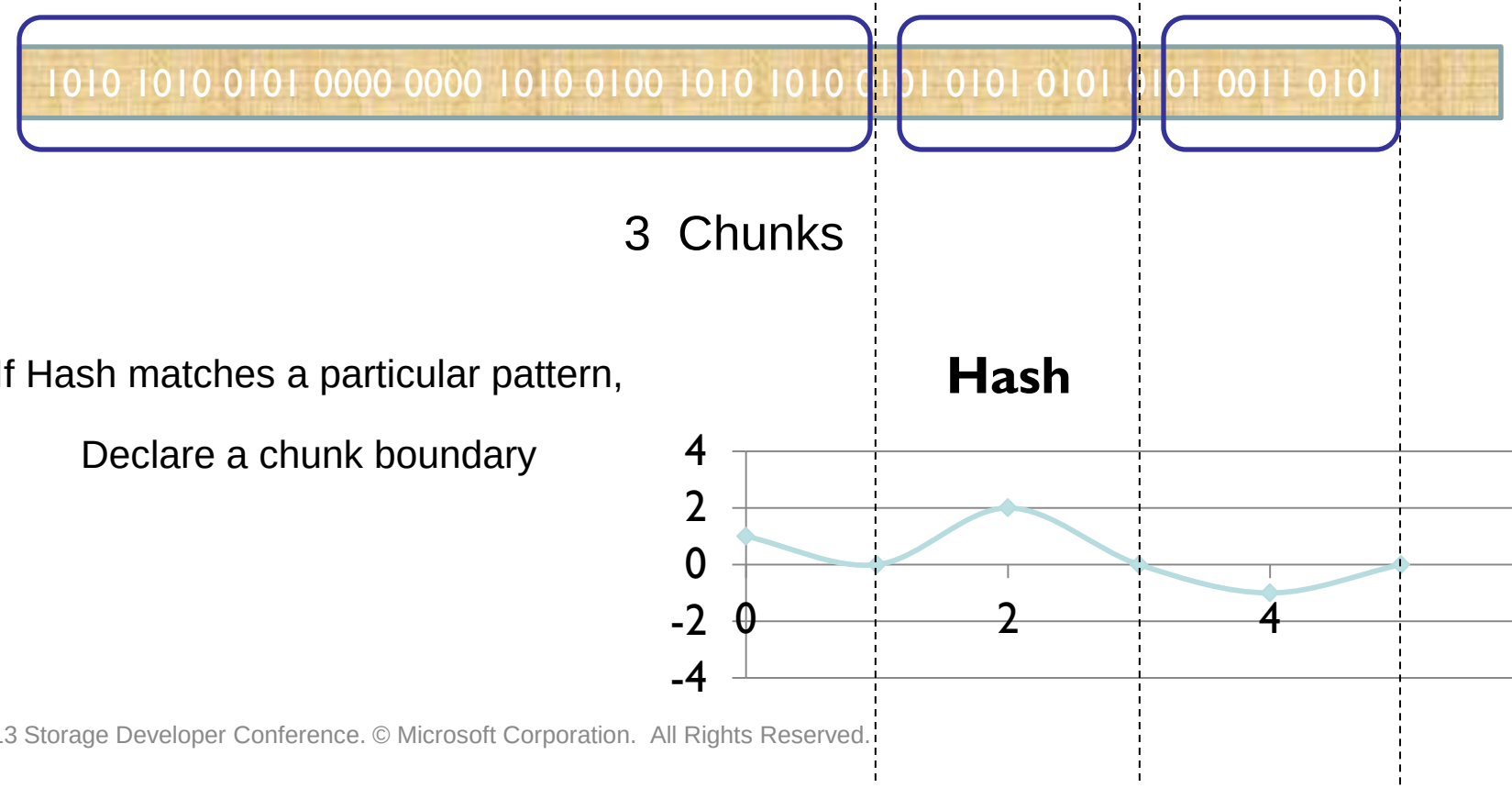
If Hash matches a particular pattern,

Declare a chunk boundary



Content based Chunking

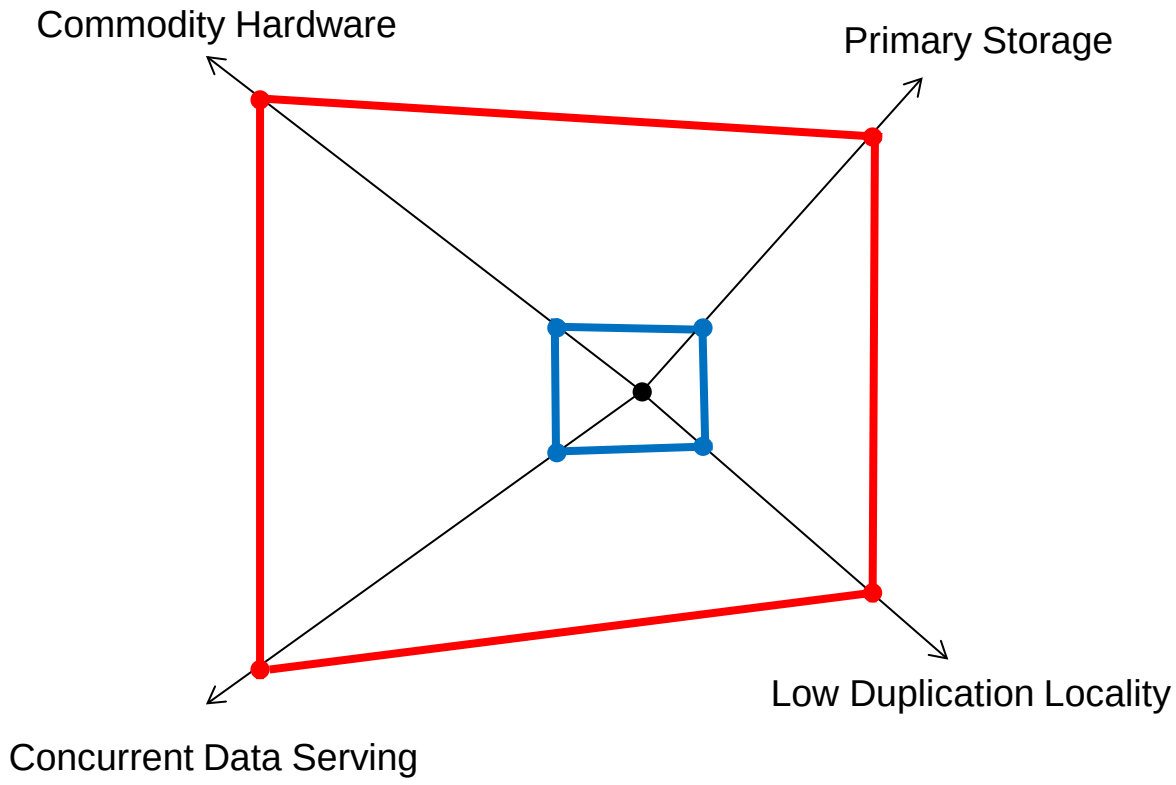
- Calculate Rabin fingerprint hash for each sliding window (16 byte)



Deduplication of Storage

- Detect and remove duplicate data in storage systems
  - Data at-rest: Storage space savings
  - Data on-wire: Network bandwidth and disk I/O savings
- High-level techniques
  - Content based chunking, detect/store unique chunks only
  - Similarity based, differential (delta) encoding
- On what type of data?
  - Backup data: mature market
  - Primary data: relatively recent interest

Deduplication Problem Space



- Backup Data dedup
- Primary Data dedup

Key Challenge

Because Windows Server needs to run well on commodity hardware and

because deduplication has to be friendly to the primary data serving workload,

this introduces unique challenges in terms of balancing

server resource usage, deduplication throughput, and deduplication space savings.

Primary Data Deduplication in Windows Server 2012

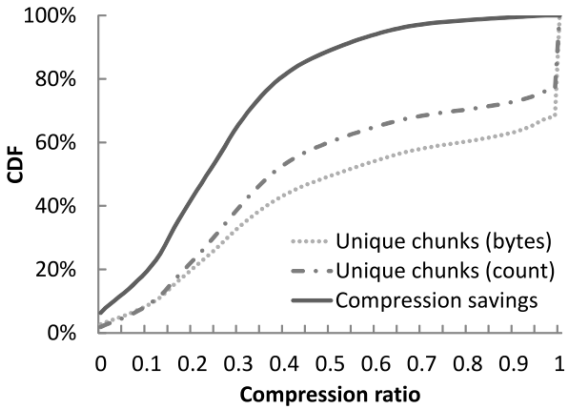
Key Design Decisions

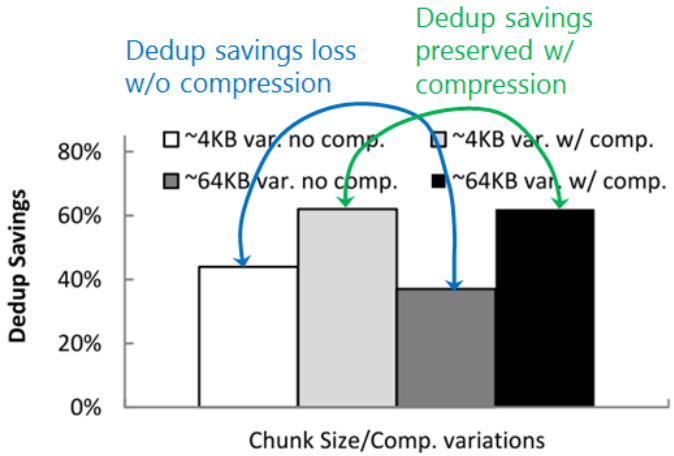
- Post-processing deduplication
  - Preserve latency/throughput of primary data access
  - Flexibility in scheduling dedup as background job on cold data
- Deduplication granularity and data chunking
  - Chunk-level: variable sized chunking, large chunk size (~80KB)
  - Modifications to Rabin fingerprint based chunking to achieve more uniform chunk size distribution
- Deduplication resource usage scaling slowly with data size
  - Reduced chunk metadata
  - RAM frugal chunk hash index
  - Data partitioning
- Crash consistent on all hardware
  - Detect, Contain, Repair, Report user and metadata corruptions

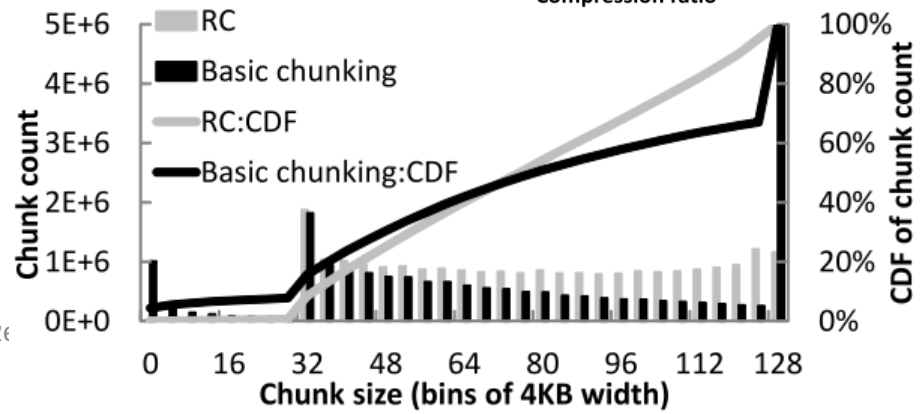
Design Decisions Driven by Data Analysis

- How do existing data dedup approaches meet Windows Server needs?
- File-level or chunk-level dedup
- Average chunk size
- To compress or not to compress chunks
- Chunking algorithm
- Index design and data partitioning technique

Dataset	Dedup Space Savings		
	File Level	Chunk Level	Gap
HF-Amsterdam	1.9%	15.2%	8x
HF-Dublin	6.7%	16.8%	2.5x
HF-Japan	4.0%	19.6%	4.9x
GFS-Japan-1	2.6%	41.1%	15.8x
GFS-Japan-2	13.7%	39.1%	2.9x
GFS-US	15.9%	36.7%	2.3x
Sharepoint	3.1%	43.8%	14.1x
VL	0.0%	92.0%	∞

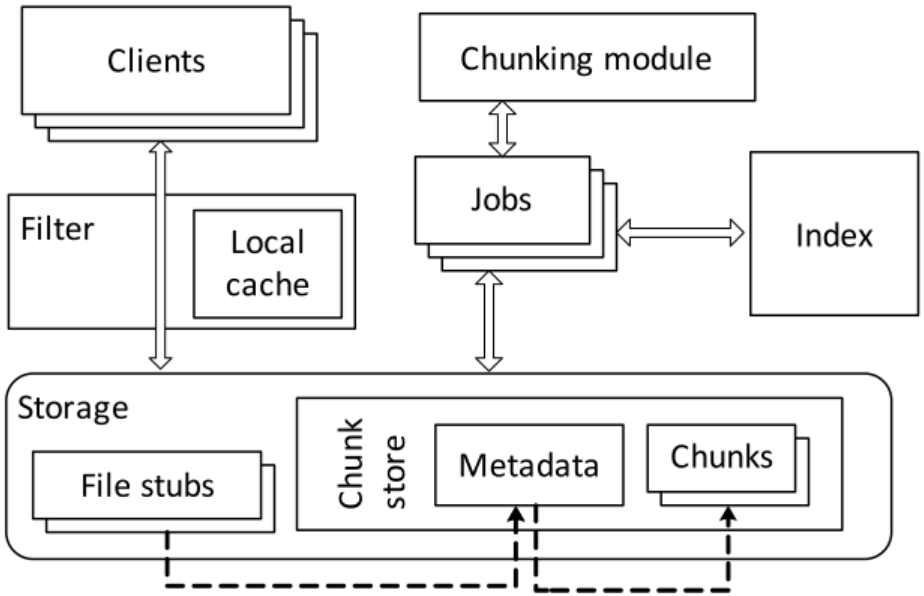






System Overview

- Deduplication pipeline
  - Post-processing
  - Data chunking
  - Index lookups + insertions
  - Chunk Store insertions
- Data path
  - Dedup filter
  - Chunk cache
  - File stub tx update
- Maintenance jobs
  - Garbage collection (in Chunk Store)
  - Data scrubbing



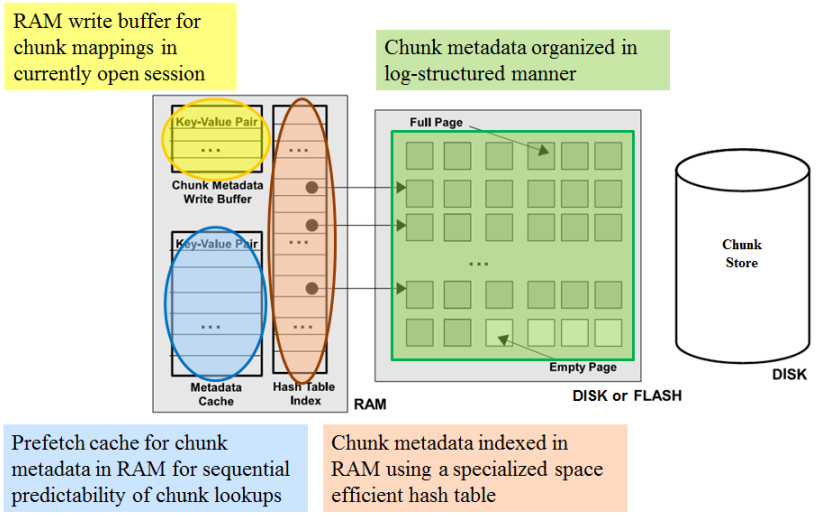
The Chunk Indexing Problem

- Chunk metadata too big to fit in RAM
  - 48 bytes per chunk: 32-byte SHA hash + 16-byte location
  - 20TB of unique data, average 64KB chunk size => 15GB of RAM
  - Not cost effective to keep all of this huge index in RAM

=> Index accesses will go to secondary storage

  - Impact on deduplication throughput

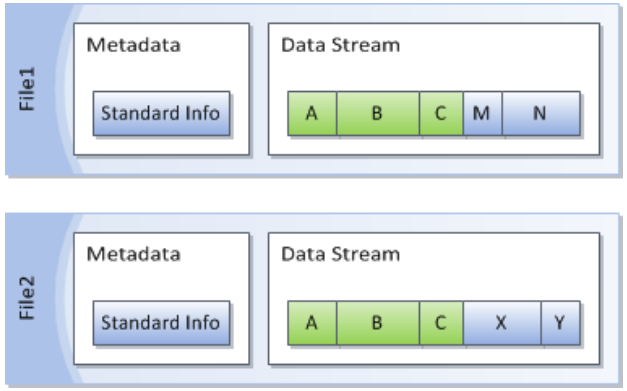
- Scaling index resource usage with data size
  - Reduced chunk metadata
  - RAM frugal chunk hash index
  - Data partitioning and reconciliation



Deduplication & on-disk structures

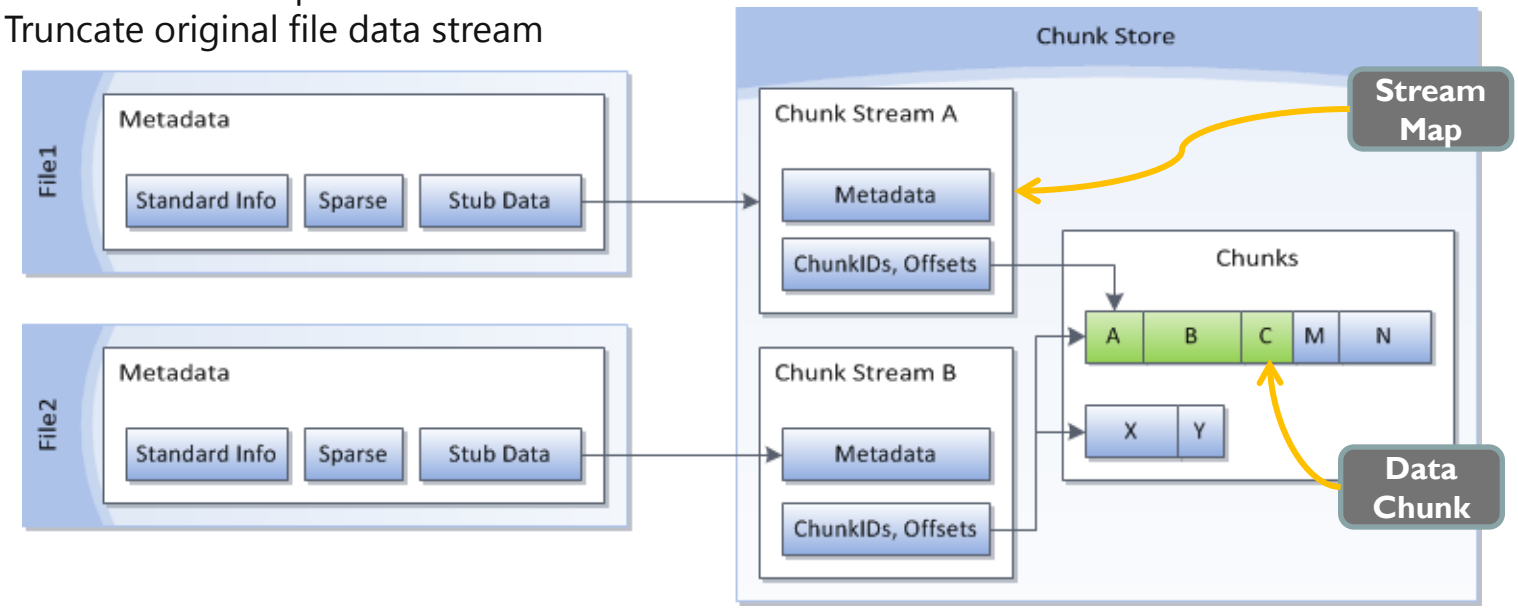
Phase I – Identify the duplicate data

1. Scan files according to policy
2. Chunk files intelligently to maximize recurring chunks
3. Identify common data chunks



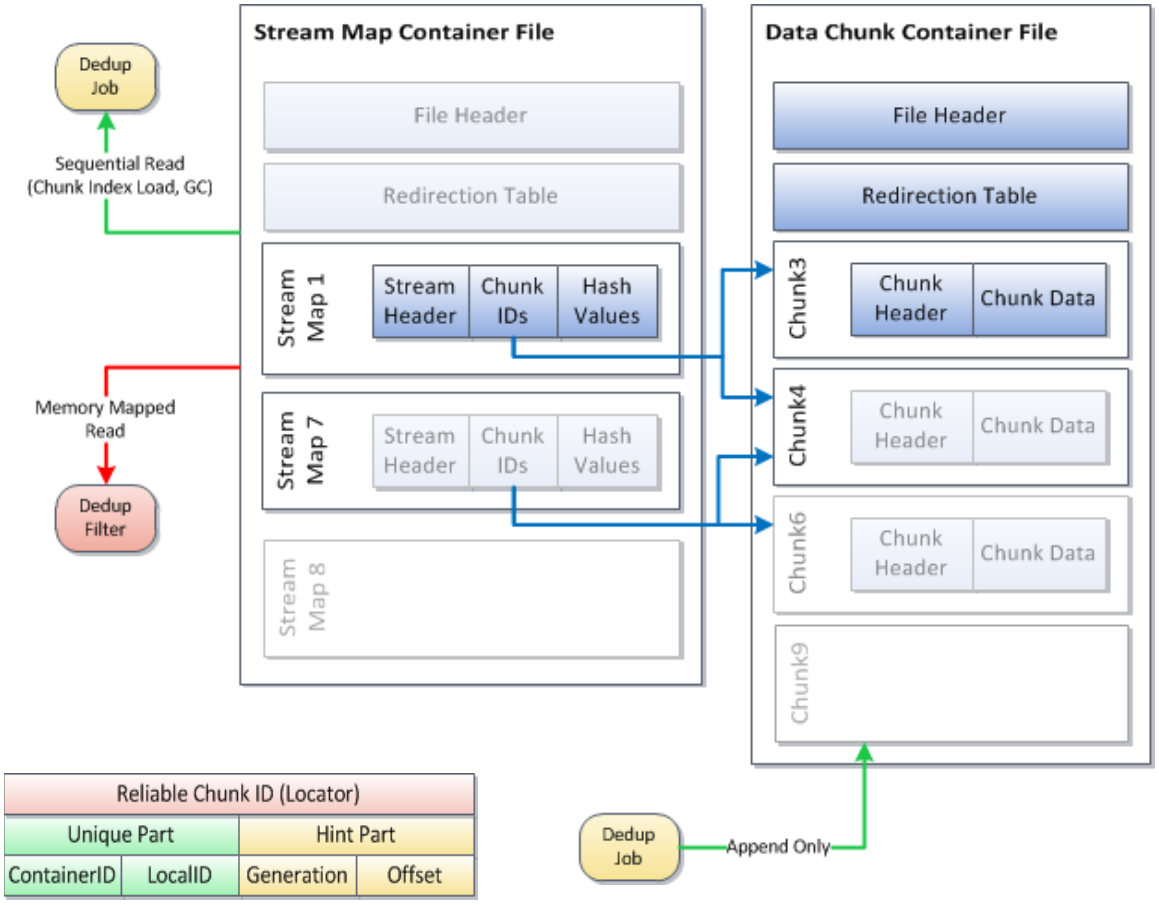
Phase II – Optimize the target files

4. Single-instance data chunks in file stream order
5. Create stream map metadata
6. Truncate original file data stream

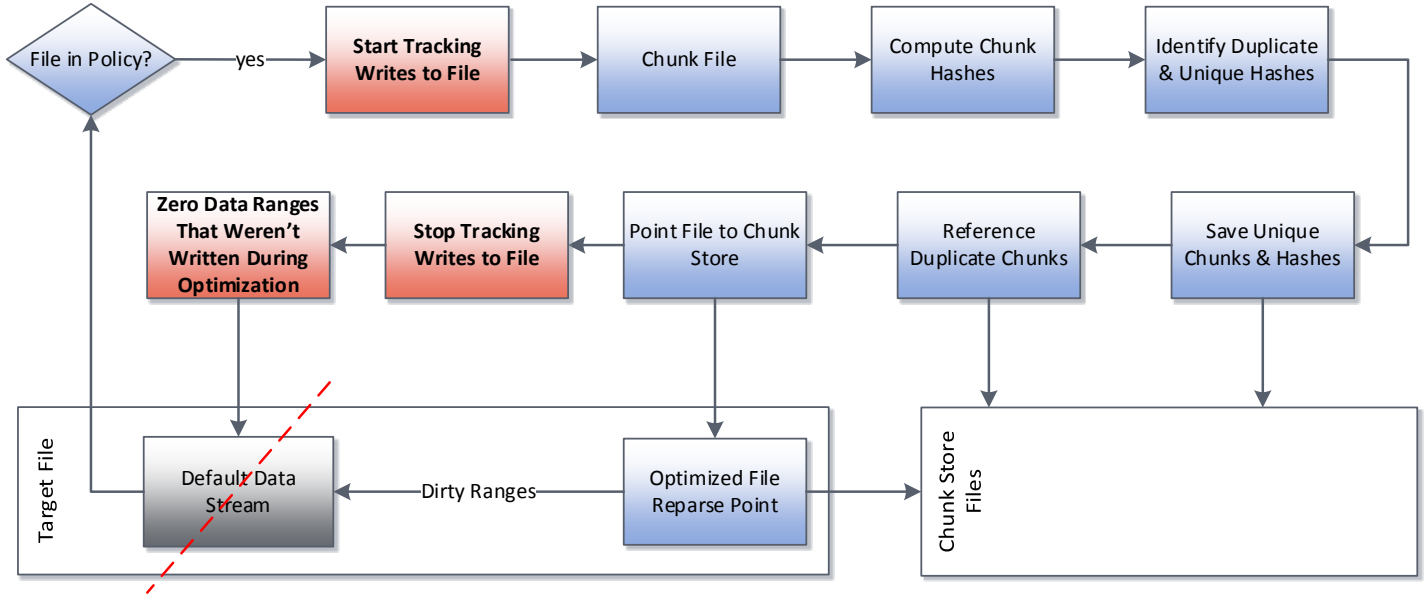


Chunk Store file layout for efficient Deduplication & Rehydration

- Container Files:
  - Scale: Avoid performance cost of large number of small files
  - Simplicity: Self-contained volume
- Optimization Performance
  - Efficient append only chunk insertion (no explicit ref-counts)
  - Efficient hash index load via sequential read
- Rehydration Performance
  - Storing in stream order reduces fragmentation
  - Reliable Chunk ID reduces seeks during rehydration



Open File Optimization



- Supports common file operations during optimization (read, write, extend, …)
- Data written during optimization will not get optimized (remains in file)
- State change is transparent to filters and apps above Dedup filter
  - Filters below Dedup will see reparse point updates and file truncation
- Optimization pipeline holds a shared file handle during optimization

IO Improvements:
Dedup Aware Caching for Reads

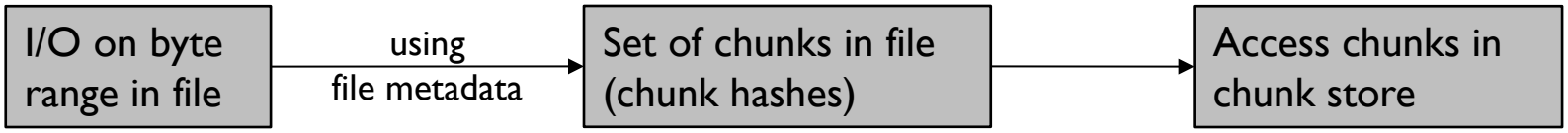
Deduplication for Virtualized Storage

- Server consolidation involves virtualization of CPU, memory, and disk resources
  - CPU virtualized using cores, shared using VM scheduling
  - Memory allocated per-VM, shared using dynamic memory
- Disk is harder to virtualize
  - Interference between VMs due to disk seek

=> Major pain point in many storage virtualization scenarios

Similar Data Access Patterns Can Help!

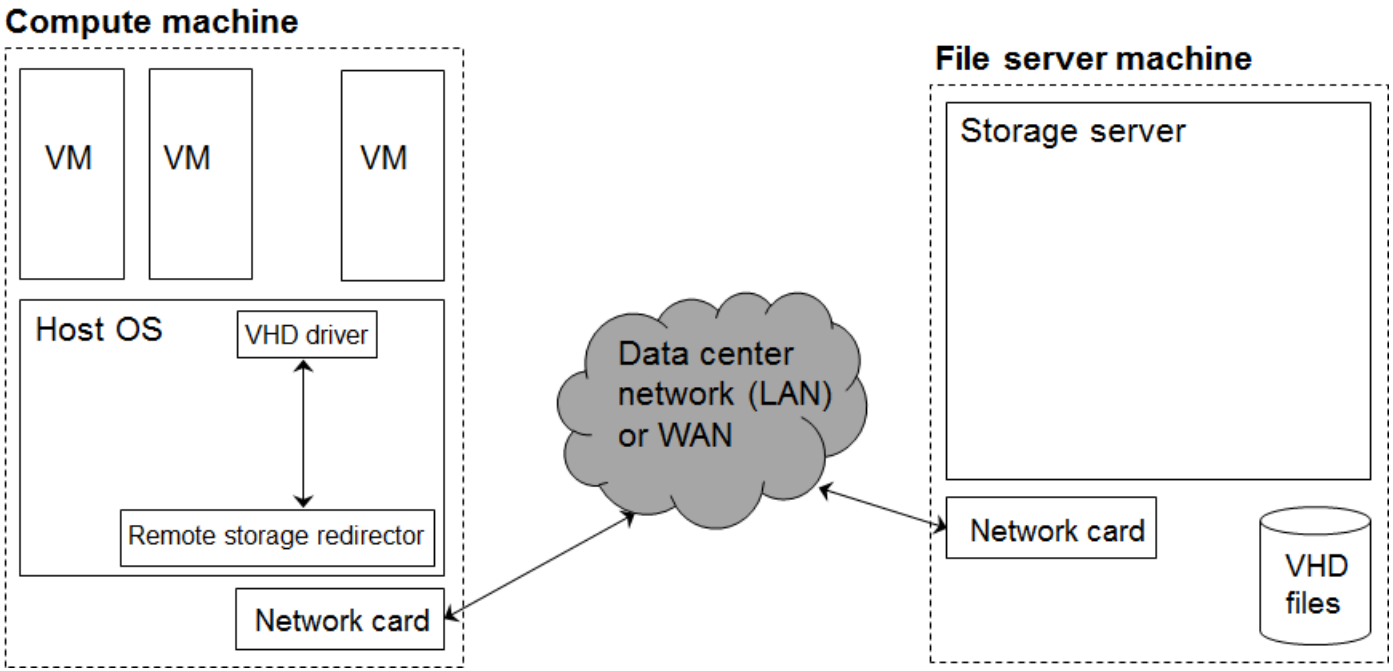
- Similar data access patterns can be exploited to mitigate either bottleneck using a two-level approach
  - Detect common data accessed across VMs
  - Place the common data on a fast access tier (RAM or SSD)
- New opportunity provided by primary data deduplication
  - Deduplication filter can detect duplicate data access
  - Not possible when accesses for identical data go to different blocks



Deduplication filter translates file byte range access to access of chunks identified by chunk hashes

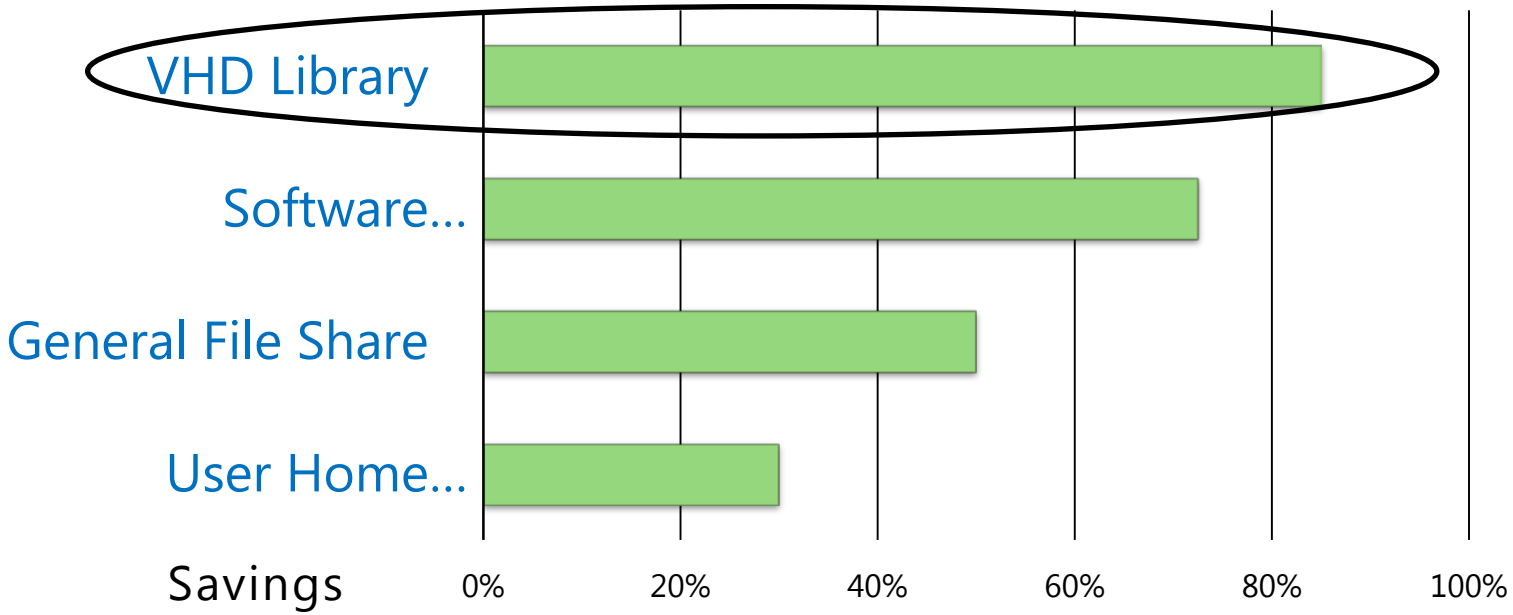
VDI and IaaS OS/Application VHDs

- Data access patterns can be exploited for
  - VDI (Virtual Desktop Infrastructure)
  - IaaS OS/Application VHDs

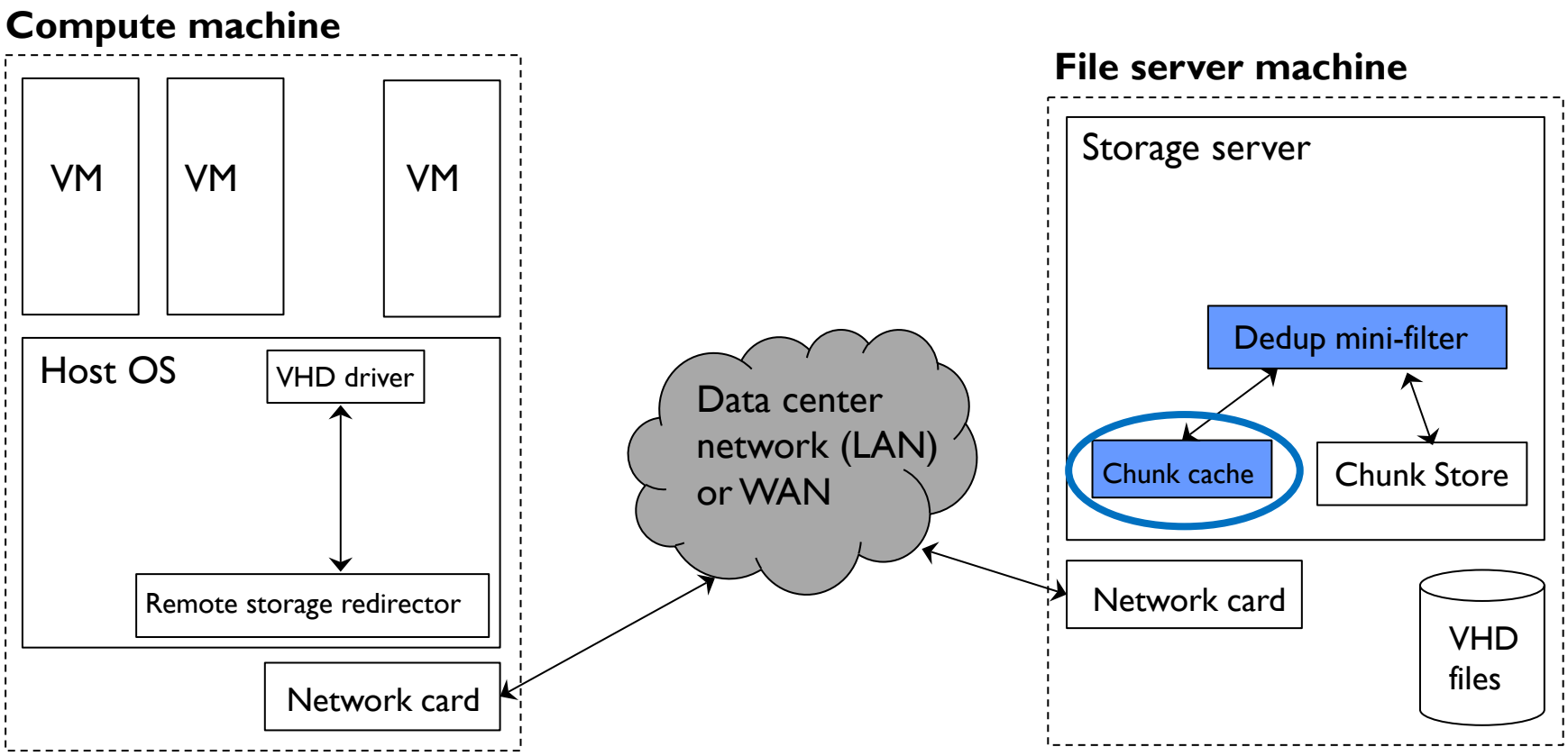


Server Dedup Value Proposition for VDI

- Bring down storage cost (which is biggest fraction of cost in VDI)
  - Commodity storage is 30% less expensive compared to SANs, another 75-95% savings because of deduplication
  - But, commodity storage has reduced IOPS



Mitigate disk IOPS bottleneck …



Deduplication Space Savings

Disk IOPs savings

VDI Boot/Login Access Pattern Analysis

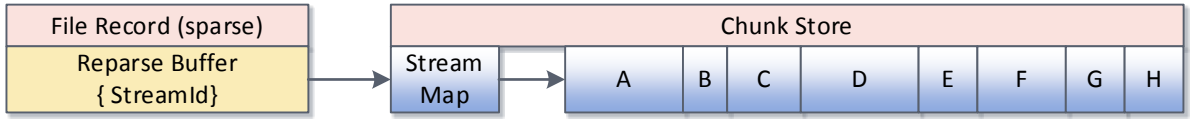
- n VMs booting in quick succession, each VM accesses
  - c amount of common data
  - ε amount of data unique to itself (on average)
- Total data accessed = $n*(c + \varepsilon)$
- Total unique data accessed = $c + n*\varepsilon$
- Illustrative example: $n = 40$, $\varepsilon = 0.1*c$
  - Fraction of data access that is unique = 11%

  => 10x reduction in bytes read from disk

IO Improvements:
Granular partial recall for Writes

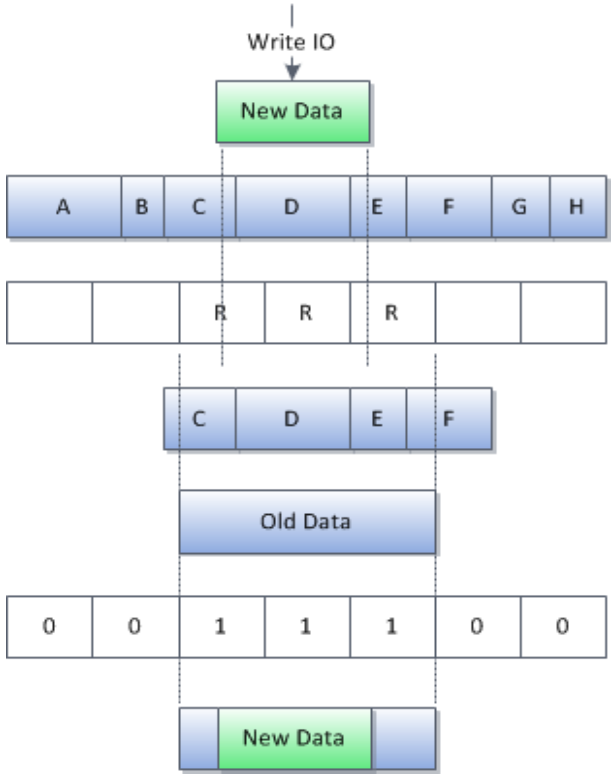
Deduplicated File Write Path

1 Pre write File Layout



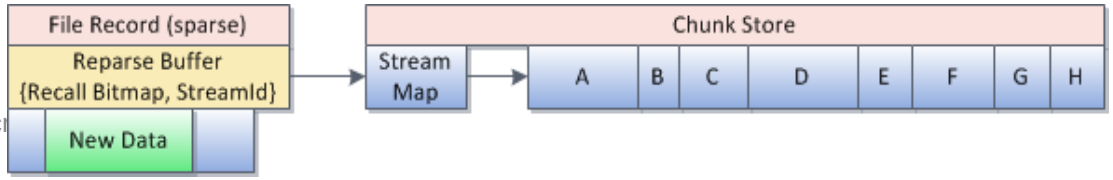
2 Write flow

1. Hold User Write IO
2. Deduped Chunk View
3. Fixed Recall Boundaries
4. Read Underlying Chunks
5. Write Recall-Aligned Data and **Flush File**
6. Write Reparse Buffer with Recall Bitmap
7. Release User Write (no explicit file flush)



- Reduce latency for small writes to large files (e.g. OS image patching scenario)
- Incremental dedup will later optimize the new data
- Recall granularity grows with file size
- GC cleans up unreferenced chunks (chunk "D" in example)

3 Post Write File Layout

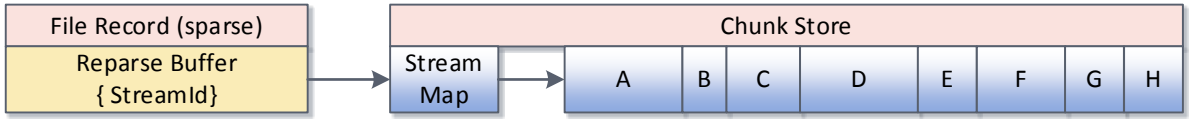


Improving Write Path for VHD Files

- VHD files see lots of small random writes
  - Recall at smaller granularity (512-byte to 4KB) can help
- VHD files see writes that are mostly sector aligned
  - Partial recall from chunk store not needed
  - For general writes, no recall is need for aligned portion and at most two recalls for unaligned portion at start/end of write region
- Smaller recall bitmap granularity (512-byte to 4KB)
  - Hierarchical (multi-level) representation for efficiency
  - Root level bitmap in file reparse point, rest in separate file

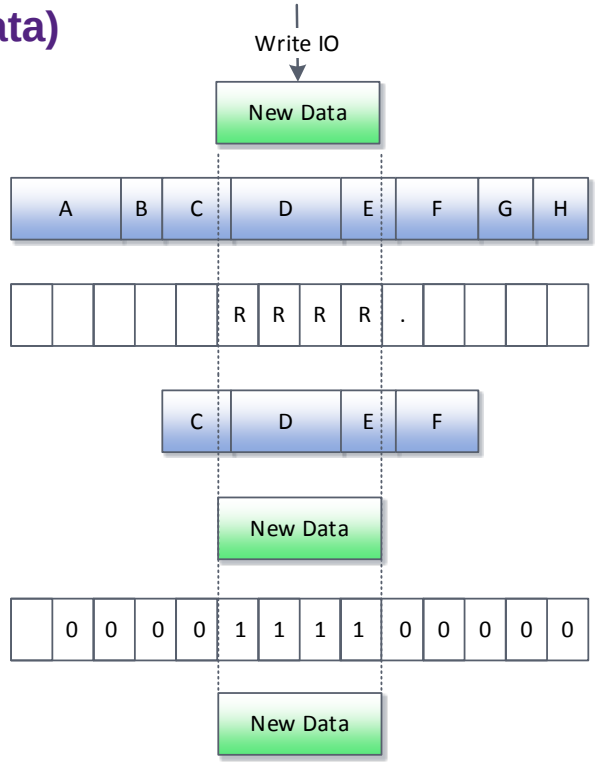
Granular Partial Recall for Efficient File Writes

1 Pre write File Layout



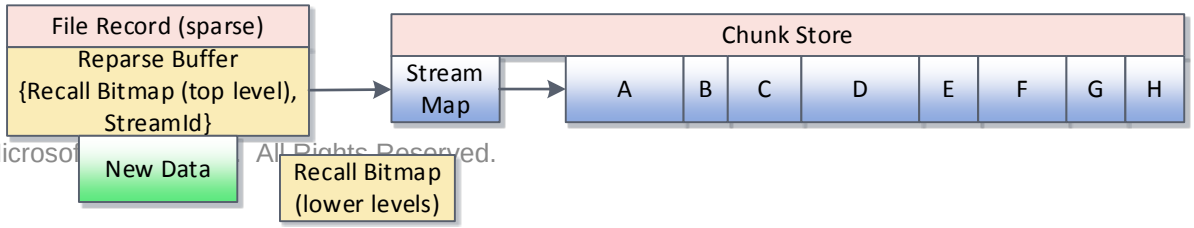
2 Write flow (sector-aligned data)

1. User write written to file
2. If user write is non-sector aligned: Recall before/after user write to align it to 512-byte
3. **Flush user file**
4. Update and flush recall bitmap file
5. Update reparse point

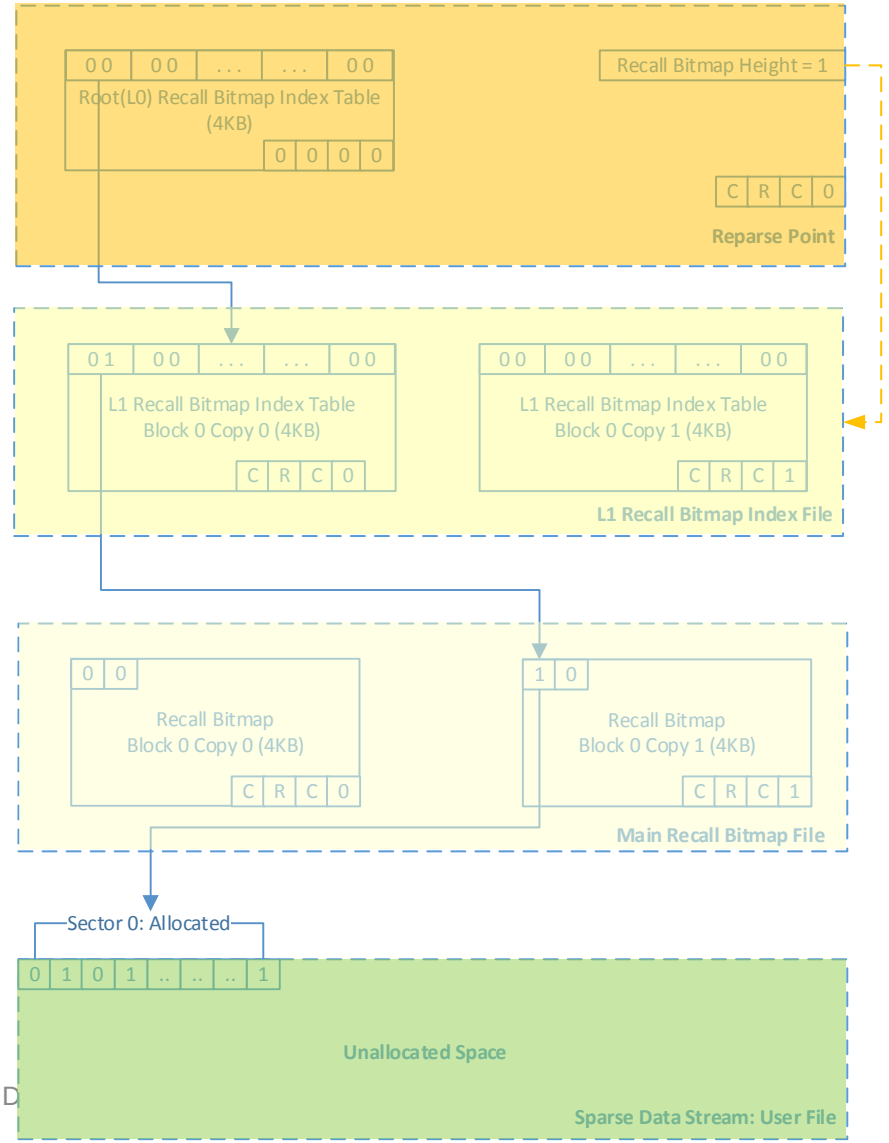


- Reduce latency for small writes to large files (e.g., Live VHD)
- Incremental dedup on open file (Live VHD) will later optimize new data
- Fine recall granularity (512-byte to 4KB)
- GC cleans up unreferenced chunks (chunk "D" in example)

3 Post Write File Layout



Multi-Level Recall Bitmap



00: completely in chunk store
11: completely in sparse file
01 | **10**: indicates which copy of next level block

HW Acceleration: Assist with scale, perf and avoiding workload-impact on the Live-VHD

Hardware acceleration: hashing

- Problem: dedup can be CPU-intensive. Bottlenecks:
  - SHA hash computation (for all chunks)
  - Compression (only for new chunks)
- Main focus: optimize hashing algorithms
  - Use more efficient hashing algorithms (multi-buffering)
  - Offload SHA hashing in hardware (FPGA, GPU)

HW. acceleration: hashing (results)

- CPU-based hashing optimizations
  - SHA-256 (default)
  - SHA-512 (default)
  - MB-SHA256
    - Multi-buffering

Platform	Hashing algorithm	Cycles/ Byte	Improvement
i7-2600 3.4GHz Win7 client	SHA-256	23.5	1x (base)
	SHA-512	15.1	1.56x
	MB-SHA256	6.8	3.46x
E5-2630 2.3GHz Win8 server	SHA-256	16.4	1x (base)
	SHA-512	10.5	1.56x
	MB-SHA256	6.0	2.73x
Q6600 2.4GHz Win8 server	SHA-256	17.8	1x (base)
	SHA-512	11.3	1.58x
	MB-SHA256	7.9	2.25x

Hw. acceleration: hashing (results)

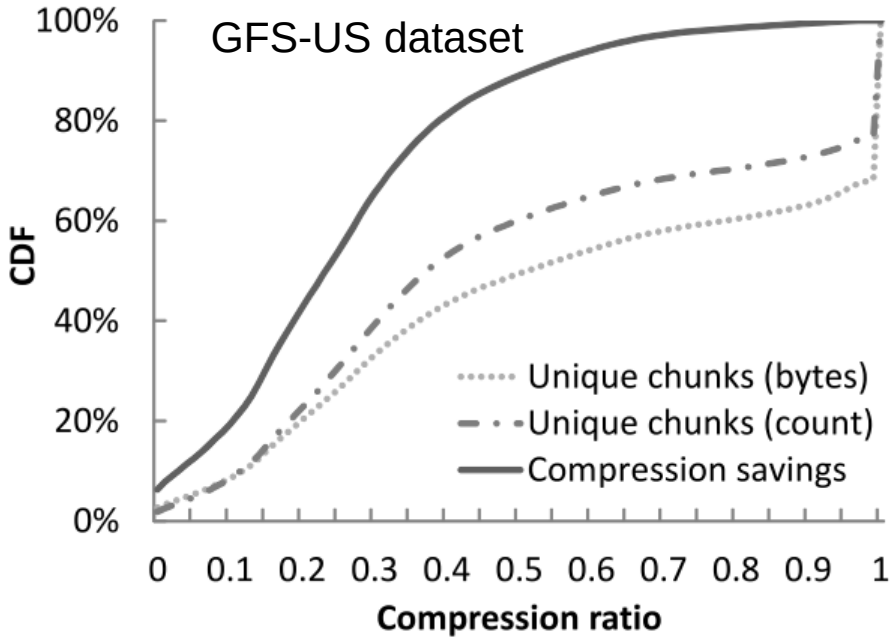
- Offloading hardware acceleration
  - GPU-based SHA-512 computation

Batch size (chunks)	512	1024	2048	4096	8192
Throughput (MB/S)	560	870	1160	1390	1410
Improvement over CPU sha-512	3.7x	5.8x	7.7x	9.3x	9.4x

  - FPGA-based SHA-512 computation
    - Virtex-5 FPGA, unrolled pipeline
    - Preliminary prototype: 1500 MB/s
    - Main bottleneck: buffering costs

Selective compression

- Compression/decompression can have a significant perf impact
- Compression savings is skewed
  - 50% of unique chunks responsible for 86% of compression savings
  - 31% of chunks do not compress at all
- Solution: selective compression
  - Reduces cost of compression for large fraction of chunks
    - While preserving most of compression savings
  - Reduces decompression costs (reduce CPU pressure during heavy reads)
    - Also: use a cache for decompressed data (important for hotspots)
  - Heuristics for determining which chunks should not be compressed



Selective compression (cont.)

- Results

Dataset	Compression time (seconds)			Deduplication time (seconds)			Deduplication savings (percentage)	
	No entropy	With entropy	Improvement	No entropy	With entropy	Improvement	No entropy	With entropy
Audio-video	67.4	7.8	88.4%	700	637	9.0%	3.6%	3.6%
Debuggers	26.1	26.1	0%	119	119	0%	61.3%	61.0%
Images	238.5	15.1	93.7%	1800	1560	13.3%	11.5%	11.5%
Pdf	36.7	10.1	72.5%	243	211	13.2%	16.5%	16.1%
Sharepoint	133.5	58.4	56.2%	991	930	6.2%	36.1%	35.9%
Vhd-data	417	283	32.1%	2040	1915	6.1%	57.1%	56.8%
Vhd-OS	116.1	103.1	11.2%	489	480	1.9%	64.7%	64.7%
GPFS	14976	10763	28.1%	73704	69271	6.0%	58.9%	58.8%
OSVHD	3306	3046	7.9%	15792	15379	2.6%	85.9%	85.8%

Summary

- Primary data deduplication in Windows Server 2012
  - Extending the problem frontier in multiple dimensions w.r.t. backup data deduplication
  - Design decisions driven by large scale data analysis
  - Primary workload friendly
  - Robust and reliable on commodity hardware
- Enhancements in Windows Server Blue for Virtualized Storage
  - Open file optimization
  - Read path optimization using dedup aware caching
  - Write path optimization using small recall bitmap granularity to exploit sector aligned writes
  - Hardware acceleration for hashing and compression