The What, Why and How of the Pure Storage Enterprise Flash Array

Ethan L. Miller

(and a cast of dozens at Pure Storage)
Enterprise storage: $30B market built on disk

- Key players: EMC, NetApp, HP, etc.
- Dominated by spinning disk
- Wait, isn’t flash the new hotness?
  - Drives today’s smartphones, cameras, USB drives, even laptops
  - Common to speed up desktops by installing SSDs
- Why can’t we swap flash into today’s disk array?
  - Current software systems are optimized for disk
  - Flash and disk are very different
Flash vs. disk

**Disk**
- Moving parts: mechanical limitations
- Locality matters: faster to read data nearby on the disk
- Read and write have symmetric performance
- Data can be overwritten in place

**Flash**
- No moving parts: all electronic
  - Much faster
  - Much more reliable
- Reads are locality-independent and 100x faster than disk

<table>
<thead>
<tr>
<th></th>
<th>Consumer Disk</th>
<th>Enterprise Disk</th>
<th>SSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>$/GB</td>
<td>$0.05</td>
<td>$0.50</td>
<td>$1</td>
</tr>
<tr>
<td>Capacity</td>
<td>~3–4 TB</td>
<td>~0.2–2 TB</td>
<td>~500 GB</td>
</tr>
<tr>
<td>Sequential read/write</td>
<td>150 MB/s</td>
<td>200 MB/s</td>
<td>~400 MB/s</td>
</tr>
<tr>
<td>IOPS (4K)</td>
<td>80–120</td>
<td>200</td>
<td>11.5K/40K</td>
</tr>
<tr>
<td>Read latency</td>
<td>12 ms</td>
<td>6 ms</td>
<td>0.15ms</td>
</tr>
<tr>
<td>Write latency</td>
<td>13 ms</td>
<td>6 ms</td>
<td>0.05ms</td>
</tr>
<tr>
<td>Worst case latency</td>
<td>15ms</td>
<td>10ms</td>
<td>100ms</td>
</tr>
</tbody>
</table>
Why not simply replace disk with flash in an array?

• Traditional arrays are optimized for disk
  • Attempt to reduce seek distances: irrelevant for flash
  • Overwrite in place: make flash work harder
  • Mix reads and writes: flash often performs worse on such workloads

• Resulting flash array would be too expensive
  • Leverage deduplication: keep only one copy of data even if it’s written to multiple locations
  • Dedup and compression can reduce cost and increase performance

• Different failure characteristics
  • Flash can burn out quickly
  • Entire disks typically fail more often than flash drives
  • Flash drives lose more individual sectors
Flash isn’t all rosy...

- Data is written in pages, erased in blocks
  - Page: 4KB
  - Erase block: 64–256 pages
  - Need a flash translation layer to map logical addresses to physical locations
  - Typically, manage flash with log-structured storage system

- Limited number of erase cycles
  - Multi-Level Cells (MLC): \(\sim3–5,000\) erase cycles
  - Single-Level Cells (SLC): \(\sim100,000\) cycles (but more expensive)

- Performance quirks
  - Unpredictable read and write response times
Pure Storage goals
Pure Storage goals

• 10x faster...
  • Provide consistent I/O latency for read and write: 200–400K IOPS
  • Leverage flash characteristics: random reads & sequential writes
  • Optimal performance independent of volume config and data layout
Pure Storage goals

• 10x faster...
  • Provide consistent I/O latency for read and write: 200–400K IOPS
  • Leverage flash characteristics: random reads & sequential writes
  • Optimal performance independent of volume config and data layout

• ... at the same cost ...
  • Use commodity MLC SSDs: scalable to 10s–100s of terabytes
  • Leverage aggressive compression and deduplication
Pure Storage goals

• 10x faster...
  • Provide consistent I/O latency for read and write: 200–400K IOPS
  • Leverage flash characteristics: random reads & sequential writes
  • Optimal performance independent of volume config and data layout

• ... at the same cost ...
  • Use commodity MLC SSDs: scalable to 10s–100s of terabytes
  • Leverage aggressive compression and deduplication

• ... always available ...
  • Run 24x7x365: non-disruptive upgrades
  • Protect against device failure and corruption
Pure Storage goals

• **10x faster...**
  - Provide consistent I/O latency for read and write: 200–400K IOPS
  - Leverage flash characteristics: random reads & sequential writes
  - Optimal performance independent of volume config and data layout

• **... at the same cost ...**
  - Use commodity MLC SSDs: scalable to 10s–100s of terabytes
  - Leverage aggressive compression and deduplication

• **... always available ...**
  - Run 24x7x365: non-disruptive upgrades
  - Protect against device failure and corruption

• **... and with standard storage features like snapshots**
Pure Storage approaches

- **Guiding principles**
  - *Never* change information, only make it outdated (append-only structures): this makes operations idempotent
  - Keep *approximate* answers: resolve when needed (more reads, fewer writes)

- **Primary structures**
  - Data and metadata logs: stored in segments
    - Append-only data structure: new entries occlude older entries
    - Separate invalidation structure to store invalid sequence numbers
  - Garbage collection to reclaim invalid data

- **Deduplicate and compress on the fly**

- **Use RAID for both reliability and performance**
  - Reconstruct data for reads during writes to flash
Purity system overview

- **Purity is (currently) a block store**
  - Highly efficient for both dense and sparse address ranges
  - Similar to a key-value store
    - Key is `<medium, address>`
    - Value is content of a block

- **Purity runs in user space**
  - Heavily multithreaded

- **I/O goes thru Pure-specific driver in kernel**
The basics: read and write path

Read

1. Generate list of <medium, LBA> pairs
2. Find “highest” <medium, LBA> → <segment, offset>
3. Get block at <segment, offset>
4. Return block to user

Write

1. Translate <volume, LBA> → <medium, offset>
2. Persist data to NVRAM
3. Block cache
4. Write data & metadata into log
5. Write metadata into pyramid
6. Identify pattern blocks and deduplicate
7. Return “success” to user

Metadata entries (t3)
Metadata entries (t2)
Metadata entries (t1)
Metadata entries (t0)
Compressed user data
Metadata entries (t3)
Metadata entries (t2)
Metadata entries (t1)
Metadata entries (t0)
NVRAM
Block cache
Basic data storage: segments

- **All data and metadata are stored in segments**
  - Segment numbers are monotonically increasing sequence numbers: never reused!

- **Segments are distributed across a set of devices (MLC SSDs)**
  - Each segment can use a different set of SSDs
  - Write unit is atomic unit in which data goes to SSD: segment need not be written all at once

- **Segment protected by RAID**
  - Across devices
  - Within a single device
Inside a write unit

- Write unit is committed to SSD atomically
- Contents include
  - Compressed user data
  - Checksums
  - Internal parity
- Page read errors can be detected and corrected locally
Storing data

• Data and metadata written to a (single) log
  • Stored first in high performance NVRAM:
    • SLC SSD
    • Other fast block-addressable NVRAM
  • Data is considered committed immediately when it hits NVRAM
  • Later, written to bulk storage (MLC SSD)

• Log persistence
  • System stores
    • Full log (containing data and metadata)
    • Metadata-only indexes
  • Metadata indexes can be rebuilt from log
Storing data: details

- Data stored in tuples
  - Key can consist of multiple (fixed-length) bit-strings
    - Example: \(<\text{medium}, \text{LBA}>\)
  - Newest instance of a tuple \textit{occludes} older instances

- Logs are append-only
  - Data written in sorted order in each segment
  - Consistency isn’t an issue
  - Need to garbage collect...

- Individual tuples located quickly using a mapping structure: \textit{pyramid}
  - Index allows tuples to be found quickly
Storing data: details

• Data stored in tuples
  • Key can consist of multiple (fixed-length) bit-strings
    • Example: <medium, LBA>
  • Newest instance of a tuple occludes older instances

• Logs are append-only
  • Data written in sorted order in each segment
  • Consistency isn’t an issue
  • Need to garbage collect...

• Individual tuples located quickly using a mapping structure: pyramid
  • Index allows tuples to be found quickly
Storing data: details

- **Data stored in tuples**
  - Key can consist of multiple (fixed-length) bit-strings
    - Example: <medium, LBA>
  - Newest instance of a tuple **occludes** older instances

- **Logs are append-only**
  - Data written in sorted order in each segment
  - Consistency isn’t an issue
  - Need to garbage collect...

- **Individual tuples located quickly using a mapping structure: pyramid**
  - Index allows tuples to be found quickly
Storing data: details

- **Data stored in tuples**
  - Key can consist of multiple (fixed-length) bit-strings
    - Example: <medium, LBA>
  - Newest instance of a tuple **occludes** older instances

- **Logs are append-only**
  - Data written in sorted order in each segment
  - Consistency isn’t an issue
  - Need to garbage collect...

- **Individual tuples located quickly using a mapping structure: pyramid**
  - Index allows tuples to be found quickly
The power of the pyramid

- **Metadata entries kept in a pyramid**
  - New entries written to top
    - May occlude entries lower down
  - Add new levels to the top
  - All levels are read-only (once written)

- **Search proceeds top-down**

- **Multiple types of metadata**
  - Mapping table: entries point to data location
  - Link table: lists non-local references
  - Dedup table: location of potential duplicates

- **Segments used for underlying storage**
The power of the pyramid

- Metadata entries kept in a pyramid
  - New entries written to top
    - May occlude entries lower down
  - Add new levels to the top
  - All levels are read-only (once written)

- Search proceeds top-down

- Multiple types of metadata
  - Mapping table: entries point to data location
  - Link table: lists non-local references
  - Dedup table: location of potential duplicates

- Segments used for underlying storage
The power of the pyramid

- Metadata entries kept in a pyramid
  - New entries written to top
    - May occlude entries lower down
  - Add new levels to the top
  - All levels are read-only (once written)

- Search proceeds top-down

- Multiple types of metadata
  - Mapping table: entries point to data location
  - Link table: lists non-local references
  - Dedup table: location of potential duplicates

- Segments used for underlying storage
Merging (flattening) pyramid layers

- One or more adjacent layers can be merged
- Remove tuples that are
  - Invalid: tuple refers to a no-longer-extant source
  - Occluded: newer tuple for the same key
- Result
  - Faster searches
  - (Some) reclaimed space
- Operation is idempotent
  - Search on merged layer has identical results to input
  - Failure partway through isn’t an issue!
  - Switch over lazily
- Improves lookup speed!
- Doesn’t reclaim data storage...
Merging (flattening) pyramid layers

- One or more adjacent layers can be merged
- Remove tuples that are
  - Invalid: tuple refers to a no-longer-extant source
  - Occluded: newer tuple for the same key
- Result
  - Faster searches
  - (Some) reclaimed space
- Operation is idempotent
  - Search on merged layer has identical results to input
  - Failure partway through isn’t an issue!
  - Switch over lazily
- Improves lookup speed!
- Doesn’t reclaim data storage...
Identifying data: mediums

- Each block is stored in a medium
  - Medium is just an identifier
  - Medium is part of the key used to look up a block
- Mediums can refer to other mediums for part (or all!) of their data
- Information tracked in a medium table
  - Typically relatively few entries
Identifying data: mediums

- Each block is stored in a medium
  - Medium is just an identifier
  - Medium is part of the key used to look up a block
- Mediums can refer to other mediums for part (or all!) of their data
- Information tracked in a medium table
  - Typically relatively few entries

<table>
<thead>
<tr>
<th>Medium ID</th>
<th>Start</th>
<th>Length</th>
<th>Underlying</th>
<th>Offset</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0</td>
<td>2000</td>
<td>0</td>
<td>0</td>
<td>Q</td>
</tr>
</tbody>
</table>
Identifying data: mediums

- Each block is stored in a medium
  - Medium is just an identifier
  - Medium is part of the key used to look up a block

- Mediums can refer to other mediums for part (or all!) of their data

- Information tracked in a medium table
  - Typically relatively few entries

<table>
<thead>
<tr>
<th>Medium ID</th>
<th>Start</th>
<th>Length</th>
<th>Underlying</th>
<th>Offset</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0</td>
<td>2000</td>
<td>0</td>
<td>0</td>
<td>Q</td>
</tr>
<tr>
<td>102</td>
<td>0</td>
<td>2000</td>
<td>101</td>
<td>0</td>
<td>QU</td>
</tr>
</tbody>
</table>
Identifying data: mediums

- Each block is stored in a medium
  - Medium is just an identifier
  - Medium is part of the key used to look up a block

- Mediums can refer to other mediums for part (or all!) of their data

- Information tracked in a medium table
  - Typically relatively few entries

<table>
<thead>
<tr>
<th>Medium ID</th>
<th>Start</th>
<th>Length</th>
<th>Underlying</th>
<th>Offset</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0</td>
<td>2000</td>
<td>0</td>
<td>0</td>
<td>Q</td>
</tr>
<tr>
<td>102</td>
<td>0</td>
<td>2000</td>
<td>101</td>
<td>0</td>
<td>QU</td>
</tr>
<tr>
<td>103</td>
<td>0</td>
<td>600</td>
<td>102</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>103</td>
<td>600</td>
<td>400</td>
<td>102</td>
<td>0</td>
<td>RU</td>
</tr>
</tbody>
</table>
Identifying data: mediums

- Each block is stored in a **medium**
  - Medium is just an identifier
  - Medium is part of the key used to look up a block

- **Mediums can refer to other mediums for part (or all!) of their data**

- **Information tracked in a medium table**
  - Typically relatively few entries

<table>
<thead>
<tr>
<th>Medium ID</th>
<th>Start</th>
<th>Length</th>
<th>Underlying</th>
<th>Offset</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0</td>
<td>2000</td>
<td>0</td>
<td>0</td>
<td>Q</td>
</tr>
<tr>
<td>102</td>
<td>0</td>
<td>2000</td>
<td>101</td>
<td>0</td>
<td>QU</td>
</tr>
<tr>
<td>103</td>
<td>0</td>
<td>600</td>
<td>102</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>103</td>
<td>600</td>
<td>400</td>
<td>102</td>
<td>0</td>
<td>RU</td>
</tr>
<tr>
<td>103</td>
<td>1000</td>
<td>1000</td>
<td>101</td>
<td>-1000</td>
<td>Q</td>
</tr>
</tbody>
</table>
Mediums & volumes & snapshots (oh, my!)

- Entries in medium table make a directed graph
  - May need to look up multiple \(<\text{medium, LBA}>\) keys

- Read goes to a specific medium
  - May create new medium, point volume at it instead
    - Medium IDs never reused
    - Once closed, medium is stable: mappings never change
  - Fast snapshots
  - Fast copy offload

- Volume points at one medium
  - Occasionally need to flatten the medium graph to reduce lookups

<table>
<thead>
<tr>
<th>medium ID</th>
<th>underlying</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>12</td>
<td>RW</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>RO</td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>RW</td>
</tr>
<tr>
<td>17</td>
<td>10</td>
<td>RW</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>RO</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>RW</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>RO</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>RW</td>
</tr>
<tr>
<td>14</td>
<td>7</td>
<td>RO</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>RO</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>RO</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>RO</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>RW</td>
</tr>
<tr>
<td>101</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>
Invalidating large ranges of sequence numbers

• All data structures in the system are append-only!
• Often useful to invalidate lots of sequence numbers at once
  • Medium being removed
  • Segment deallocated (link & mapping entries invalidated)
• Keep a separate table listing invalid sequence ranges
  • Assumes invalid sequence numbers never again become valid!
• Merge adjacent ranges together
• Invalid sequence number table limited in size to number of valid sequence numbers
  • In turn, limited by actual storage space
  • In practice, this table is much smaller than maximum size possible
Garbage collection

• Only operates on data log segments
  • GC frees unused pyramid segments too...

• Scan metadata from one or more segments
  • Check data for liveness
  • Read & rewrite data if still live
  • Segregate deduplicated data into their own segments for efficiency
  • Rewrites done just like “real” writes

• Segment may be invalidated when it contains no live data
  • Operation is idempotent!
Garbage collection

• Only operates on data log segments
  • GC frees unused pyramid segments too...

• Scan metadata from one or more segments
  • Check data for liveness
  • Read & rewrite data if still live
  • Segregate deduplicated data into their own segments for efficiency
  • Rewrites done just like “real” writes

• Segment may be invalidated when it contains no live data
  • Operation is idempotent!
Garbage collection

- Only operates on data log segments
  - GC frees unused pyramid segments too...
- Scan metadata from one or more segments
  - Check data for liveness
  - Read & rewrite data if still live
  - Segregate deduplicated data into their own segments for efficiency
  - Rewrites done just like “real” writes
- Segment may be invalidated when it contains no live data
  - Operation is idempotent!
Garbage collection

- Only operates on data log segments
  - GC frees unused pyramid segments too...
- Scan metadata from one or more segments
  - Check data for liveness
  - Read & rewrite data if still live
  - Segregate deduplicated data into their own segments for efficiency
  - Rewrites done just like “real” writes
- Segment may be invalidated when it contains no live data
  - Operation is idempotent!
Picking segments to garbage collect

- Use our own space reporting data to estimate occupancy of any given segment
- Based on estimate, do cost-benefit analysis
  - Benefit: space we get back
  - Cost
    - Higher for more live data: more reads & writes
    - Higher if the segment has more log pages: more data to scan during GC
    - Higher for live dedup data and more incoming references: GC on deduplicated data is more expensive
Tracking reference counts

• Need to know when data can be freed
  • Dedup allows incoming references from elsewhere

• Keep a link table: list of external links to a given segment
  • No exact reference count!
  • Easy to know where to look for incoming pointers during garbage collection
  • Link table maintained as part of the pyramid
    • All links to a block look the same!

• Need not be exact: correctness verified at GC
  • No harm in outdated information
  • Information never becomes incorrect: segment IDs never reused!
Data reduction: compression & deduplication

- **Compression**
  - Primary concern is fast decompression speed!
    - Lots more reads than writes
  - Patterned blocks: pattern stored in index
    - Improves performance: writes never done
    - Saves a lot of space
    - Allows for compact representations of large zeroed ranges

- **Deduplication (at the 512B sector level)**
  - Deduplicate data as soon as possible
    - Reduce writes to SSD
    - Increase apparent capacity
  - Dedup process somewhat inexact
    - OK to “miss” some duplicates: find them later
    - OK to have false positives: checked against actual data
Finding duplicates

- Calculate hashes as part of finding pattern blocks
  - No need for cryptographically secure hashes!

- Use in-memory table for hints, but verify byte-by-byte before finalizing dedup
  - Tables can use shorter hashes
    - Smaller tables
    - Faster hashes (non-cryptographic)

- Different tables (in-memory vs. SSD-resident) use different-size hashes to adjust false positive rate
  - Only impact is number of extraneous reads...
Deduplication and garbage collection

- Easy to find “local” references to blocks in a segment
- How are “remote” references found?
- Use the link table
  - List of logical addresses that reference a given block in the log
  - Lookup of logical address no longer points here ➔ reference must be invalid
- When new reference to duplicate block is written, record new link table entry
  - Old entry is superceded...
Segregated deduplicated data

• During dedup, segregate deduplicated blocks into their own segments
  • Deduplicated data is less likely to become invalid
  • More expensive to GC deduplicated data: keep it separate from other data to save processing

• References to dedup blocks remain in regular segments
  • References to dedup blocks can die relatively quickly
  • Data in dedup blocks lives a long time: until all references die
Flash Personality Layer

- Goal: encapsulate SSD properties in a software layer
- Enable per-SSD model optimization for
  - RAID / data protection
  - Data layout & striping
  - I/O timing
- Allows Purity to use multiple drive types efficiently with minimal software change
Data protection: RAID-3D

- Calculate parity in two directions
  - Across multiple SSDs (traditional RAID)
  - Within a single SSD: protect against single-page read errors
- Turn off drive reads during writes to that drive (if needed)
  - Often, writes interfere with reads, making both slower
  - Latency of reads during write operations can be unpredictable
  - During write to a drive, reads to that drive are done with RAID rebuild
Data protection: RAID-3D

- Calculate parity in two directions
  - Across multiple SSDs (traditional RAID)
  - Within a single SSD: protect against single-page read errors
- Turn off drive reads during writes to that drive (if needed)
  - Often, writes interfere with reads, making both slower
  - Latency of reads during write operations can be unpredictable
  - During write to a drive, reads to that drive are done with RAID rebuild
Data protection: RAID-3D

- Calculate parity in two directions
  - Across multiple SSDs (traditional RAID)
  - Within a single SSD: protect against single-page read errors

- Turn off drive reads during writes to that drive (if needed)
  - Often, writes interfere with reads, making both slower
  - Latency of reads during write operations can be unpredictable
  - During write to a drive, reads to that drive are done with RAID rebuild
Data protection: RAID-3D

- Calculate parity in two directions
  - Across multiple SSDs (traditional RAID)
  - Within a single SSD: protect against single-page read errors
- Turn off drive reads during writes to that drive (if needed)
  - Often, writes interfere with reads, making both slower
  - Latency of reads during write operations can be unpredictable
  - During write to a drive, reads to that drive are done with RAID rebuild
Data protection: RAID-3D

- Calculate parity in two directions
  - Across multiple SSDs (traditional RAID)
  - Within a single SSD: protect against single-page read errors

- Turn off drive reads during writes to that drive (if needed)
  - Often, writes interfere with reads, making both slower
  - Latency of reads during write operations can be unpredictable
  - During write to a drive, reads to that drive are done with RAID rebuild
Data protection: RAID-3D

- Calculate parity in two directions
  - Across multiple SSDs (traditional RAID)
  - Within a single SSD: protect against single-page read errors
- Turn off drive reads during writes to that drive (if needed)
  - Often, writes interfere with reads, making both slower
  - Latency of reads during write operations can be unpredictable
  - During write to a drive, reads to that drive are done with RAID rebuild
Data protection: RAID-3D

- **Calculate parity in two directions**
  - Across multiple SSDs (traditional RAID)
  - Within a single SSD: protect against single-page read errors

- **Turn off drive reads during writes to that drive (if needed)**
  - Often, writes interfere with reads, making both slower
  - Latency of reads during write operations can be unpredictable
  - During write to a drive, reads to that drive are done with RAID rebuild
Data protection: RAID-3D

- Calculate parity in two directions
  - Across multiple SSDs (traditional RAID)
  - Within a single SSD: protect against single-page read errors
- Turn off drive reads during writes to that drive (if needed)
  - Often, writes interfere with reads, making both slower
  - Latency of reads during write operations can be unpredictable
  - During write to a drive, reads to that drive are done with RAID rebuild
Data protection: RAID-3D

- Calculate parity in two directions
  - Across multiple SSDs (traditional RAID)
  - Within a single SSD: protect against single-page read errors
- Turn off drive reads during writes to that drive (if needed)
  - Often, writes interfere with reads, making both slower
  - Latency of reads during write operations can be unpredictable
  - During write to a drive, reads to that drive are done with RAID rebuild
Data protection: RAID-3D

- Calculate parity in two directions
  - Across multiple SSDs (traditional RAID)
  - Within a single SSD: protect against single-page read errors

- Turn off drive reads during writes to that drive (if needed)
  - Often, writes interfere with reads, making both slower
  - Latency of reads during write operations can be unpredictable
  - During write to a drive, reads to that drive are done with RAID rebuild
Customize layout of data on drives

- Lay out segment shards at offsets that optimize drive performance
  - May vary on a per-drive model basis

- Lay out segment data across drives to trade off parallelism and number of drives per (large) request
  - Fractal layout...

<table>
<thead>
<tr>
<th></th>
<th>D0</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>58</td>
<td>59</td>
<td>60</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td>57</td>
<td>56</td>
<td>61</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
<td>8</td>
<td>9</td>
<td>54</td>
<td>55</td>
<td>50</td>
</tr>
<tr>
<td>15</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>53</td>
<td>52</td>
<td>51</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>46</td>
</tr>
<tr>
<td>19</td>
<td>18</td>
<td>29</td>
<td>28</td>
<td>35</td>
<td>34</td>
<td>45</td>
</tr>
<tr>
<td>20</td>
<td>23</td>
<td>24</td>
<td>27</td>
<td>36</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>25</td>
<td>26</td>
<td>37</td>
<td>38</td>
<td>41</td>
</tr>
</tbody>
</table>
Architecture summary

• Manage SSDs with append-only data structures
  • Information is never changed, only superseded
  • Keep indexes for fast lookup
  • Use an append-only invalidation table to avoid unchecked table growth

• Run inline dedup and compression at high speed
  • Optimized algorithms
  • Best-effort algorithms at initial write, improve later

• Optimize data layout and handling for SSDs
  • RAID reconstruct during normal writes
  • Choose optimal layout for each SSD
Questions?