Reliable and Efficient Metadata Storage and Indexing Using NVRAM

Ethan L. Miller
Kevin Greenan • Andrew Leung • Darrell Long • Avani Wildani
(and others)
Storage Systems Research Center
University of California, Santa Cruz

Goals

- Metadata is often the limiting factor in file system performance
  - More than 50% of FS operations are metadata
  - Metadata operations are difficult to pipeline
- Metadata is small!
  - Often fewer than 50 bytes of information per file
  - Billions of files → only tens of gigabytes of metadata
- Goal: use NVRAM to make metadata efficient
- Challenges
  - Performance
    - Block-based reads (for NAND flash)
    - Block-based writes (for NAND & NOR flash)
  - Efficiency: need to keep index structures small
  - Reliability: can't afford to lose metadata!

Metadata & indexing: the basics

- Metadata: information ***about*** files
  - File ownership, size, age, type, usage patterns
    - Tends to be relatively small
  - File content: indexing *terms*
    - Can be very large, depending on the files
    - Can be text-based or derived in other ways using a transducer
- Basic metadata is very small
  - Inode is less than 50 bytes, excluding location information
- Indexing metadata can be larger
  - Many terms per file
  - Very aggressive compression is possible

Metadata performance

- Metadata performance is critical to file system performance
  - 50–70% (or more) of requests are for metadata
  - Operations are small and often random
    - Difficult to pipeline (prefetch) metadata operations
- Metadata and search are increasing in importance
  - Larger file systems → more difficult to find data
  - Larger file systems → new approaches to organizing data
    - Link-based file systems
    - View-based file systems
- Metadata is critical to continued file system growth

Designs for NVRAM-based metadata

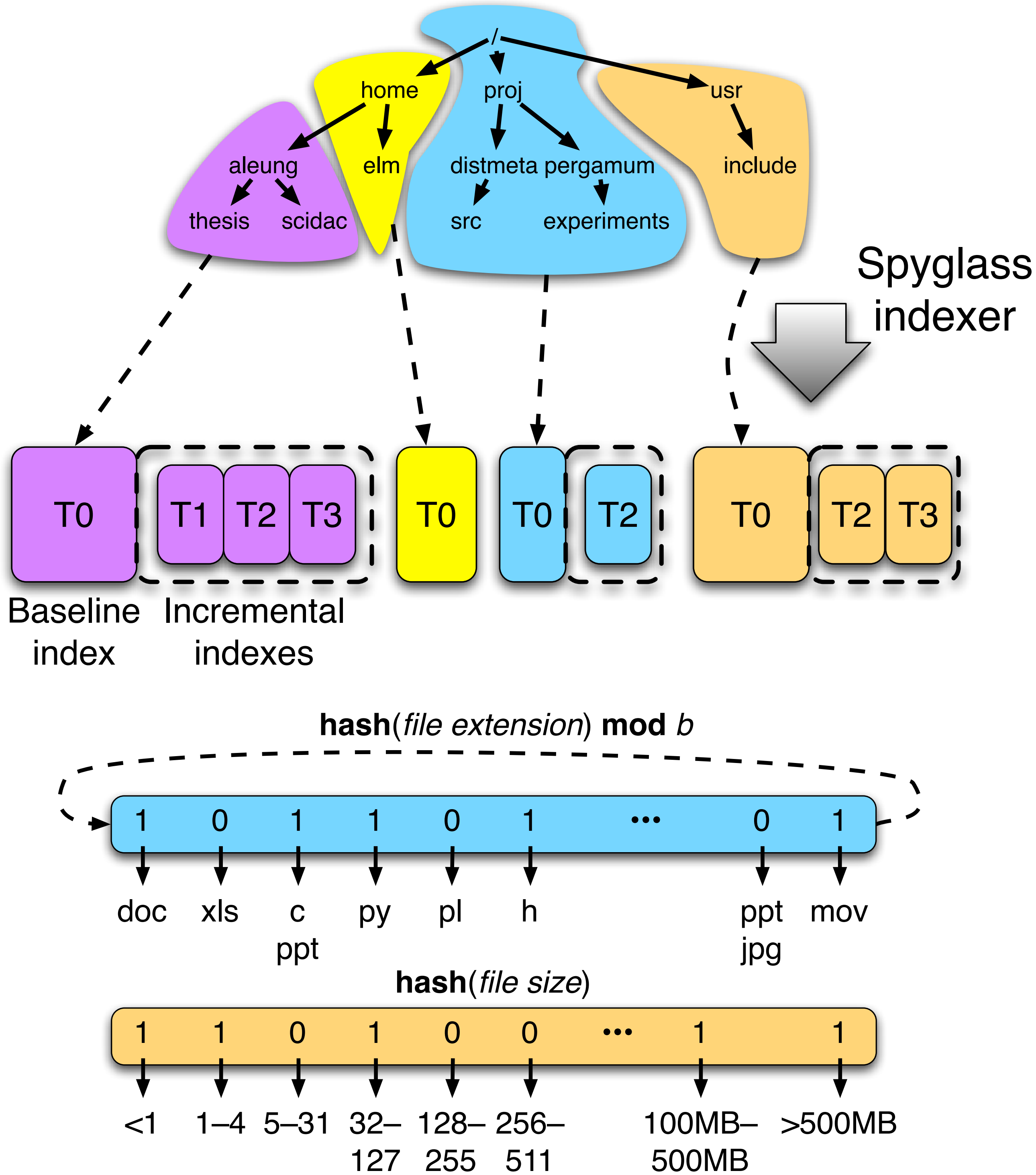
- Metadata is a perfect fit for NVRAM
  - Metadata is small, typically 0.1–1% of data space
    - Can be much lower for large scientific computing
  - Read access to metadata is mostly random
  - Updates can be logged: sequential writes
- Explore three issues
  - Metadata on flash: block access for writes and (for NAND) reads
  - Metadata on word-accessible NVRAM: data structures
  - Reliability
    - Hardware errors
    - Software errors

Why not use NVRAM-based databases?

- Why not use a standard off-the-shelf DB in NVRAM?
  - Advantage: code is already written and works well!
  - Drawbacks
    - Databases aren't very space-efficient: expensive on NVRAM
    - Databases aren't optimized well for FS metadata
    - Databases are very complex: lots of locking necessary for disk-based DBs
- Potential gains for task-specific metadata indexes in NVRAM
  - Higher performance: our experiments show 1–3 orders of improvement
  - Lower space consumption
  - Lower software complexity: no need to wait for disk-based I/O

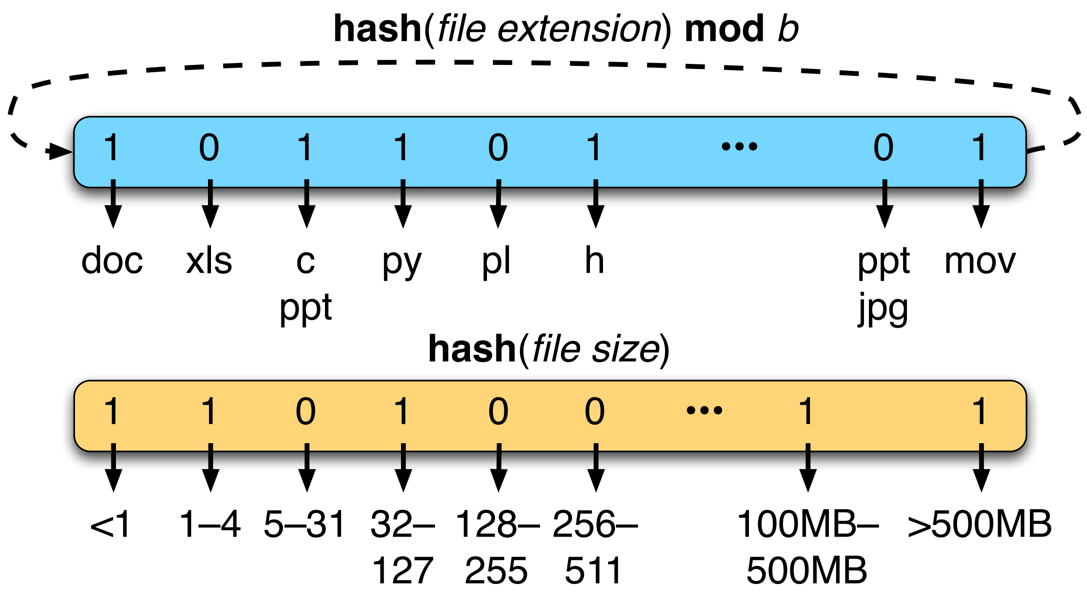
Metadata on flash: block access

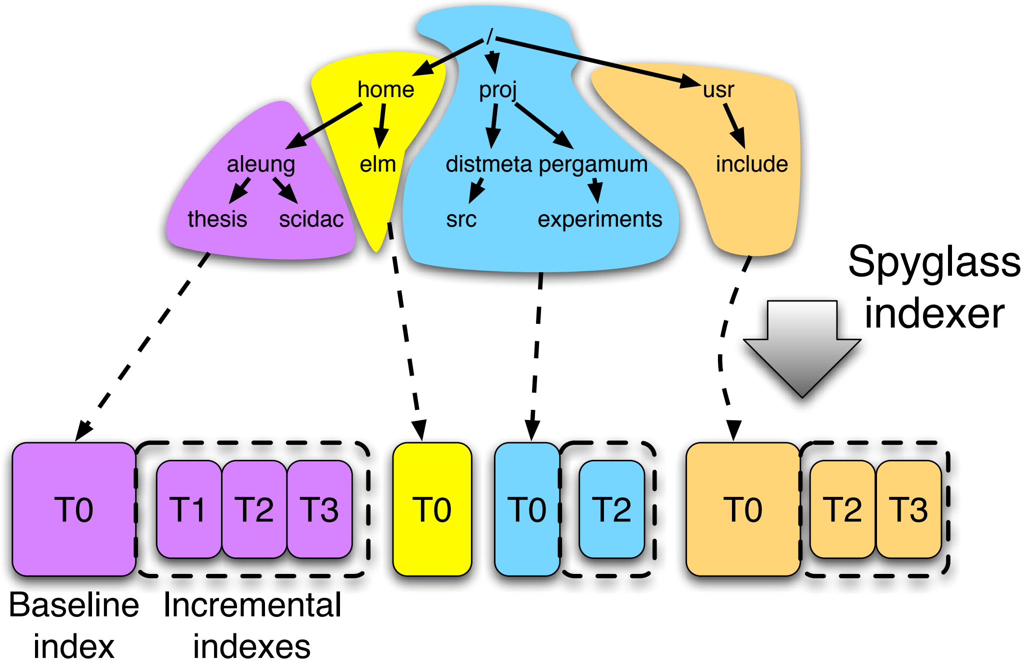
- Goal: use block-based NVRAM to hold metadata
- Partition file system hierarchy by subtree
  - Each subtree is an independent subindex
- Summarize contents
  - Rule out entire subindexes that don't have files that satisfy query
- Log incremental changes
  - Rebuild index when there are "enough" changes
- Integrity is much easier
  - Rebuild subindex, not entire index



Using partitioned indexes

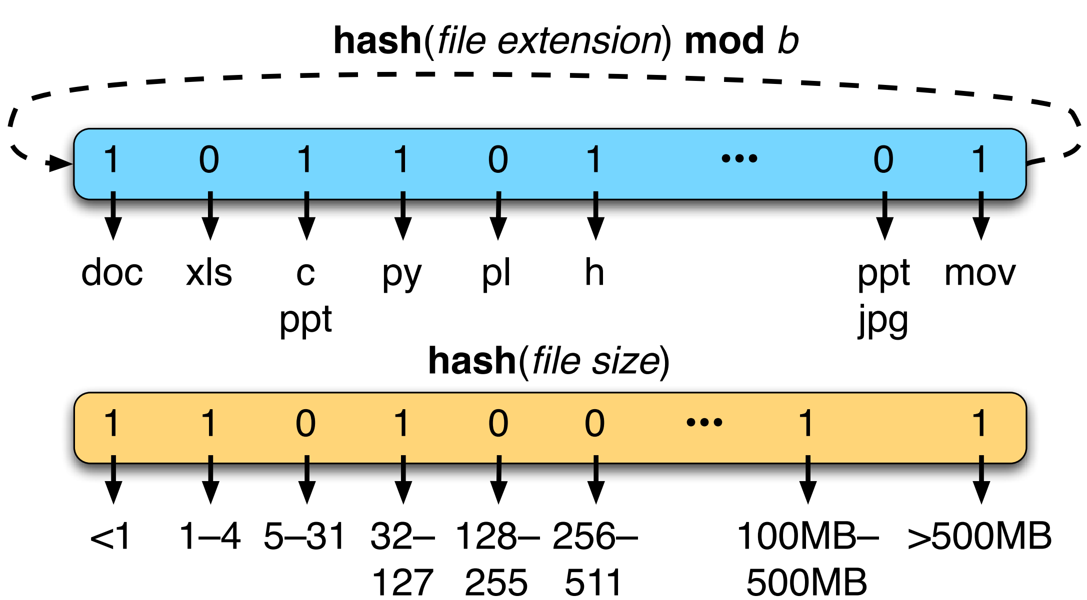
- Updates and rebuilds are more localized
  - Updates are appended to the "base" index
    - Subsequent searches look in both base and incremental indexes
  - Periodic "rebuilding" of base index consolidates updates
- Fewer reads / traversals of actual indexes
  - Only potentially relevant subindexes are scanned
  - Individual subindexes cover about 100K files in 3 MB
    - Index is a k-d tree (for now)
    - Search may only need to read a part of relevant subindexes

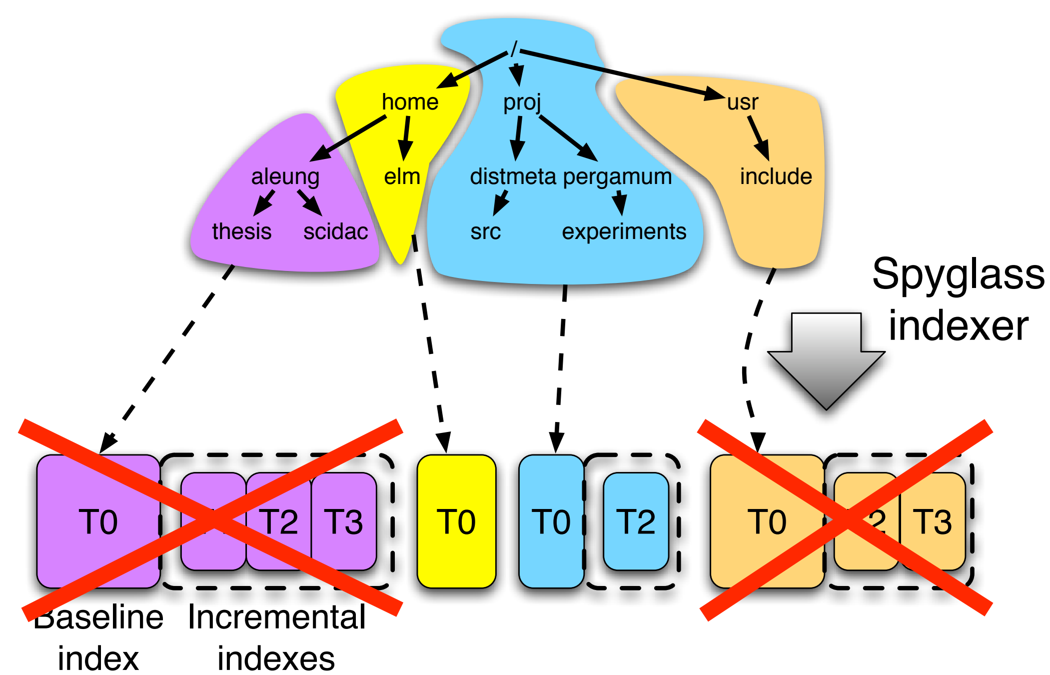




Using partitioned indexes

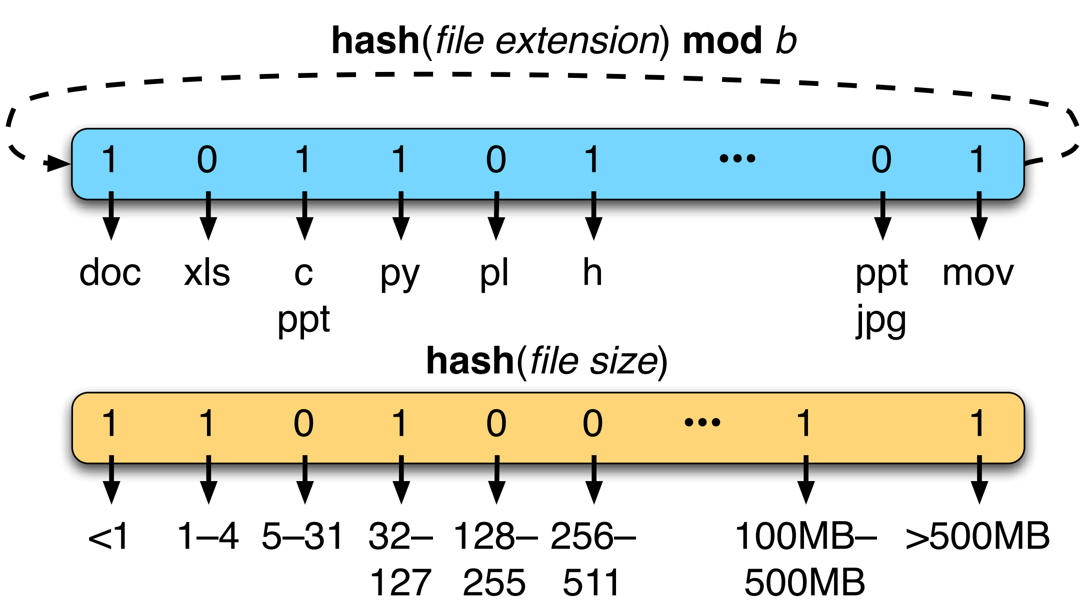
- Updates and rebuilds are more localized
  - Updates are appended to the "base" index
    - Subsequent searches look in both base and incremental indexes
  - Periodic "rebuilding" of base index consolidates updates
- Fewer reads / traversals of actual indexes
  - Only potentially relevant subindexes are scanned
  - Individual subindexes cover about 100K files in 3 MB
    - Index is a k-d tree (for now)
    - Search may only need to read a part of relevant subindexes

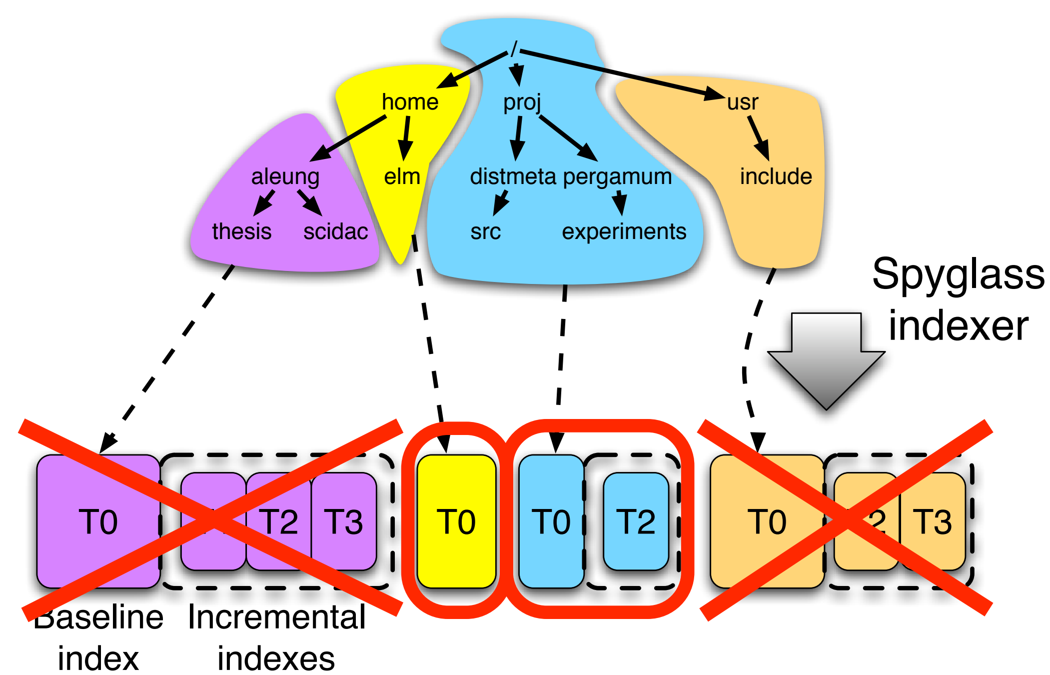




Using partitioned indexes

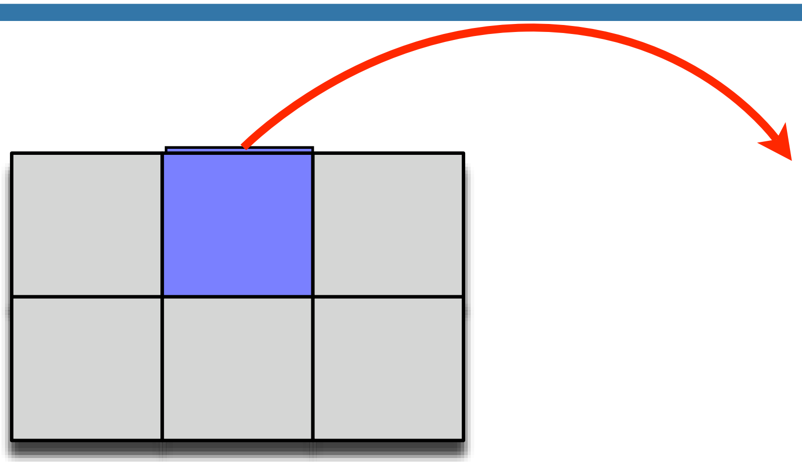
- Updates and rebuilds are more localized
  - Updates are appended to the "base" index
    - Subsequent searches look in both base and incremental indexes
  - Periodic "rebuilding" of base index consolidates updates
- Fewer reads / traversals of actual indexes
  - Only potentially relevant subindexes are scanned
  - Individual subindexes cover about 100K files in 3 MB
    - Index is a k-d tree (for now)
    - Search may only need to read a part of relevant subindexes





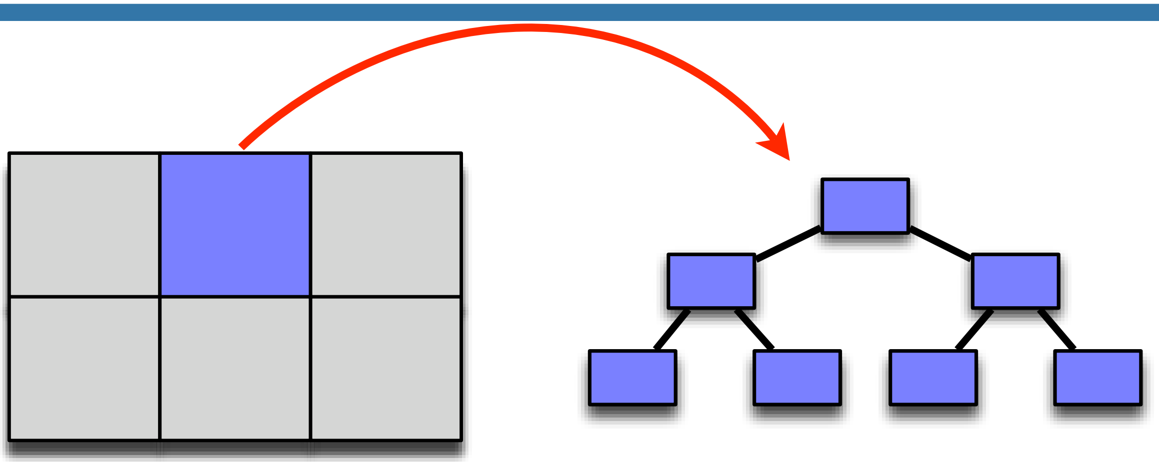
Byte-addressability for reads

- NAND flash: block reads and writes
- NOR flash: word addressable for reads
  - How does this affect metadata accesses?
- Current design: decompress entire subindex to search it
  - Slower, but more space efficient
- New design: search in compressed tree
  - Less space efficient?
  - Faster?



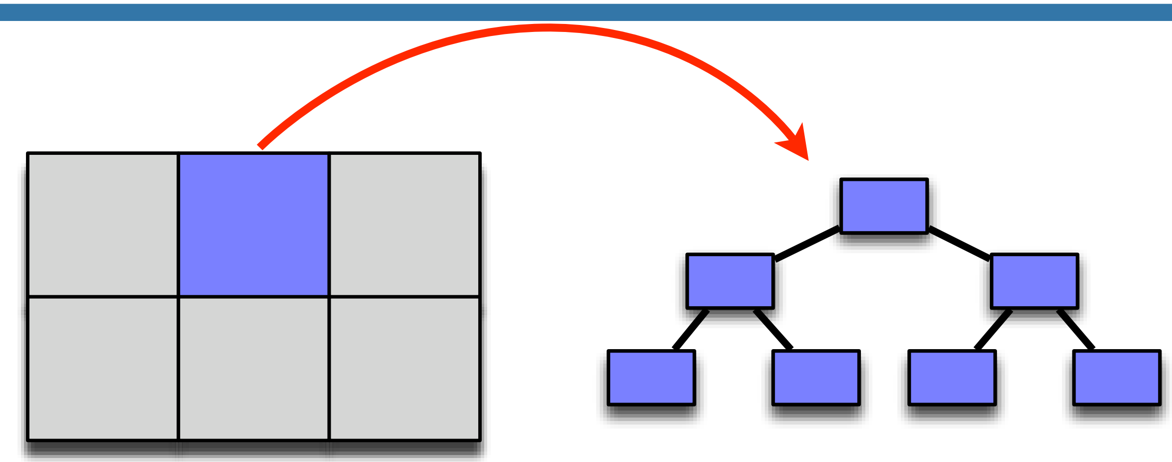
Byte-addressability for reads

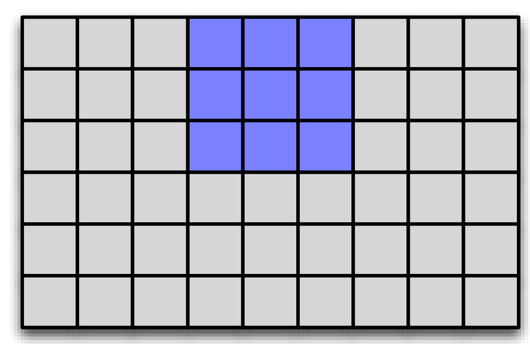
- NAND flash: block reads and writes
- NOR flash: word addressable for reads
  - How does this affect metadata accesses?
- Current design: decompress entire subindex to search it
  - Slower, but more space efficient
- New design: search in compressed tree
  - Less space efficient?
  - Faster?



Byte-addressability for reads

- NAND flash: block reads and writes
- NOR flash: word addressable for reads
  - How does this affect metadata accesses?
- Current design: decompress entire subindex to search it
  - Slower, but more space efficient
- New design: search in compressed tree
  - Less space efficient?
  - Faster?





Storing metadata efficiently

- Compress values aggressively
  - Compress numbers
  - Use tables for frequent values
  - Store times as differences from base times
- Save space on log entries
  - Only log modified fields
  - Point to earlier full copy of metadata
  - Modifications can be chained...
    - Start looking at most recent update & follow chain
  - Eventually need to "clean up" metadata
- These techniques work far better with byte addressability

00000000000000000000000000000001

Storing metadata efficiently

- Compress values aggressively
  - Compress numbers
  - Use tables for frequent values
  - Store times as differences from base times
- Save space on log entries
  - Only log modified fields
  - Point to earlier full copy of metadata
  - Modifications can be chained…
    - Start looking at most recent update & follow chain
  - Eventually need to “clean up” metadata
- These techniques work far better with byte addressability

00000000000000000000000000000001

↓

0

Storing metadata efficiently

- Compress values aggressively
  - Compress numbers
  - Use tables for frequent values
  - Store times as differences from base times
- Save space on log entries
  - Only log modified fields
  - Point to earlier full copy of metadata
  - Modifications can be chained…
    - Start looking at most recent update & follow chain
  - Eventually need to "clean up" metadata
- These techniques work far better with byte addressability

00000000000000000000000000000001

↓

0

UID	GID	PERM	#
elm	faculty	rwxr-x---	0
kmgreen	grads	rwx------	1
elm	faculty	rw-r--r--	2
elm	ssrc	rwxrwx---	3

Storing metadata efficiently

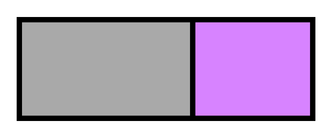
- Compress values aggressively
  - Compress numbers
  - Use tables for frequent values
  - Store times as differences from base times
- Save space on log entries
  - Only log modified fields
  - Point to earlier full copy of metadata
  - Modifications can be chained…
    - Start looking at most recent update & follow chain
  - Eventually need to “clean up” metadata
- These techniques work far better with byte addressability

00000000000000000000000000000001

↓

0

UID	GID	PERM	#
elm	faculty	rwxr-x---	0
kmgreen	grads	rwx------	1
elm	faculty	rw-r--r--	2
elm	ssrc	rwxrwx---	3



Storing metadata efficiently

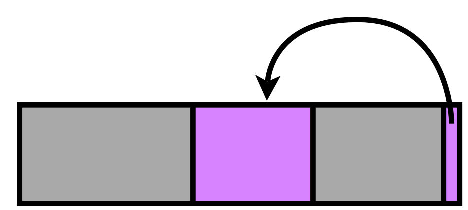
- Compress values aggressively
  - Compress numbers
  - Use tables for frequent values
  - Store times as differences from base times
- Save space on log entries
  - Only log modified fields
  - Point to earlier full copy of metadata
  - Modifications can be chained…
    - Start looking at most recent update & follow chain
  - Eventually need to "clean up" metadata
- These techniques work far better with byte addressability

00000000000000000000000000000001

↓

0

UID	GID	PERM	#
elm	faculty	rwxr-x---	0
kmgreen	grads	rwx------	1
elm	faculty	rw-r--r--	2
elm	ssrc	rwxrwx---	3



Storing metadata efficiently

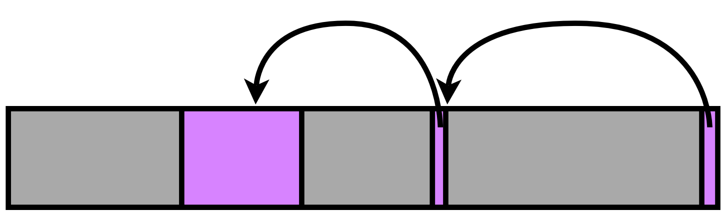
- Compress values aggressively
  - Compress numbers
  - Use tables for frequent values
  - Store times as differences from base times
- Save space on log entries
  - Only log modified fields
  - Point to earlier full copy of metadata
  - Modifications can be chained...
    - Start looking at most recent update & follow chain
  - Eventually need to "clean up" metadata
- These techniques work far better with byte addressability

00000000000000000000000000000001

↓

0

UID	GID	PERM	#
elm	faculty	rwxr-x---	0
kmgreen	grads	rwx------	1
elm	faculty	rw-r--r--	2
elm	ssrc	rwxrwx---	3



Storing metadata efficiently

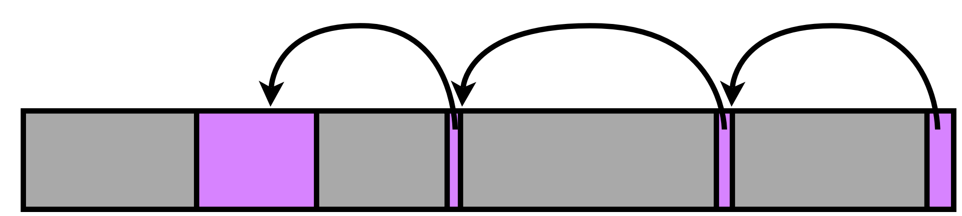
- Compress values aggressively
  - Compress numbers
  - Use tables for frequent values
  - Store times as differences from base times
- Save space on log entries
  - Only log modified fields
  - Point to earlier full copy of metadata
  - Modifications can be chained...
    - Start looking at most recent update & follow chain
  - Eventually need to "clean up" metadata
- These techniques work far better with byte addressability

00000000000000000000000000000001

↓

0

UID	GID	PERM	#
elm	faculty	rwxr-x---	0
kmgreen	grads	rwx------	1
elm	faculty	rw-r--r--	2
elm	ssrc	rwxrwx---	3



Term indexes in NVRAM

- Many standard techniques for indexing in memory
  - Inverted indexing most common
  - Many optimizations require byte-addressability
    - Skip lists
- Use partitioned term indexes?
  - Only need to scan a few term lists anyway
  - May be helpful for multi-term searches
    - Only if terms are uncommon
- Alternate approaches
  - Redesign skip lists for block-based access
  - Instead of interleaving, keep multiple lists joined together
- Byte-addressable NVRAM
  - No need to modify algorithms: existing approaches are append-only
  - Still may need partitioning, and definitely need reliability!

NVRAM for archival storage metadata

- NVRAM has advantages for archival storage
  - Low power consumption
  - Avoids spinup of primary media (disk)
  - Highly scalable: associate relatively small NVRAM with each media
- Searches
  - Broadcast request to all nodes
  - Use earlier strategies to narrow down nodes to search
- Scrubbing and reliability
  - Cache disk checksums in NVRAM
  - Allow consistency checking between nodes without having all disks spun up simultaneously

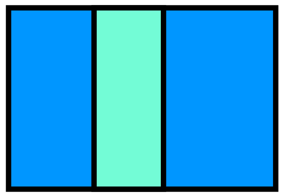
Reliability for metadata in NVRAM

- Durability of recently-completed operations
  - Metadata operations can't complete until committed to stable store
  - Need to guarantee both efficient storage and fast commit
- Reliability of data already written
  - Guard against corruption
- Correctness of data written to NVRAM
  - Guard against file system bugs!
  - Double-check (or more) writes that are sent to NVRAM

Reliable NVRAM: approach

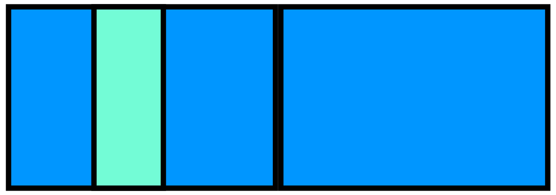
- Updates to indexing data structures done by append
  - Avoids corrupting old (but valid) structures
  - Reduces lock contention: reader/writer locks only on current block
- Appended information includes
  - Index updates
  - Checksums (signatures / hashes)
  - Erasure code blocks
- Appended information checked by separate process
  - No overwrite in place!
  - Easier to recover from errors

Reliable NVRAM: approach



- Updates to indexing data structures done by append
  - Avoids corrupting old (but valid) structures
  - Reduces lock contention: reader/writer locks only on current block
- Appended information includes
  - Index updates
  - Checksums (signatures / hashes)
  - Erasure code blocks
- Appended information checked by separate process
  - No overwrite in place!
  - Easier to recover from errors

Reliable NVRAM: approach

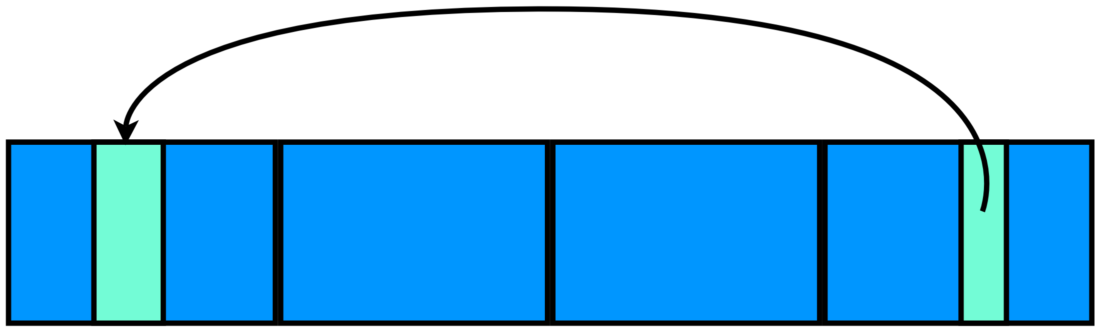


- Updates to indexing data structures done by append
  - Avoids corrupting old (but valid) structures
  - Reduces lock contention: reader/writer locks only on current block
- Appended information includes
  - Index updates
  - Checksums (signatures / hashes)
  - Erasure code blocks
- Appended information checked by separate process
  - No overwrite in place!
  - Easier to recover from errors

Reliable NVRAM: approach

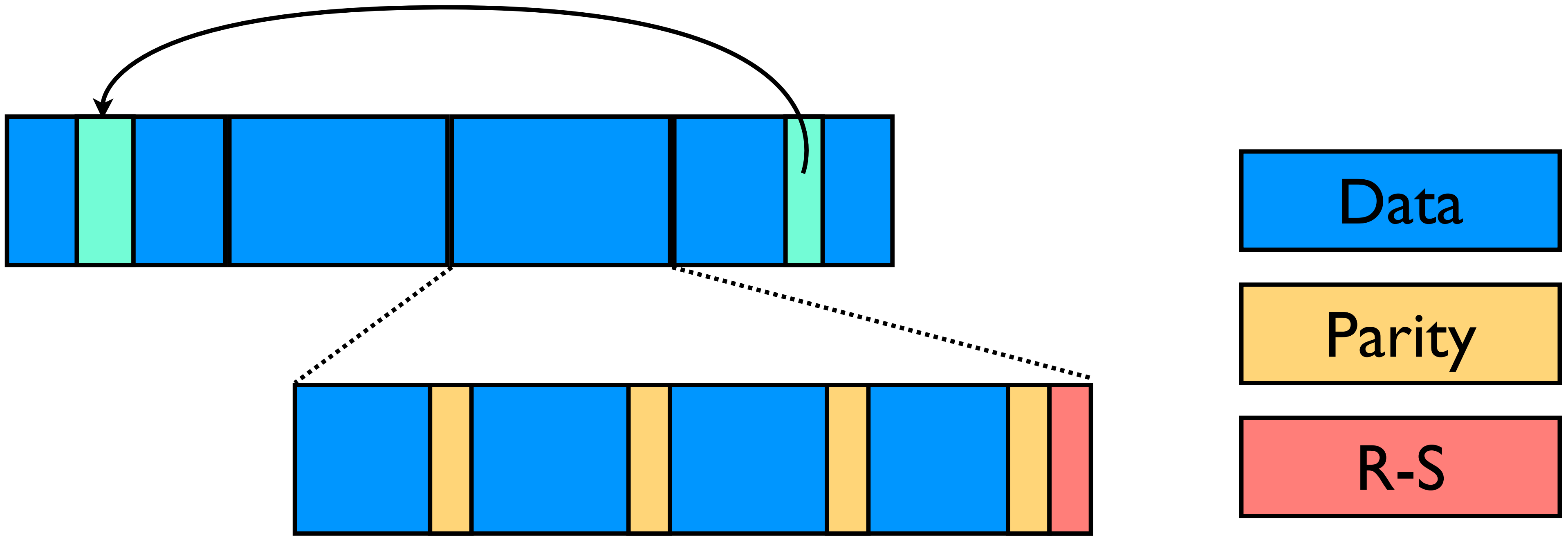
- Updates to indexing data structures done by append
  - Avoids corrupting old (but valid) structures
  - Reduces lock contention: reader/writer locks only on current block
- Appended information includes
  - Index updates
  - Checksums (signatures / hashes)
  - Erasure code blocks
- Appended information checked by separate process
  - No overwrite in place!
  - Easier to recover from errors

Reliable NVRAM: approach



- Updates to indexing data structures done by append
  - Avoids corrupting old (but valid) structures
  - Reduces lock contention: reader/writer locks only on current block
- Appended information includes
  - Index updates
  - Checksums (signatures / hashes)
  - Erasure code blocks
- Appended information checked by separate process
  - No overwrite in place!
  - Easier to recover from errors

Reliable NVRAM: approach



- Updates to indexing data structures done by append
  - Avoids corrupting old (but valid) structures
  - Reduces lock contention: reader/writer locks only on current block
- Appended information includes
  - Index updates
  - Checksums (signatures / hashes)
  - Erasure code blocks
- Appended information checked by separate process
  - No overwrite in place!
  - Easier to recover from errors

Storage Systems Research Center pdsi

Baskin Engineering UC SANTA CRUZ

Reliable byte-addressable NVRAM

- Byte-addressable NVRAM allows for small in-place modifications
  - Problem: potential for data structure corruption
  - Problem: wear leveling (PRAM)
- Use log structure for byte-addressable NVRAM
  - Smaller appends ➙ faster metadata commit
  - Similar approach to block-based: write followed by correctness check to handle file system errors
- Protect NVRAM with signature/hash and erasure code
  - Signature/hash detects corruption
  - Erasure code corrects it

Conclusions

- Reliable, efficient metadata for large-scale storage is a good fit for NVRAM
  - Small data size
  - High IOPS
  - Ability to guard against hardware & software errors
- NVRAM is suitable for both high-performance and archival storage
- Approach changes slightly as NVRAM moves from block-based to byte-addressable accesses

Questions?