What Differentiated Everpure™ from the Crowd

How a New Entrant Succeeded in a Mature Data Storage Market

When the first Everpure™ (then Pure Storage) storage systems shipped in 2012, the company was a new entrant in an enterprise storage market dominated by established vendors catering to conservative customers. It differentiated itself from its competitors by delivering reliable, affordable all-flash storage systems and focusing intently on simplifying the user "storage experience."

With no track record to demonstrate the value its products provided, it had to publicize its uniqueness. It first published this report, TR-150901, in late 2015--over ten years ago—to lend credibility by describing its technology and the advantages it conveyed in some detail.

The past decade has seen enormous change in information technology, its applications, and the expectations of its users, especially in data storage. Today, most vendors offer some form of technology and business capabilities that Pure pioneered—all-flash systems, data reduction, and so forth. In some important areas, however, including operational simplicity and lifetime non-disruptive upgrading, the company continues to stand apart, even as its product portfolio has been greatly expanded and modernized. Companion Technical Brief TB-260301 enumerates the factors that continue to make Everpure unique among enterprise storage vendors today.

The company continues to make this 10-year old report available to illustrate the roots of its tradition of differentiation and to show how "thinking out of the box" catapulted it from startup entering a mature market to major force in enterprise storage and data management.

Fifteen FlashArray™ Differentiators

Technical Report 150901-v02d01

purestorage.com

Pure Storage, Inc.
650 Castro Street, Suite 400

Mountain View, CA 94041

Contributors: John Colgrove, Ganesh Ramanarayanan, Sandeep Singh

http://www.purestorage.com

Commit: 2289ceb722d0d78df5e94b6da0431b0d312bdc66
Sunday, May 3, 2026, 4:21

Contents

Figures

In recent years, technology and economics have combined to make it feasible to implement enterprise-class storage arrays using flash memory (generally in the form of solid state devices, or SSDs) rather than rotating magnetic disks as their persistent storage. The performance and environmental advantages of flash over disk are overwhelming; consequently a genre of **all-flash arrays** *(***AFA***s) has arisen.*

To date, most AFAs have either been legacy disk-based, with flash solid state devices (SSDs) replacing disks, or new designs that draw heavily on disk array principles. But while flash SSDs resemble disks externally, their performance and reliability properties differ significantly, so a flash-optimized array architecture would differ in many respects from a disk array one. Recognizing this, Pure Storage developed the FlashArray™ architecture specifically to exploit the unique advantages of flash and overcome its limitations to deliver reliable, high-performing, full-function enterprise-class storage arrays.

In some respects, the FlashArray architecture appears to fly in the face of conventional storage array wisdom. But in so doing, it delivers benefits that are well beyond what can be achieved with disk-based designs. This paper describes fifteen ways in which AFAs based on the FlashArray architecture deliver new levels of performance, robustness, administrative simplicity, and cost-efficiency in enterprise data storage.

1: Self-Everything

From the company's inception, the Pure Storage mission has been to utilize the capabilities of flash memory-based *solid state devices* (SSDs) to radically change enterprise data storage. It is well known that flash storage offers order of magnitude improvements in I/O performance, reliability, and environmental (power, cooling, and rack space) cost compared to disks. The cost per byte of raw storage is higher, but AFA developers control cost by *reducing* (removing redundancy from) the stored data. Pure Storage FlashArray™ AFAs deliver these flash storage benefits...more effectively than other array architectures, as the body of this paper demonstrates.

Highlights

- Administrative simplicity is *the* key Flasharray differentiator
- Arrays eliminate resource management tasks common in conventional arrays—no RAID groups, storage tiers, cache management, block alignment, etc.
- Administrators concentrate on storage for the IT mission—volumes, snapshots, replicas.

Come for the performance; stay for the simplicity.

So what's left? Having improved availability, environment, and performance, how can a storage developer effect radical change? The answer lies in a sometimes-neglected storage cost: administration. As arrays have grown in scale, reliability, and performance, they have become more complex. Administrators face a growing number of tasks that include configuring RAID groups, allocating and balancing load across resources, managing client connections, placing the most appropriate data on each cost/performance storage tier, and fixing whatever breaks.

This would be challenging enough in a static environment, but a historical fact of information technology is that demand for data and access to it is insatiable. Not only must administrators create reliable storage environments that perform to expectations; they constantly face demands for more capacity and performance, which they are expected to satisfy without impacting existing applications and data sets. As the demand for storage grows, so does the need for skillful administration of a constantly expanding universe of data. The number of terabytes a single administrator can manage has become a common data center management metric.

Administration is the final frontier in enterprise storage, and is where FlashArrays outshine competitive AFAs. Rather than adapt legacy design principles, Pure Storage architects asked themselves how to use flash to improve administration along with other metrics. The [blindingly obvious] answer is, *simplicity*. A large part of storage administration consists of managing objects like RAID groups and performance tiers that have more to do with the physical properties of arrays and devices than with the information technology mission of an enterprise.

Simplicity: The Other Game Changer

From the outset, the Pure Storage goal has been to simplify administration to the point where administrators can concentrate on what is germane to the IT mission, rather than with physical properties of their equipment. To that end, FlashArrays are:

Self configuring:

Arrays manage their storage as a single pool and present clients with *volumes*, commonly known as numbered *logical units*, or *LUNs*. They dynamically allocate *segments* of media in which they store reduced representations of client data, along with descriptive metadata. They distribute segments across SSDs, and dynamically RAID-protect them against two or more concurrent failures…*all automatically*.

Self healing:

Although electronic devices are generally more reliable than mechanical ones, they can fail. FlashArrays recover from component failures autonomously wherever possible. (Obviously, fans, power supplies, and similar items must be physically replaced, but arrays are designed to operate with less than a full complement of these components, and all of them are hot-swappable.)

Of greater concern, however, are major component (SSD and controller) failures. To protect against these, arrays not only incorporate redundancy; they recover without human intervention, and in the case of SSDs, restore full data protection by automatic distributed rebuilding of at-risk data segments.

Self adapting:

A lot goes on inside a storage array, most of which is invisible externally, *except when it affects performance*. Absorbing client data, computing RAID checksums, persisting and tracking it, executing read commands, recovering from transient errors, and, for AFAs, *reducing* data, all require processing, memory, and I/O resources. Historically, administrators have been responsible for array optimization, using indirect tools such as RAID group definition, data set placement, cache management, and storage network port restrictions. But administrators can only recognize and react at human speeds, whereas conditions in an array change many orders of magnitude faster than that.

FlashArrays automatically adapt to changing conditions in real time. For example, when write load is heavy, they defer inline data reduction to remain responsive. When rebuilding data segments, they may slow down *deep reduction* that minimizes the footprint of long-lived data. When SSDs are busy writing, they often use RAID rebuilds to satisfy read requests. In extreme cases, when free space becomes low, they may *throttle* writes to keep resources available for *recycling* (also called *garbage collecting*) storage that holds obsolete content. Arrays make these, and dozens of similar adjustments, automatically to continually optimize utilization of their internal resources.

What's Missing from This Picture?

The net effect is that many common administrative functions don't exist in FlashArray environments. For example:

No RAID groups:

FlashArrays manage storage as a single pool. They persist incoming data in approximate arrival order rather than in a fixed LBA-physical location relationship. Administrators do not decide which SSDs to virtualize as which LUNs, how to RAID-protect data, or what spares to reserve. Arrays themselves allocate space for storage and rebuilding, transparently and optimally.

No tiers:

Because arrays treat all storage the same, there is no tiering, as is typically found in hybrid arrays. Administrators do not associate specific devices with specific clients, applications, or data sets. Similarly, there is no I/O performance tuning. Administrators connect clients and volumes, and arrays manage their resources to respond fairly to all. Arrays are better-positioned to rebalance resources in real time than an administrator observing and responding to changing conditions at human speed could possibly be.

No cache management:

Typical SSD read access times are in the range of a couple hundred microseconds, so there is little need for read cache in an AFA. The majority of FlashArray DRAM is used to buffer data waiting to be persisted, and to cache metadata. Arrays manage their own DRAM; there are no associated administrative tasks.

No alignment constraints:

Aligning blocks of client data with physical media locations is vital for disk array performance. But because SSDs do not seek or rotate, alignment need not be a consideration. But some legacy designs align virtual and physical block addresses, and even the more modern designs that use content addressing depend on logical-to-physical alignment, neither of which is relevant with flash.

FlashArrays deduplicate and compress incoming data and pack it into segments for writing. Packing is dense—there is no padding to sector or other boundaries. A block's LBA has no alignment relationship to the data's media representation. Administrators can size volumes, and specify database and file system block sizes, based on application requirements rather than storage constraints.

No binding of ports to array resources:

Some older array designs effectively partition resources (IO processors, cache, storage device connectivity) so that not all storage is accessible from all storage network ports. All FlashArray ports provide access to all volumes. Any constraints on client port access are imposed by the storage network, not the array. If a client port and an array port are in the same zone, the client can access its volumes on that port. There is nothing for the array administrator to configure or manage.

FlashArray administrators create named volumes of specified size, connect them to clients, group them for snapshots and replication, resize them when applications need more storage, and destroy them when they are no longer needed. All of these tasks satisfy client and application requirements, none of them relate to physical properties of the array. The only physically motivated administrative tasks are adding shelves when more storage is required and replacing the occasional failed component. The array itself takes care of everything else.

2: Scaling: Up vs. Out

Most enterprises' storage needs increase over time. Thus it is important that arrays be able to grow, or *scale*, capacity and performance during their lifetimes. Arrays grow either by *scaling-out* or *scaling-up*. Scale-out arrays typically consist of *bricks*, each of which is essentially a complete array, with storage, controller, storage network ports, and connections to peer bricks. Scale-up arrays have a fixed number of controllers (usually two) that connect to a variable amount of back-end storage. FlashArray architecture is scale-up, a decision taken at the outset for the reasons outlined in the paragraphs below.

Hardware Reliability

A failure tolerant single-brick scale-out array typically has about as many hardware components as a scale-up array of equivalent capacity. As the two scale, however, the scale-out array's component count increases faster than that of the scale-up one, because each brick adds controllers, memory, ports, power/cooling, and cabling as well as storage. And while data center-grade electronic components are generally pretty reliable, reliability mathematics shows that systems with more components are inherently more failure-prone.

A FlashArray has exactly two controllers. Arrays scale up by the addition of *storage shelves* that contain SSDs, power and cooling, and SAS interfaces. Thus, their component count grows more slowly with additional capacity than that of a typical scale-out array.

Highlights

- Scale-out arrays' hardware and software component counts and interactions between bricks inherently increase faster than scale-up ones as capacity increases.
- A system with more components (and more interactions among them) is inherently more failure-prone than a system with fewer components.
- The FlashArray architecture scales *up*. Capacity and performance can be increased independently of each other.

  With a scale-out array, additional performance may or may not be useful, but cost must always be paid.

- Non-disruptive upgrade technology and the Evergreen storage model make it easy to track technology, with no risk of downtime or data migration.
- Shelf evacuation makes it possible to upgrade SSD technology non-disruptively by replacing entire shelves of SSDs with newer models.

Software Reliability

Architecturally, storage arrays are *clusters* whose controllers cooperate to provide coordinated storage and I/O services to clients. A FlashArray is a two-node cluster. In a scale-out array, each brick is a node (or two, if bricks are internally redundant). Each controller runs a software kernel, and storage services. Each software instance is susceptible to bugs, and by the same reliability mathematics that apply to hardware, a system with more software instances inherently has a higher probability of failure than one with fewer instances.

Perhaps a more troublesome source of scale-out array failures is inter-controller interactions. Scale-out array controllers intercommunicate constantly to reserve storage devices and to exchange work items. From a reliability standpoint, there are potential negative effects:

- Errors in one controller may cause errors in another. For example, if a controller unlocks a device prematurely, the next controller to write to the device might have its data overwritten
- Again, reliability mathematics indicates that as the number of controllers in an array increases, the potential for interaction-related errors increases as well.

Clustering is among the most complex problems in applied computing, and complexity increases with inter-system interactions. Each brick added to a scale-out array means more interactions, and therefore the probability of errors. In contrast, FlashArray controllers have specific primary and secondary roles that limit interactions (Section 6 discusses FlashArray controller roles). In particular, controllers do not contend for access to SSDs, thus eliminating one major source of potential errors.

Cost-Effectiveness

Scale-out advocates argue that adding processing power, memory, and storage network ports along with storage guarantees that performance will scale with capacity. But with additional resources comes increased latency, due primarily to the need for controllers to interact. Moreover, data collected by the Pure1 support facility suggest that few users approach FlashArray performance limits. Thus, while a scale-out array may have the *potential* for higher performance, it is performance that will never be used in most instances.

What is always true, however, is that the cost of added components must be paid when a scale-out array is actually scaled out. In contrast, each FlashArray model is designed to deliver advertised performance with the maximum amount of supported storage. When an array *does* require more performance (with or without additional capacity), a *non-disruptive upgrade* (NDU) can replace its controllers with more powerful ones. With FlashArray NDU there is (a) no service outage, and (b) little or no increase in controller components and interactions.

Technology Tracking: Controllers

With FlashArray non-disruptive upgrade, components, including controllers and SSDs, can be replaced while an array is serving clients. This is possible partly because controllers are *stateless*—neither one contains any persistent information about the array.

Stateless controllers can be replaced online, but they have an even more important long-term benefit—they make *non-disruptive technology refresh* possible. Processor, memory, storage network, and other technologies all improve over time. It is almost inevitable that by the time an array has been in service for 3-5 years, a better model is available, offering more capacity, higher performance, lower space and power consumption, additional functionality, and/or other benefits.

Data center and application managers are generally eager to take advantage of new capabilities, but downtime is a huge inhibitor to change as enterprises increasingly rely on continuously available digital information. Downtime for *forklift upgrades* and for migrating petabytes of data, is disruptive at best, and destructive at worst.

Together, FlashArray technology and the Evergreen storage model (Section 15) eliminate both downtime and economic barriers to regular technology refresh. Periodic technology refresh is integral to FlashArray service contracts; NDU eliminates the cost and risk of downtime when it occurs. Moreover, stateless controllers eliminate data migration—after a FlashArray controller upgrade, data is just where it was before.

> *NDU isn't limited to technology refresh. As capacity, performance, or space efficiency requirements grow, users can purchase and install "flex bundle" upgrades at any time, again without disrupting service.*

Technology Tracking: Devices

Flash technology is evolving rapidly; SSDs are getting bigger and more capable. Because requirements for digital data always seem to increase, the ability to increase storage array capacity is key to long-term success.

Scale-out arrays typically balance load by distributing data across their bricks, for example by hashing content to determine where to store a given client block. This design usually requires bricks of equal capacity, which can make non-disruptive SSD capacity upgrades awkward.

With FlashArray log-structured media,[1] the location of a block of data is independent of both its content and its size, so arrays can support SSDs of different capacities. A new storage shelf can therefore be populated with the most appropriate devices available *when it is added to an array*. Arrays automatically adjust space allocation parameters to favor newer devices with more free space, dynamically equalizing I/O load and flash wear with no administrative interaction.

1 FlashArray media management is described in Technical Report TR-141001, available at http://community.purestorage.com/t5/Pure-Internal-Knowledge-Base/FlashArray-Technical-Reports/ta-p/6190

3: SLC vs. eMLC vs. cMLC

Another important Pure Storage architectural decision was to base arrays on so-called *consumer-grade multi-level cell* (*cMLC*) devices rather than the more expensive single-level cell (SLC) flash that was favored for enterprise applications at the time, or the enterprise MLC (eMLC) technology that has since become more popular.

In an SLC device, each flash cell encodes one bit as an electrical charge, so a single voltage threshold determines whether a cell contains a 0 or a 1. MLC encodes two bits of information in each cell, so four distinct thresholds must be detectable, one for each of the four possible two-bit patterns.

There are technical advantages to SLC, particularly its *endurance*. Due in part to more forgiving voltage thresholds, an SLC cell can be written 10 to 100 times more than an MLC one before it becomes unusable. The motivation for MLC, however, is cost. MLC flash is used in millions of smartphones, tablets, cameras, and other consumer devices. In these applications, writes are infrequent, so endurance is not stressed. What *is* of paramount importance, however, is minimizing cost, and MLC's higher data density and manufacturing economy of scale achieve that.

Highlights

- FlashArray architecture is predicated on cMLC storage.
- Consumer-driven economies of scale drive cMLC cost and, more importantly, performance to specified reliability.
- The unique FlashArray media layout and management architecture is what makes cMLC flash enterprise-capable.
- Techniques for maximizing flash lifetime include data reduction, minimizing, balancing, and aligning SSD writes, and writing complete RAID stripes to minimize program-erase cycles.
- In addition to maximizing cMLC lifetime, the abovementioned FlashArray media management techniques enhance array performance.

Driven by consumer market volumes, cMLC technology investment is likely to be greater than in other variants, leading to even more rapid evolution.

Overprovisioning

SSD developers typically compensate for MLC's lower endurance by *overprovisioning*—by building in more flash cells than the devices expose. Devices detect and retire faulty banks of cells, and substitute others for them. The amount of overprovisioning is the primary difference between cMLC and eMLC. A typical cMLC SSD contains about 10% more capacity than it exposes. eMLC devices, on the other hand, are typically overprovisioned by 30-60%. They fall between low-cost cMLC and high-cost SLC, with greater apparent endurance than the former, but less than the latter, with a cost per byte between the two.

The specified endurance of eMLC devices is due to their ability to tolerate more worn-out cells than cMLC, however, not to any significant difference in cell technology. Moreover, any net endurance increase over cMLC has a cost—an eMLC device of a given exposed capacity contains more flash memory than a cMLC one of the same capacity, and somehow or other, that memory has to be paid for.

Cost-Driven Reliability

Arguably, most flash memory goes into consumer applications—embedded intelligence, personal appliances, laptop computers, and so forth. These markets operate on thin margins, driven partly by hardware cost, but more importantly, by the cost of service. Trouble reports, service calls, manufacturer recalls, and field upgrades are deadly enemies of profitability. Thus there is a strong motivation for vendors to implement manufacturing processes and testing regimens that ensure field reliability and minimize service calls.

In enterprise IT, however, although service calls are disruptive and undesirable, they are a generally accepted fact of life. Thus, paradoxically cMLC-based devices can actually be expected to be more reliable than devices made for the enterprise, within their specified operating parameters. This doesn't mean that a cMLC device is likely to have greater endurance than an eMLC or SLC one. It means that cMLC devices are the most likely to

perform to advertised specifications. If cMLC specifications can be made to work in an enterprise storage array, there's no reason to pay the cost premium for eMLC, or the even greater SLC one.

The Innovator's Dilemma Effect

The concept of technology disruption is well known, especially in computing. A new technology is inferior to an established one in all conventional metrics, but solves some heretofore unrecognized problem. The new technology proliferates, motivating further investment, and eventually improves to the point of squeezing the incumbent into a narrowing niche.

Disk drives are a case in point. When they were introduced, personal computer disk drives were lower-performing and less reliable than high-RPM enterprise ones. But the demand for personal and bulk storage motivated investment. Reliability and capacity increased, and surrounding technologies evolved to enhance low-cost drive capabilities. Today, enterprise drives command a small and decreasing share of the storage device market.

Something similar appears to be occurring with flash. cMLC has lower endurance and performance than eMLC and SLC, but its low cost and reliability are so compelling in the consumer market that an increasing share of investment is being devoted to it. SLC is already being eclipsed as a data center storage technology, and as the market share of eMLC contracts, investment in furthering that technology is also likely to decrease. To use a popular ice hockey analogy, by focusing on cMLC, Pure Storage *has skated to where the puck was going to be*.

Making it Work

The weakness of cMLC is *endurance*—the number of times a device's flash cells can be written before read errors become intolerable. Endurance is seldom a problem in consumer applications, where duty cycles are low, and overwrites are infrequent. But enterprise storage arrays operate continuously, often under heavy load, and are expected to perform reliably for 3-5 year life cycles. In adopting cMLC, Pure Storage implicitly committed to make it suitable for use in the enterprise. The architects devised a unique media layout and accompanying read/write strategy to make it work.[2] Specifically, FlashArray software is designed to:

- Minimize the number of SSD writes
- Balance writes as evenly as possible across SSDs for maximum endurance
- Minimize SSDs' internal *program-erase* churning by always writing full *erase blocks*
- Maximize the *value* of each SSD write with payloads that consist entirely of new and reorganized data
- Eliminate the read-recalculate-rewrite cycles characteristic of typical RAID implementations
- Minimize simultaneous SSD writes to keep the maximum number of devices available for reading.

When a client writes data to a FlashArray, the array:

- Reduces the data (removes patterned and most duplicate blocks, and compresses what's left)
- Copies the resulting representation to NVRAM
- Indicates to the client that its write is complete.

The array then buffers the reduced data along with descriptive metadata for eventual writing to SSD. Actual writing is deferred until a group of 1-megabyte buffers (called a *segio*, usually consisting of 8-10 *shard buffers*) has filled and been protected by RAID-3D checksums. Thus, incoming client data is organized as a log in approximate arrival order. An array contains a *cbmap* that relates data addresses in volume/LBA space to media locations. There is no fixed relationship between a block of client data and its location in an array.

2 The FlashArray media layout.is described in Technical Report TR-141001, available at http://community.purestorage.com/t5/Pure-Internal-Knowledge-Base/FlashArray-Technical-Reports/ta-p/6190

The FlashArray scheme has several advantages, from both media endurance and performance points of view:

Writes to media are large and aligned with SSD internal structures:
Large writes incur less processing overhead per byte than small ones. But the real advantage is that aligning writes with SSDs' internal structures minimizes the devices' internal *program-erase* churning, thus improving both write performance and flash endurance.

RAID-related read-recalculate-write cycles are eliminated:
Each segio includes two buffers in which arrays calculated RAID-3D checksums as they accumulate incoming data. Each segio is a *full RAID stripe*. Because arrays write entire stripes, there is no need to read unmodified data to calculate checksums.

SSD writes can be serialized:
Write buffer contents are never altered, so there is no need to lock SSDs while they are being written to prevent in-flight modifications. Arrays write one or two buffers at a time, so most of their devices are free to perform (much faster) reads most of the time. (To read from a device that *is* busy writing, an array can read from other SSDs and rebuild the requested data.)

Allocation can balance wear:
Because arrays manage media in 8-megabyte *allocation units*, they do not require devices of equal capacity. When allocating space, they favor devices with shorter time in service and with more free space. Thus, over time, they direct roughly equal numbers of writes to all SSDs. To increase capacity or refresh device technology, users can choose the most appropriate capacity/technology point without being forced into a forklift upgrade.

Disk array architectures are not generally optimized to minimize the number of device writes, but they do strive to minimize the performance impact of mechanical latency. The greater endurance of eMLC SSDs is a necessity for disk array architectures that have been repurposed to accommodate flash media. The FlashArray architecture delivers performance, reliability, and lifespan suitable for enterprises at cMLC cost. It is likely that over time, increasing investment motivated by the consumer flash market will cause cMLC technology to evolve more rapidly than lower-volume variants, forcing SLC and eMLC into ever-narrowing market niches.

4: Data Reduction: *All the Data, All the Time*

Despite rapid decline in SSD prices, a byte of flash storage costs more than a byte of disk at the present time. Moreover, the limited *endurance* fundamental to all forms of flash technology (see Section 3), means that an SSD's useful lifetime is largely determined by the amount of data written to it. In other words, the less data written to an SSD over time, the longer its useful life is likely to be.

To make flash storage economically viable while its raw cost is greater than that of disk, as well as to maximize the useful lifetimes of flash devices, AFA vendors *reduce* (remove redundancy from) data before storing it on SSDs. Data can be reduced by:

Pattern elimination:
Replacing blocks that consist entirely of one repeated byte with a metadata representation.

Deduplication:
Replacing blocks that are identical to blocks stored elsewhere in the array with metadata.

Compression:
Replacing repeated byte sequences within a block with more compact representations.

These techniques represent client data more compactly, but the representations are almost never multiples of the 512 byte legacy disk sector size. For disk array-based architectures, this tends to limit the effectiveness of data reduction.

Disk array-based AFA architectures typically map sequences of client LBAs to sequences of disk sectors, with a goal of minimizing physical gaps between adjacent LBAs.

Some AFA designs map LBAs to SSD media locations by managing 4-16 kilobyte *blocks* of consecutive LBAs and hashing the data they contain to determine where to place them on the array's SSDs. On the surface, such mappings are useful, because they automatically deduplicate identical blocks.

Highlights

- Data reduction (pattern elimination, deduplication, and compression) is key to AFA cost-effectiveness and flash device longevity
- Arrays that map client LBAs to media sectors provide limited opportunities for data compression
- Arrays that map client LBAs to media sectors may not recognize duplicate data sequences that are not identically aligned in client LBA space or that are embedded within larger blocks
- Arrays store reduced data without regard for block alignment; there is no padding to specific boundaries
- FlashArray medium graphs map large blocks of data that are part of snapshots, clone volumes, or xcopied ranges; they make snapshot creation and xcopying nearly instantaneous
- Arrays reduce data in two stages: initially as it enters, and more exhaustively in the background during GC.

FlashArray two-stage data reduction minimizes impact on client response and eliminates the burden of deep-reducing transient data.

Both direct mapping and content hashing schemes have a significant limitation, however: LBA-to-sector mapping is not optimal for data compression. Because each LBA corresponds to a media sector, there is nothing to be gained by compressing the data within a sector.

Some arrays do compress data by subdividing blocks. For example, in an array that maps 32 sector LBA sequences to 16 kilobyte blocks, a block that compresses to less than half its 16 kilobyte size can share physical storage with a similarly compressible block. Mapping becomes more complex—the array must track the locations of data in subdivided blocks—but some data compression savings can be realized. Blocks can be divided in half, or perhaps even quarters, but beyond that, mapping becomes impracticably complex.

An obvious limitation of these schemes is client data that compresses to slightly more than the sub-block size. For example, a block that compresses to 9 kilobytes, leaves 7 kilobytes of physical storage unused.

Content addressing schemes have an additional limitation. They do not deduplicate sequences of identical sectors that do not align with physical storage management boundaries. For example, an array that manages 16 kilobyte blocks will recognize duplicate content that clients write to LBAs 0, 32, 64, 96, etc., but will not recognize duplication if the content is written to sectors 0, 48, 72, etc.

A third limitation of content addressing schemes is in the sizes of duplicate blocks they can accommodate. Even ignoring the possibility of *aliasing*—two blocks with different content that hash to the same value—two blocks must be of the same size and alignment for their hashes to be identical. For example, the hash for a 16 kilobyte block that contains 10 kilobytes of data duplicated elsewhere in the array will differ from the hash of the block that contains the duplicate data, and the 10 identical kilobytes will be stored twice.

To summarize, while AFAs that map client block addresses to blocks of physical storage can compress and deduplicate data, they have three important limitations:

- Compression is inherently wasteful, because the compressed size of a block is rarely an exact multiple of the array's storage management block size
- Deduplication is limited to blocks with the same alignment in LBA space. Duplicate blocks that are not identically aligned are not deduplicated
- Deduplication is limited to blocks of the same size. LBA sequences within a block or that span two blocks are not deduplicated, even if duplicate data exists in the array.

FlashArray Data Compression

The FlashArray media organization avoids the limitations described in the preceding paragraph.[3] Arrays store reduced client data in 1-megabyte blocks called *shards*, in approximate order of arrival, without regard to LBA. They maintain LBA-to-data location mappings in a persistent structure called the *cbmap*. There is no fixed relationship between either LBA or content and data's location on SSD media.

Section 3 describes the beneficial effect of the media layout on flash endurance. The layout also benefits data reduction efficiency. Arrays reduce incoming data, copy it to NVRAM, and place it in shard buffers. Initial reduction is not exhaustive; its goal is reasonable efficiency with minimal impact on client response. Compression algorithms minimize CPU consumption at the expense of thoroughness. Deduplication compares incoming data against high-probability candidates. After initial reduction, a client data block may be represented by:

- Metadata that indicates a repeated byte pattern (for example, zero-filled database pages)
- Metadata that points to an identical already-stored block
- A compressed representation of the data
- The original data (if it is uncompressible).

Representations are packed into shard buffers *densely*—each begins exactly where the previous one ends. A 16 kilobyte block that compresses to 9 kilobytes occupies 9 kilobytes of buffer space (and eventually SSD media). The next block is immediately adjacent; there is no padding to a specific boundary. Similarly, 16 kilobytes of zeros or 16 kilobytes of data duplicated elsewhere in the array occupy only the few bytes required to identify the content; there is no pad between them and the next block's representation.

FlashArray Deduplication

FlashArray deduplication is similarly efficient. An array computes a lightweight hash signature for each sector of an incoming block. During initial reduction, it compares these to signatures of high-probability duplicates held in memory tables. Upon finding a match, the array retrieves and compares stored sectors for adjacent LBAs until it

3 The FlashArray media layout is described in Technical Report TR-141001, available at http://community.purestorage.com/t5/Pure-Internal-Knowledge-Base/FlashArray-Technical-Reports/ta-p/6190

encounters a signature mismatch. (To limit fragmentation, only sequences of 8 or more sectors are deduplicated.) Thus, they detect duplicates of any length (up to a 64 sector maximum), whatever the alignments in LBA space.

Like initial compression, initial deduplication is designed for minimal impact on client response. Thus, its search scope is limited, and hash signatures are simple to minimize computation. To eliminate the possibility of aliasing, however, arrays compare actual stored and incoming data before replacing the latter with pointers to already-stored representations.

Mediums: Large-Scale Deduplication

Virtualized data centers often need to create large quantities of duplicate data. Snapshots are one example, but perhaps the prime example is *virtual desktop* or *server integration* (VDI or VSI) in which a single array hosts dozens or hundreds of (initially) identical virtual system disks. To simplify creating many large blocks of identical data, the SCSI storage protocol includes an *xcopy* command for copying a large amount of data from one array location to another. But while a client can copy gigabytes with a single command, what happens inside an array is another matter. Arrays that xcopy by physically reproducing data can take hours to execute a single command.

FlashArray data organization makes both snapshot creation and xcopying nearly instantaneous. Each array's cbmap relates sector addresses to media locations. Volumes and snapshots are represented by graphs of *mediums*. Each medium contains pointers to blocks of cbmap addresses, so it indirectly maps LBAs to data locations. To take a snapshot, for example, an array creates two new mediums:

For the snapshot:
Effectively captures the state of the volume and does not change

For the volume:
Records future writes to the volume.

Initially, both mediums point to the volume's original medium, which becomes read-only when the snapshot is taken. Thereafter, when clients write data, the array creates new cbmap entries for the affected LBAs and updates the volume's new medium to point to them so that reads return the most recently written content. References to unaltered LBAs fall through to the original medium.

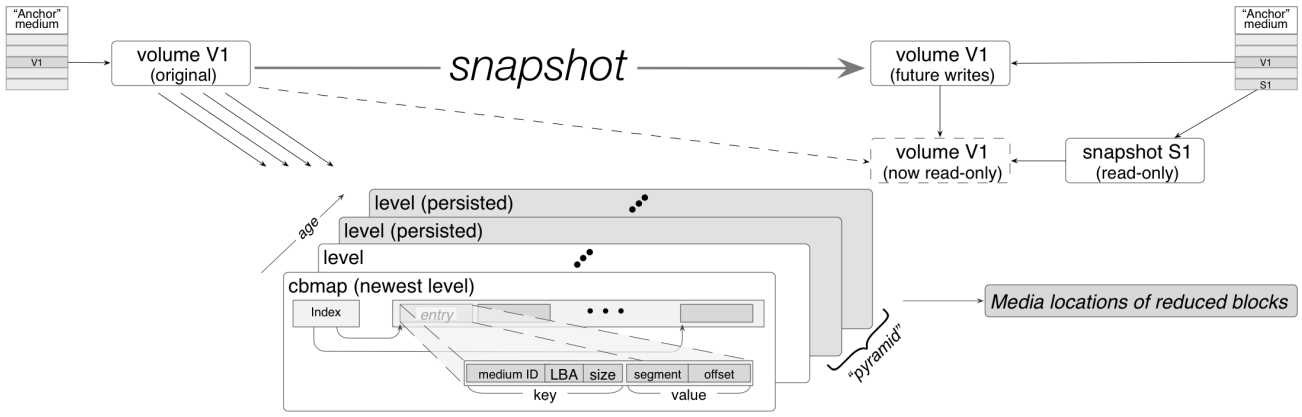

Figure 4.1: FlashArray Medium Maps

Similarly, to create clones or perform xcopies, an array creates mediums for the new objects that initially point to the medium that maps the underlying data. Later updates to the originating volume, to the clone, or to the xcopied

block range are mapped by the new structures' mediums. The original mediums become read-only, and continue to point to any unaltered data.[4]

Two-Stage Data Reduction

Because FlashArray SSD contents are essentially a log of client writes, overwrites create "holes" of superseded data within segments. A background task known colloquially as *garbage collection* (GC) processes segments to recoup this space for reuse, and simultaneously performs two other important functions:

Data consolidation:
GC consolidates live data blocks from the segments it processes in new segments, choosing each block's new location based on criteria designed to optimize future read performance. For example, it places duplicated blocks in *dedup segments*, assuming that they are likely to remain unaltered longer than unique blocks. Similarly, it attempts to co-locate blocks with proximate LBAs, assuming that they may be part of the same file or database, and are therefore likely be read at the same time.

Deep reduction:
Initial data reduction is designed to minimize impact on client response. This is sometimes portrayed negatively, because it does not necessarily result in the smallest data footprint. Potential duplicates may be missed because the signature search is not exhaustive; highly-compressible data may not be detected because its compression pattern is complex. GC counters this by *deep-reducing* data. As it processes segments, GC performs a more global duplicate search, and attempts tighter compression using a two-stage (LZO-Huffman) algorithm. Two-stage data reduction has two advantages. First, the (more intensive) computation is not in the client response path. Second, no resources are expended to reduce data that has been overwritten by the time GC processes its segment. This contrasts with completely inline reduction which necessarily compromises either client response time or quality of reduction.

Always-on Data Reduction

Data reduction is integral to FlashArray operation. Administrators cannot disable or throttle it, although arrays may adjust their own behavior, for example by throttling client writes or bypassing initial compression when client activity is especially high. The philosophy underpinning this design is twofold: (1) options are the enemy of simplicity, and (2) arrays have the best knowledge of what is happening within them, and are better positioned to make rapid adjustments in response to changing conditions than administrators observing from the outside.

4 Technical Report TR-15601 describes the medium map structure in greater detail, including examples. The report is available at http://community.purestorage.com/t5/Pure-Internal-Knowledge-Base/FlashArray-Technical-Reports/ta-p/6190

5: Stateless Controllers

Every FlashArray includes two controllers and one or more shelves of SSDs (the first "shelf" in a FlashArray//m is contained in the base array chassis). Controllers are *stateless*—they contain no persistent information about the array's configuration (volumes, snapshots, client connections, network addresses and parameters, administrator accounts, etc.) or about the data stored in it—all information is stored in SSDs or, for data in flight, in NVRAMs. Stateless controllers have several important advantages:

Fast failover:
Because all persistent information is stored on SSDs and NVRAMs, failover from one controller to another is effectively array startup (see Section 7 for additional advantages of minimizing code paths). It occurs quickly, and there is no controller-dependent context for the new incumbent controller to recover.

Non-disruptive controller replacement:
To replace a controller, one powers it off (causing the array to fail over), disconnects cables, removes it from the rack or FlashArray//m chassis slot, and installs, cables, and powers on the replacement. When powered on, the new controller automatically joins the array, and is eligible (but not required) to assume the primary role.

Easy technology transition:
The technologies inside a FlashArray controller evolve rapidly. But stateless controllers make it possible to upgrade a running array (including FA-400 series to FlashArray//m upgrades). This capability eliminates forklift upgrades. When users opt for Forever Flash (Section 15), the only constant is their data. As technology evolves, controllers can be replaced with more capable ones; as capacity needs increase, the best available SSD technology can be added, all without disrupting service to clients.

Really secure data security:
The configuration information stored in FlashArray SSDs includes frequently-refreshed keys used to unlock the devices for access. Arrays distribute keys so that more than half their devices are required to reconstruct them. Thus, unlike arrays that store encryption keys in controllers, there is nothing in a FlashArray controller that can be used to gain access to an array's data.

Storage device migration:
Because controllers are stateless, it is possible to connect an array's entire complement of SSDs to a different pair of controllers with no loss of data or configuration information. The new controller pair effectively becomes a clone of the original array.

Highlights

- FlashArray controllers are stateless. All configuration information and data is stored in SSDs or NVRAMs
- Because controllers are stateless, failover, controller replacement, and controller upgrade are all non-disruptive
- FlashArray controllers do not contain any data encryption or SSD unlocking information; it is distributed among an array's SSDs
- An array's SSDs can be connected to a different controller pair, which becomes an identically configured array containing the original data.

With Forever Flash and the non-disruptive upgrades that stateless controllers make possible, the only long-term constant in an array is the data.

6: The Second Controller

Conventional storage array wisdom holds that *active/active* architectures in which all controllers access storage devices concurrently and share the client I/O load are superior to *active/passive* ones in which a *primary* controller serves clients and the other activates only if the primary fails. On the surface, active/active designs are attractive for two reasons:

More resources:
Intuitively, if two or more controllers are active, twice the processing, twice the cache, twice the access to back-end storage, and twice the number of I/O ports are available to handle the I/O load.

Fast failover:
If a controller fails, the second controller is already operating. Assuming that clients have access to both controllers, they retry I/O in flight on alternate paths and direct future requests to the second controller.

Highlights

- It is commonly believed that active/active array designs are universally superior to active/passive ones
- Active/active arrays typically experience severe performance degradation when a controller fails
- FlashArray architecture can be described as *active/active front end, active/passive back end*
- Both controllers communicate with clients, but the primary executes all commands
- When a controller fails, clients redirect I/O to the other controller, which takes over the array's command execution role

The FlashArray architecture delivers virtually full performance, even when an array is operating with a single controller.

But there are offsetting drawbacks to the active/active architecture. One significant one is the need for controllers to constantly coordinate access to storage devices so that writes do not interfere destructively. (For example, if controller A reads data intending to update and rewrite it, and between the read and rewrite, controller B overwrites the data, controller A's rewrite obliterates what controller B wrote). Avoiding destructive interference requires constant messaging among controllers to reserve and release devices. Constant messaging is detrimental to both throughput and latency, and moreover, multiplies the number of code paths, increasing the probability of difficult-to-reproduce-and-debug errors (see Section 7 on code paths).

But the most significant drawback of active/active arrays is their performance while degraded. When a controller fails, the other controller takes over the entire I/O load. If a two-controller array is at or near full capacity, it becomes supersaturated, and client response time increases exponentially. Prudent administrators avoid this scenario by *overprovisioning* so that each controller operates below half of maximum capacity under normal circumstances. When the array is reduced to one controller, it can perform acceptably, albeit with increased latency. Thus, an active/active array administrator's choices are:

- Provision for the required throughput and accept the risk of unacceptable response if a controller fails
- Provision for twice the required throughput, and pay the cost to avoid the risk.

Historically, users have not regarded active/passive designs (which do not exhibit his problem) favorably. Active/passive arrays avoid the complexity of high-frequency controller interactions, but they have the same inherent cost as active/active ones, with longer failover times. A passive controller must recognize failure, take control, and deal with I/O in flight. Clients have to redirect I/O to previously dormant storage network addresses.

The FlashArray *Active/Active Front End, Active/Passive Back End* Design

Pure Storage has adopted a third approach to controller high availability that achieves fast failover, avoids contention for storage devices, and most important, delivers virtually full performance when operating with a single controller. Simply put, the FlashArray architecture is active/active with respect to communicating with clients, but active/passive with respect to accessing SSDs and executing I/O commands.

At any point in time, one FlashArray controller is *primary*, and the other *secondary*. Both communicate with clients, but the primary executes all commands. The secondary controller relays commands, and exchanges data with the primary via redundant 56 Gb/S InfiniBand connections (FA-400 family) or via a non-Transparent Bridge (NTB) between the controllers' PCIe buses (FlashArray//m family). Relaying typically adds only 3-5% (typically 30 to 50 microseconds) to client latency. This design has three big advantages over active/active ones:

Simple device management:
The primary controller manages all SSD access. There is no controller-SSD affinity, temporary or permanent, so controllers do not interact to manage momentary device reservations. This more than compensates for the additional latency of relaying information between controllers.

Active/active client perspective:
From a client perspective, arrays are active/active—they do not restrict port connections (storage network administrators may use zones to restrict port access). All array ports accept commands for any volume to which the client is connected. For optimal I/O balance when both controllers are operating, Pure Storage recommends configuring clients to use *shortest-queue* path selection where it is available.

Full performance during single-controller operation:
Perhaps the most important advantage of the FlashArray design is that because one controller executes all commands, controller failure has a negligible effect on client performance—all that is lost is communication with the failed controller's storage network ports. As long as clients can route traffic to the other controller, the array continues at full performance, because commands are always processed by a single controller.

Each FlashArray controller's maximum port capacity is matched to its internal processing and back-end I/O capability. Administrators need not configure double capacity to provide for failure scenarios, or alternatively, risk overload when a controller fails, nor is any special tuning required to balance I/O during degraded operation.

7: Streamlined Code Paths

Considering what a virtualizing storage array does, its software is inherently complex. Typically, mainline read and write code paths are straightforward; complexity arises because arrays must recognize and recover from every error that can occur in every path. Error recovery makes software unreliable for two reasons:

Complexity of state when errors occur:
Every error, and hence every recovery path, involves some state—memory that is allocated (or not), data structures that are initialized and/or altered (or not), data that has been persisted (or not), metadata that is up-to-date (or not), and so forth. Each recovery path must properly account for all possibilities, increasing the number of recovery scenarios exponentially.

Frequency of path exercise:
Most of the time, most things work properly, so error recovery code paths are exercised infrequently. As a consequence, most recovery testing is artificial—tests are generally based on assumptions about what can go wrong and how. Thus, error recovery code quality is strongly correlated with the quality of test design.

Highlights

- Error recovery accounts for a large percentage of a typical array's software
- A fine-grained approach to error recovery multiplies recovery paths, and therefore testing requirements exponentially
- FlashArray software classifies errors as broadly as possible to minimize the number of situation-specific error recovery paths
- Examples of code path minimization include treating failure to allocate memory as controller failure, absence of an orderly shutdown procedure, and using RAID rebuild to execute client reads to busy devices

Minimizing the code paths in an array promotes software robustness and therefore array reliability.

The net result is that a large percentage of the software in a typical array lies in seldom-exercised recovery code paths that may only have been tested by artificial error injection. This is unfortunate, because the occurrence of unpredictable errors is precisely when software robustness is most critical. The Pure Storage approach is to:

- Classify errors broadly to minimize the number of code paths and enable common recovery
- Design so that as much of the code as possible is exercised during normal operation.

For example, there are thousands of instances in which FlashArray software allocates memory. Any one can fail if no memory is available. Rather than developing separate recovery code for each one, the software treats any memory allocation failure as controller failure and fails over. The new primary controller discovers outstanding writes from NVRAM and completes them. (Client multipathing retries any commands not yet committed to NVRAM.) The original controller restarts in the secondary role, and relays commands and data to the new primary via the InfiniBand links. When (if) the original controller again becomes primary, it also starts cold, discovering in-flight I/O from NVRAM and completing it during startup.

Perhaps the most obvious example of minimal code paths is the absence of any shutdown procedure. Because a controller must be able to restart after an unplanned shutdown such as a power failure or software crash, there is no real reason for a separate orderly shutdown procedure. Thus, the approved FlashArray shutdown procedure is powering off its controllers, no matter what they are doing. When the array is powered on, it reads its NVRAM to determine what work was in progress at the time of shutdown and completes it.

RAID-3D offers a further example of incorporating what is normally considered error recovery into the main-line read-write path so that it is exercised regularly during daily operation. In a typical array, RAID rebuild code is exercised only when reads fail. In contrast, a client read to a FlashArray may be executed by rebuilding if the SSD containing the data is busy with a lengthy write. Writing flash memory may take as much as two orders of magnitude longer than reading, especially inasmuch as FlashArray SSD writes are 1-megabyte shard buffers. If a

client reads data located on a device busy with a series of 1-megabyte writes, the array reads the necessary logical pages from other SSDs (in parallel) and rebuilds the requested data. This has two benefits:

- Average read performance is more consistent because there are few if any occasions on which a read waits for lengthy writes to complete
- RAID-3D rebuild code is constantly exercised during normal operation, so not only is there motivation to make it perform optimally, there is also less need to devise specific tests for RAID rebuilds because errors in the code are quickly uncovered during main-line I/O testing.

Experience indicates that in a typical FlashArray installation, as many as 15% of reads are executed by rebuilding.

8: Really Usable Snapshots

Snapshots of live data have become an expected storage array feature. Users employ them for everything from creating stable backups of moving data to starting points for disaster recovery, testing, and development. Over time, snapshot technology has evolved to share common content with parent volumes, and to produce independent writable *clone volumes*. (Users generally place high importance on clones because they enable realistic test and development scenarios.)

FlashArray snapshots are unique because:

- They are immutable. Administrators can destroy snapshots and eradicate their content, but as long as a snapshot exists, its content never changes
- They are only visible to array administrators. Clients cannot mount them as read-only devices. They are administration objects, not a special type of read-only volume.

But FlashArray snapshots *are* the basis for clones. Array administrators create clones by *copying* snapshots to new or existing volumes. Snapshot immutability means that every volume created from a given snapshot starts with identical content, so for example, all tests that use volumes created from a particular snapshot start out with identical contents.

FlashArray volumes are organized into *protection groups* (*pgroups*) for snapshot purposes. Pgroup snapshots are *atomic*—all volumes in a group are snapped at the same logical instant. This allows administrators to and place data sets on volumes solely according to application requirements. Making a pgroup of an application's volumes guarantees that snapshots are atomic with respect to all of its data.

Array administrators manage snapshots of entire pgroups. It is not possible, for example, to destroy a snapshot of a single volume in a multivolume pgroup. One can, however, clone a single volume, or copy a snapshot over its parent volume to *roll back* the volume's contents. The latter can be used to recover from data corruption. By rolling back volume contents to a snapshot that predates a data-corrupting event, integrity can be restored. Valid operations can then be replayed to recover work lost on account of the event.

Highlights

- Snapshots and clones are generally regarded as necessities for enterprise storage arrays
- FlashArray snapshots are immutable, and only visible to array administrators
- Arrays take snapshots of administrator-defined protection groups of volumes. All volumes in a protection group are snapped atomically
- Administrators can copy snapshots of individual volumes to create clones or to roll the contents of a volume back to a recovery point
- Snapshots can be ad hoc or scheduled at regular intervals. Scheduled snapshots are destroyed after a retention period
- When an administrator destroys an ad hoc snapshot, there is a 24-hour grace period during which the snapshot can be reinstated
- Arrays use snapshots to implement a 24-hour eradication delay when volumes are destroyed. During the delay, an administrator can recover the volume.

FlashArray immutable snapshots are the basis for cloning and rolling back volumes, as well as for making destroyed volumes recoverable for 24 hours.

FlashArray snapshots are fast—taking one amounts to creating a few persistent data structures (see Section 4), so elapsed time is comparable to that of a typical client write (less than a millisecond). Administrators can therefore use snapshots liberally, for example to create frequent application recovery points, or to perform incremental backups while applications operate on live data. These facilities may be constrained, or even unavailable, with other snapshot technologies.

FlashArray administrators can use the GUI or CLI to create *ad hoc* snapshots at any time, for example to meet unforeseen data protection requirements, such as might precede hardware or application upgrades. More commonly, administrators schedule snapshots to be taken automatically at regular intervals. An array keeps

scheduled snapshots for a *retention period*, after which it automatically destroys them. Once a snapshot schedule has been created, the array manages application data recovery points automatically.

By default, arrays do not destroy ad hoc snapshots automatically; unless retention is specified at creation time, they exist until explicitly destroyed. Destroying an ad hoc snapshot initiates a 24-hour *eradication delay* grace period during which it can be recovered. An administrator can override the grace period, and *eradicate* a snapshot, for example, to initiate free space reclamation if necessary.

Finally, snapshots protect against erroneous destruction of volumes. Destroying a volume implicitly creates a snapshot with the usual 24-hour eradication delay period during recovery is possible. After 24 hours the implicit snapshot is automatically eradicated, and storage unique to it becomes eligible for reclamation. An administrator can terminate the 24-hour period by explicitly eradicating the volume, which eradicates the implicit snapshot. This is another example of keeping array software simple using a single path to serve multiple purposes (Section 7).

9: Realistic Space Accounting

While data reduction clearly helps make AFAs affordable, it makes accurate reporting of storage utilization challenging. This is particularly true for FlashArrays, which continuously reduce long-lived data. In addition, FlashArray space accounting reports generally lag client activity—writes that increase storage utilization and deep reduction that reduces it are not immediately reflected in reports.

Pure Storage distinguishes between *data reduction*—the difference between the number of bytes written by clients and the amount of physical storage required to represent them—and *thin provisioning*—virtual storage that is apparent to clients, but for which no physical space has been allocated because no data has been written to it. FlashArray space accounting reports make this distinction clear by displaying values for both **Data Reduction** and **Total Reduction**. An example may clarify:

- An array administrator creates a 100 gigabyte volume and connects it to clients
- Clients write data to 10 gigabytes of unique LBAs in the volume
- The array compresses the data to half its nominal size, so it occupies 5 gigabytes of SSD space.

FlashArray volumes are inherently thin-provisioned—no physical storage is allocated for an LBA until a client actually writes data to it. Thus, the array in this example does not allocate any storage for the 90 gigabytes of LBAs to which no client has written data. Since no client data has been written to them, they do not figure in the data reduction calculation. Space accounting reports display:

Total reduction: 20:1
100 gigabytes of virtual space visible to clients occupy 5 gigabytes of physical storage.

Data reduction: 2:1
10 gigabytes of data written to unique LBAs occupies 5 gigabytes of physical storage.

When evaluating vendors' data reduction claims, users should distinguish between (a) reduction of actual client data and (b) apparent but never-used LBAs for which no storage has been allocated. This is particularly important for forward planning. In the example above, the array might reasonably be expected to hold 200 gigabytes of similarly reducible data, not 2 terabytes as the total reduction figure might naively suggest.

Unmapping further complicates space accounting. Block storage arrays have no awareness of data context—once a client writes an LBA, it occupies space, even if the client deletes the file of which it is part. As thin provisioning became common, this became problematic. Once data was written, it would occupy physical storage even after the files containing it were deleted.

To counter this, standards organizations specified, and array developers implemented, *unmapping* to allow clients to instruct an array to deallocate the physical storage corresponding to specified LBAs. Unmapping is essential for thin provisioning arrays. Used regularly (most file systems unmap either automatically or via administrative utilities), it ensures that physical storage is used for data that clients value. But it can make space accounting

Highlights

- FlashArray space accounting reports distinguish between data reduction and unused thin-provisioned virtual space
- Array space accounting reports do not include unmapped space in the data reduction ratio
- Space accounting reports include the virtual size of clone volumes in the **Total Reduction** calculation, but only data unique to a clone is included in the **Data Reduction** ratio
- FlashArray **Shared** space represents physical storage occupied by data that is part of two or more volumes and/or two or more volumes' snapshots. Shared space cannot be reclaimed until all objects that use the data in it have been eradicated.

Reporting actual Data Reduction separately from Total Reduction gives users the ability to predict how much data an array can be expected to accommodate after reduction.

ambiguous. Building on the example above, if a client unmapped 5 gigabytes of previously written LBAs, FlashArray space accounting would report:

Total Reduction: 40:1
100 gigabytes of provisioned space occupying 2.5 gigabytes of physical storage.

Data Reduction: 2:1
5 gigabytes of unique LBAs containing client data occupying 2.5 gigabytes of physical storage.

This illustrates the point that unmapping LBAs does not change the reduction of the data that remains. Again, users comparing arrays should determine how they report unmapped space. Some argue that unmapped space represents obsolete client data, so its inclusion in data reduction calculations is justifiable. (In the foregoing example, this would change the **Data Reduction** ratio to 4:1). But as with thin provisioning, this is deceptive for forward planning. If at some future time, clients do write data to previously unmapped LBAs, it would occupy physical storage would have to be accounted for.

Snapshots are another space accounting dilemma, particularly for arrays that support writable clones. A space-saving snapshot shares unmodified data with its parent volume and with other snapshots created from it. Arrays that support clones present LBA spaces for both them and their parent volumes, but at least initially, all of the data in those spaces is shared.

For example, if an array creates 4 clones of the abovementioned 100 gigabyte volume, it presents a total of 500 gigabytes of virtual space. But until data in the clones is modified, volume and clones together occupy only 5 gigabytes of physical storage. In principle, an array could report this as 100:1 reduction.

But this calculation is deceptive. The reason for cloning is to allow modification of data without affecting the parent volume. As time passes, volume and clone contents diverge, and reducibility of live data becomes more of a factor in the array's reduction ratio. Counting all exported LBA spaces independently makes for a poor predictor of an array's capacity to absorb data of a given type.

FlashArray immutable snapshots largely avoid this potential space accounting pitfall. Because snapshots are not exported, their virtual capacity does not figure in an array's **Total Reduction** calculation.

When a clone is created from a FlashArray snapshot, the array's exported virtual space *does* increase by the size of the clone, but physical storage utilization does not increase at all. Arrays expose the overlap between parent and clone volumes by reporting a **Shared** space category that represents physical storage occupied by data that is part of two or more volume (or snapshot) objects. Distinguishing shared space from space unique to a single volume or snapshot allows an administrator to determine roughly how much storage can be freed by destroying an object—space occupied by shared data cannot be freed until all objects that share it have been eradicated.

10: RAID for Flash Storage

The value of RAID for recovering from storage device failures and read errors is so overwhelmingly obvious that the technology has become ubiquitous. RAID enables array designers and users to balance protection quality against cost by adjusting RAID group sizes and/or checksum algorithms.

Typical RAID implementations assume that (a) device failures and unreadable sectors are equally probable, (b) undetected read error probability is vanishingly small, and (c) all devices in a RAID group have the same capacity. None of these is characteristic of flash storage. With no mechanical parts, SSDs rarely fail completely. On the other hand, flash media lifetime (measured in *program-erase cycles*) is limited, and SSD firmware is extremely complex, so undetected read errors—both incorrect data reported as correct and data delivered from the wrong location-- are more of a concern.

RAID also affects I/O performance, especially when rebuilding data after a failed read or device. Here as well, the concerns are different for rotating disks and flash devices. For example, rebuilding data from a failed disk read is a last resort because of the latency it incurs, whereas reading from flash is so fast that it is often preferable to rebuild data to satisfy reads from devices that are busy writing.

General guidelines for flash-optimized RAID might be:

- Provide strong protection against both undetected read errors and mis-positioning errors
- Use (fast) reads to minimize (slower) writes
- Optimize for localized failures such as undetected read errors
- Minimize the impact of rebuilding entire devices
- Adjust protection dynamically based on available devices and/or the type of data being stored
- Provide consistent RAID protection for multiple device types and capacities in an array.

Highlights

- RAID-3D is optimized for flash storage, not rotating disks
- RAID-3D protects each segment of data and/or metadata independently of others
- Each segment is protected by parity, a Reed-Solomon polynomial residual, and checksums on each of its 4-kilobyte logical pages
- RAID checksums protect against two simultaneous read failures in a segment
- Logical page checksums protect against read errors not detected by SSD mechanisms as well as mis-positioning errors
- Arrays allocate segments semi-randomly, favoring SSDs with more free space in order to equalize writes across the array
- Arrays may satisfy reads from SSDs that are busy with lengthy writes by rebuilding
- Segment-level RAID protection allows arrays to support SSDs of different capacities, including non-disruptive device upgrades.

RAID-3D protects all data and metadata in an array automatically and transparently. There are no administrative requirements.

FlashArray RAID-3D does all of these. The key property that distinguishes it from disk-oriented RAID is that it protects each *segment* of data and/or metadata independently. (A segment consists of three or more dynamically selected 8-megabyte *allocation units* (AUs) located on different SSDs.)

An AU consists of eight consecutive 1-megabyte *shards*. Each set of corresponding shards in a segment's AUs comprises a *segio*, the unit for buffering, RAID-protecting, and writing data. Dual checksums (parity and Reed-Solomon) protect the shards in a segio against two simultaneous read failures.

Arrays treat each shard as a column of 4-kilobyte *logical pages*. The third D in RAID-3D is a checksum over each logical page. Checksums are *salted* (initialized) with the pages' SSD locations and segment IDs so they verify not only data correctness but also that the intended page was read. Thus, each logical page is protected against both undetected read errors and mis-positioning by devices.

Arrays allocate AUs for a segment from space on the SSDs within a *write group*. Nominally a write group consists of the SSDs in half of the bays in a shelf or in a FlashArray//m chassis. Arrays store group membership metadata on the SSDs themselves, so once a group is formed, devices can be moved anywhere in the array's SAS network. (While moving SSDs is possible, but not generally recommended for production arrays, because it increases evacuation time.)

Arrays allocate AUs semi-randomly, with a bias toward newer devices and devices with more free space. Over time, this distributes writes evenly, and therefore balances wear across devices. (SSDs' internal mechanisms equalize wear within a device.)

Arrays choose the number of AUs for a segment based on (a) the number of SSDs in the (randomly) selected write group, and (b) the type of data or metadata to be stored. The number is always at least two less than the size of the write group to ensure that two devices are eligible in case a double rebuild is required.

Segments with more AUs use physical storage more efficiently, but for certain types of data and metadata, smaller ones are preferable. For example, background deduplication consolidates duplicate data descriptors in narrow segments to maximize the *appeal* to GC. It consolidates duplicate data separately because it tends not to change over time. Duplicate data segments contain fewer than the maximum number of allocation units so they provide enhanced protection for the data they contain.

Protecting individual segments also speeds rebuilding. When an SSD does fail, arrays rebuild only live AUs to restore full protection. Moreover, because arrays allocate AUs for rebuilding randomly, there is no rebuild target bottleneck such as occurs with device-level RAID.

Section 7 describes how logical page rebuilds satisfy reads directed at SSDs busy with lengthy writes. This improves both average latency and RAID robustness, the latter because it places RAID rebuild code in a path that is frequently exercised during normal operation.

Finally, segment-level RAID allows an array to support devices of different capacities—a flexibility that is not generally available with device-level protection. This is particularly relevant given the velocity of flash technology evolution. By the time SSDs are due for replacement (typically 5+ years for FlashArray devices), technology is almost certain to have evolved. Replacing an SSD with a larger one is seamless: the new device becomes part the single storage pool. Allocation favors it because (a) it is newer, and (b) initially, it has a higher percentage of free space. Over time, fill percentages, and therefore writes, equalize across all SSDs, so wear and back-end I/O tend to balance.

Unlike conventional RAID implementations, RAID-3D is completely automatic—there are no administrative tasks related to RAID protection. Arrays choose the optimal size and layout for each segment as they allocate it. This simplifies administration compared to conventional arrays, and at the same time optimizes protection, because it is (a) highly granular, and (b) based on conditions in the array at the time of allocation.

11: Security: *All the Data, All the Time*

FlashArray Data Security

Potential threats against the security of data stored in FlashArrays include: (a) unauthorized administrative access, (b) unauthorized access to data, (c) recovery of data from misappropriated SSDs and NVRAMs, and, (d) recovery of data from decommissioned arrays. This section describes how FlashArray software protects against each of these threats.

Highlights

- Only credentialed administrators can interact with FlashArrays. Local passwords, PPK, and Active Directory are supported. Each administrator has a *role* (array, storage, or read-only) that determines permissible operations
- FlashArray SSDs power on locked. Every 24 hours, arrays create, partition, and store new *access keys*. More than half of an array's SSDs are required to reconstruct access keys
- Arrays use either device or software-based AES-256 to encrypt all data and metadata on SSDs and NVRAMs. They partition their software data encryption keys, distribute the partitions across SSD headers, and re-encrypt them each time they regenerate SSD access keys
- Optional Rapid Data Locking (RDL) enhances data security with hardware *security tokens* that use a write-only passphrase to enhance software secrets
- When Pure Storage decommissions an array, it securely erases its SSDs. Secure erasure overwrites headers, and erases all erase blocks, including those not exposed outside the devices. Secure erasure obliterates access keys, array encryption keys, and devices' internal randomization/encryption keys, as well as erase block contents
- FlashArrays are FIPS 197 and FIPS 140-2 Level 3 compliant.

SSD access locking and data encryption are always enabled. No administration is required.

Administrative Access

Administrators interact with FlashArrays remotely, using either the command line interface (CLI) running in a terminal console or the Graphical User Interface (GUI) running in a browser on a workstation or mobile device. Login access is restricted to credentialed administrators; credentials may be either a public/private key (PPK) pair or a password validated locally within the array or by integration with an organization-wide Active Directory (AD) server. Each account is associated with a *role* (array, storage, or read-only) that corresponds to a set of permissible actions.

Arrays record both successful administrator logins and failed login attempts. The information recorded includes the source (IP address) and interface mechanism (CLI, GUI, or REST) used. Array administrators can use the GUI or CLI to audit login attempts. For arrays connected to syslog servers, administrator login records can also be audited there.

All communication between browsers and arrays uses HTTPS for authentication and for encryption of the information exchanged; arrays do not support unencrypted http. Users can supply their own SSL certificates for authentication. Pure1 supports TLS encryption (SSL is also supported to accommodate older array software versions).

Finally, FlashArrays implement SNMP v3, with its secure protocols for authentication and communication, to interact with data center monitoring and management facilities.

FlashArray storage and array administrators perform administrative functions such as creating and resizing volumes and connecting them to hosts. They do not have read-write access to data stored in an array. Thus, while an administrator *could* obliterate data by eradicating volumes without authorization, there is no way to retrieve or alter a volume's content without access to a host that can be physically and logically connected to the array.

When a Pure Storage Technical Support technician needs access to an array, an administrator must first enable a *remoteassist* session. To prevent unauthorized access, technicians perform a challenge-response procedure using a one-time password generated by the array and forwarded to the technician for response via a secure network path that verifies that the respondent is indeed a Pure Storage representative.

Host Access to Data

FlashArrays store *sectors* of data located at logical block addresses (LBAs) that are organized as *volumes.* They present the volumes to *hosts* (client computers) for reading and writing over Fibre Channel and iSCSI storage networks, both using the SCSI mass storage command protocol. Storage network administrators typically provide the first level of FlashArray data access security by configuring Fibre Channel *zones* or Ethernet *VLANs* for iSCSI, both of which limit intercommunication to specified host and array network ports.

FlashArray administrators implement the second level of data access security by creating logical *connections* between *host objects* (client computers identified to the array by their storage network addresses) and volumes. Until an array administrator establishes a host-volume connection, the array does not respond to any commands from the host to the volume.

Thus, in order for a host to gain unauthorized access to data stored in a FlashArray, (a) at least one of the host's storage network ports must be included in a zone (Fibre Channel) or VLAN (iSCSI) that includes the array's ports, and (b) an array administrator must create a corresponding host object and establish a connection between it and the volume(s) to be accessed.

Protecting Data on SSDs and NVRAMs Removed from an Array

FlashArrays use the AES-256 algorithm to encrypt all data that they store on SSDs or hold temporarily in NVRAMs. With SSD models that are certified to conform to the relevant FIPS encryption specification, arrays use the devices' internal encryption facilities. For other SSD models, arrays software-encrypt using special processor instructions to minimize computational load. Thus, an SSD need not be AES-256 capable to be supported as FlashArray storage.

An SSD must be properly initialized before it can be added to an array. When an initialized SSD is added to an array, the array creates and stores a device-specific *access key*; without which data on the device cannot be read or written. Every 24 hours, or whenever SSDs are added or removed, the array's primary controller generates a random *secret* and uses it to create a new access key for each device. The controller *partitions* the secret, and stores each partition in an SSD header. (SSD header information is accessible even when devices are locked.) The partitioning is such that two more than half of an array's SSDs must be available to regenerate the secret and recreate the current access keys. SSD access keys are never exposed on any array interface.

Arrays use *data encryption keys* to encrypt data stored on SSDs and held in NVRAMs. For FIPS certified models, the devices themselves encrypt data, using their built-in *device data encryption keys*. For other models, array software encrypts data, using an *array data encryption key* generated during array installation. Arrays partition the key, encrypt each partition using an SSD access key, and store the partition in the corresponding SSD's header. Partitioning is such that two more than half of an array's devices must be available to (a) regenerate the current secret, (b) recreate the access key for each device, and (c) decrypt the array data encryption key. An array's data encryption key is constant for the life of the array, but it is re-encrypted each time the array creates new device access keys. Like SSD access keys, array data encryption keys are never exposed on any interface.

Rapid Data Locking

Where physical security is a concern, for example while transporting arrays that contain sensitive data, optional *Rapid Data Locking* (RDL) hardware can be added. With RDL, customer-programmable *Spyrus Rosetta II Smartcards* in USB readers act as *security tokens* that enhance SSD access keys and participate in key regeneration at power-on. Once unlocked, SSDs are accessible until power cycled, but when smartcards are removed from an RDL-enabled array, its SSDs cannot be unlocked at power-on. Thus for example, if an array is transported separately from its smartcards, data on its SSDs cannot read or written, even if some or all of them are connected to a separate computer running the software algorithm for generating access keys.

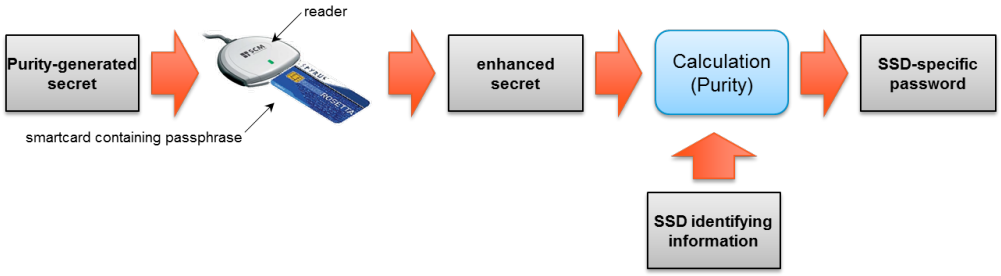

Figure 11.2: Enhancing SSD Access Keys with RDL

If smartcard readers containing initialized smartcards are connected to an array, RDL can be enabled during installation, or at any time thereafter. Once enabled, RDL is permanent. It applies to all of an array's SSDs, including those added afterward.

If a single smartcard is lost or destroyed, replacement is possible. Loss or destruction of both smartcards, however, makes all data in an array permanently inaccessible. For this reason, Pure Storage recommends that additional (properly initialized) smartcards be *escrowed* (stored securely, separately from the array) to avoid data loss if the primary cards are lost or destroyed.

The *FlashArray Rapid Data Locking (RDL) Guide*, Pure Storage part number 40-0058, describes installation and use of the smartcard initialization program.

Array Decommissioning and Permanent Removal of SSDs

When Pure Storage decommissions a FlashArray for removal from customer premises (for example when a proof of concept evaluation finishes) a technician uses one of the array's controllers to *securely erase* all of its SSDs. During secure erasure, the controller (a) overwrites SSD headers, including the partitions of the shared secret used to regenerate access keys and to decrypt the array's AES-256 data encryption key, and, (b) erases all SSD erase blocks, including overprovisioned ones that are not exposed outside the devices. In addition to data, block erasure obliterates the internal randomization keys that SSDs use to scramble stored data for uniform 0/1 bit distribution regardless of the content written by hosts (self-encrypting SSDs use their internal encryption keys for this purpose), and destroys the *flash translation layer* (FTL) tables that map exported sector addresses (LBAs) to storage locations within the devices.

Typically, blocks of flash memory can be erased, even if they are "bad blocks" that cannot be read or written reliably. Even if block erasure were incomplete, however, any residual data on a FlashArray SSD would be software-encrypted (using a key destroyed during header overwrite), or encrypted by the device (using an internal key destroyed during block erasure). Finally, because block erasure destroys the exported LBAs that correspond to physical device locations, there would be no way to associate any data remnants with the LBAs to which they had originally been written.

In addition to this decommissioning procedure, Pure Storage Technical Support offers an optional SSD erasure service for situations in which organizational policies require additional data destruction measures.

At the time of publication, FlashArrays are certified for the FIPS 197 and FIPS 140-2 Level 3 information security standards published by the US Federal government. Work on additional certifications, such as US federal government Common Criteria certification is ongoing.

12: Metadata: The Secret Sauce

Much of FlashArray uniqueness lies in the media layout, and the robustness, space efficiency, scaling, performance, and functionality enabled by the metadata that describes where and how data is represented.

Robustness

FlashArray metadata describes both client data and array configuration and state. Metadata is both protected by RAID-3D and redundant, with different representations in at least two locations so that if a catastrophic error occurs in one, the affected information can be reconstructed from the other.

For example, as an array buffers client blocks prior to persisting them, it inserts their locations (segment ID, offset) in its cbmap and also stores them in the buffer with the data. Because each segment contains a complete description of the data in it, including a generation number for each block, an array can reconstruct its cbmap if necessary.

As a second example, when an array flattens its cbmap pyramid, it retains the old levels until the next flattening cycle so that if the new (flattened) level is corrupted, the array can still use the old levels to locate data and metadata.

More generally, like client data, metadata is effectively log-structured—it is not directly overwritten. Thus, superseded metadata is usually available for recovery is newer, not-yet-persisted metadata is destroyed.

Highlights

- FlashArray metadata is both redundant and protected by the same RAID-3D mechanisms that protect client data
- Arrays reduce data by eliminating patterns and duplicate blocks and compressing what is unique
- Arrays use space efficiently by packing data representations densely in segments and using cbmap metadata to locate them
- The metadata design partially decouples metadata space requirements from the amount of data stored in an array
- Arrays augment persistent metadata by caching frequently-accessed structures such as cbmap indexes and sector hash signature tables to improve performance
- Persistent metadata augmented by cache will allow array developers to balance array cost and performance in the future.

The FlashArray metadata design enables much of the arrays' robustness, space efficiency, scaling, performance, and even functionality.

Space efficiency

Most of a typical array's media is occupied by client data representations, so efficient data storage is a paramount design priority. FlashArray metadata optimizes client data representation in three fundamental ways:

Data reduction:
Arrays reduce redundancy in client data by (a) replacing repeating byte patterns with descriptors, (b) replacing duplicate blocks with pointers to a single instance, and (c) compressing unique data. As Section 4 describes, arrays reduce data in two stages—upon entry, and later in the background. Two-stage reduction minimizes the impact on client write response and eliminates reduction of transient data.

Compact storage:
The internal representation of a client block (pattern descriptor, duplicate pointer, or compressed data) occupies exactly as much space as it requires. There is no padding to sector or other fixed boundaries. Each block is buffered and stored immediately adjacent to the preceding one. The cbmap relates the LBAs used by clients to media locations (segment IDs and offsets).

Factoring:
Arrays represent snapshots and client xcopies as *medium maps* that concisely describe large amounts of data that are part of two or more objects. Changes to the data described by a map are represented as exceptions. As long as the majority of data shared by two or more objects is identical, it is represented by a single instance pointed to by multiple medium maps.

Scaling

Arrays that manage space in fixed-size blocks require metadata in proportion to the amount of storage they support. The FlashArray architecture uses three techniques to decouple metadata requirements from the amount of physical storage they support:

Variable representation:
The metadata in a cbmap contains the locations of blocks of client data. Each map entry describes as large a block as possible (with a maximum of 64 sectors), based on client write size and/or duplication.

Describe only actual data:
A cbmap contains only the locations of blocks that clients have written. No metadata space is reserved for unused LBAs, or LBAs that have been trimmed by a client.

Only cache frequently-used metadata:
Arrays store metadata persistently, but cache frequently accessed structures to minimize lookup times. As they persist metadata, they compress it to minimize its volume. Thus, metadata DRAM requirements grow more slowly than the amount of physical storage connected to an array.

Performance

Arrays cache frequently accessed metadata to optimize performance. For example, they cache cbmap indexes to minimize search time during client reads, and hash signatures for high-probability duplicate sectors to detect duplicates immediately with minimal impact on client response time. (Deep reduction in the background is more thorough, but in practice, the majority of duplicates are identified as they enter an array.)

Two-tier (cached and persistent) metadata allows developers to balance performance against cost: more DRAM increases array cost, but more metadata in cache leads to higher performance. Less DRAM minimizes cost, but at the expense of performance, because metadata must be retrieved from persistent storage more frequently.

Functionality

Finally, the structure of FlashArray metadata, notably the use of medium maps to describe large swathes of data, is in large part what makes efficient xcopies, snapshots, and replicas possible. As described in Appendix A, a medium map initially describes an arbitrarily large amount of data by reference to an array's cbmap. Two or more maps describing the same data can represent different volumes or snapshots. Thus, to create a snapshot or clone volume, an array creates and persists two new medium maps—one to describe the parent volume or snapshot and the other to describe the new object—a process that takes about as long as a typical client write.

13: Toward the DIY Storage Array

An Pure Storage guiding philosophy that can get lost among the performance, robustness, and cost-effectiveness headlines is *simplicity*. The simplicity is obvious to administrators; many common tasks either do not exist or are significantly less complex in FlashArray environments. Most of these are discussed elsewhere in this paper; this section summarizes them to illustrate the focus on simplicity.

Highlights

- The Pure Storage philosophy is to reduce storage administration to tasks that are related to the user's IT mission
- FlashArrays eliminate or greatly simplify most routine storage administration tasks
- Non-disruptive hardware and software upgrades are already being performed by trained Pure Storage and partner technicians
- The company's goal is to extend the arrays' day-to-day administrative simplicity so that users can perform non-disruptive upgrades
- Eventually, all array tasks, including installations, will be simplified and error-proofed for end user execution.

Come for the performance; stay for the simplicity.

RAID Group/Spare/Cache Configuration and Management

Administrators of conventional arrays configure RAID groups and reserve spare capacity. In doing this, they make implicit judgments about the relative value of datasets and the reliability of array components—judgments that should be made by application owners and array vendors respectively.

The FlashArray architecture completely eliminates RAID configuration. Arrays protect all data and metadata against a minimum of two read failures, adjusting dynamically based on array state. RAID protection is completely automatic; no administrative action is required, or indeed, possible.

Similarly, FlashArray administrators do not manage cache. Partly this is due to flash read performance, which obviates much of the need for read cache. Partly it is due to the use of NVRAM to defer writes. But mostly it is due to self-management of the data and metadata that an array *does* cache. Arrays manage the memory that they use to cache cbmaps, sector signature tables, and other data and metadata completely autonomously.

Volume Configuration

As they create virtual volumes, administrators of conventional arrays typically specify the physical devices they will occupy, usually with the goal of influencing performance. A FlashArray administrator creates a volume by specifying its name and size (number of LBAs it presents)…period. Arrays allocate storage only when clients write data, so there is no fixed relationship between LBAs and physical locations for administrators to manage.

Data Encryption and Key Management

Arrays encrypt all data and metadata all the time, both that stored persistently on SSDs and that held temporarily in NVRAM. They use SSDs' build-in encryption where possible, and processor instructions that perform software AES-256 encryption where device encryption is not present or inadequate. In addition, all SSDs are locked at power-on; controllers must use a device-specific key, regenerated every 24 hours, to unlock each one for access.

None of this data security is optional or configurable—every array does it automatically all the time.

Client Connection

In some older arrays not all ports have access to all storage. FlashArrays do not have this restriction—all volumes are accessible on all storage network ports on both controllers. Array-wide volume access minimizes complexity and the potential for storage network configuration errors: all port access restrictions can be managed at the

storage network level. FlashArray administrators connect clients to volumes logically; storage network administrators use Fibre Channel zoning or iSCSI connections to restrict per-port access as required.

Performance Tuning

Fundamentally, the goal of performance tuning is to deploy resources to maximize some desired effect. But demands on a storage array change from second to second; there is no viable way for an administrator to dynamically rebalance available bandwidth, cache, and processing power for optimal utilization.

FlashArrays eliminate routine administrative performance tuning, because arrays themselves are in the best position to make real-time adjustments. For example, they defer SSD writes until they need NVRAM and/or buffer space for incoming data, thus limiting SSD wear. They serialize writes to keep as many devices as possible available for reading. They decide dynamically whether to wait for a busy SSD or rebuild data to satisfy a read. They schedule GC, deduplication sweeps, cbmap flattenings, and other internal tasks based on client load. The frequency of these tasks and the detailed knowledge they require make human scheduling impossible.

Fault Management

FlashArray controllers fail over autonomously in seconds, with no loss of data or I/O in flight, and with minimal performance impact when a controller fails. Arrays detect and correct SSD read errors at the logical page level. When an SSD does fail, the array rebuilds only its live data, and distributes rebuilding so there is no write bottleneck and no requirement for after-the-fact rebalancing.

Through the Pure1 Support facility, Pure Storage provides *phone home* problem detection and remediation. In addition, however, Pure1 Support servers continuously match customer arrays' configuration and performance profiles against fingerprints of known issues to identify arrays that might be susceptible to them and suggest preventive action to users and partners.

Beyond Day-to-Day Administration

The foregoing paragraphs illustrate how Pure Storage enhances data center productivity by simplification—by eliminating routine tasks unrelated to the user's IT mission. Ultimately the company plans to extend this simplicity to encompass array installations and hardware/software upgrades, with the goal of making them so simple that users will routinely perform them in-house rather than requiring assistance from Pure Storage or its partners.

Already, FlashArray hardware and software upgrades are non-disruptive—trained technicians can and do perform them on arrays that are servicing client I/O requests. The company is continuing to simplify and error-proof upgrades and other infrequent operations so that they become routine administrative tasks.

Array installation and hardware upgrades inherently require handling of physical equipment. Greater integration, as in FlashArray//m arrays, simplifies rack mounting and cabling, but obviously physical handling cannot be eliminated entirely. Through a combination of physical guidance (e.g., extending the current color/shape coding used today on cables to other components) and automated step-by-step checking for errors, the company intends to make installation and hardware upgrade simple enough to be routinely performed by end users/

14: Fixing It Before It Breaks

As storage arrays have become both more complex and more mission-critical, dealing with the faults that are inevitable with any physical equipment has become a key issue for users and vendors. Today, self-diagnosis, and automatic vendor notification of actual or impending faults are expected features. Some vendors boast of recognizing problems and initiating remediation before users become aware of them.

FlashArrays meet this standard, and go beyond what users expect and what other vendors deliver. Unless restricted by user policy, arrays report configuration information, and performance and health statistics to Pure1 servers every 30 seconds, and in greater detail every hour and 24 hours. The Pure1 database is a long-term record of how arrays are configured, used, and perform in the field.

Like other vendors, Pure Storage responds to alerts that indicate actual or impending problems, often initiating service actions before users are aware of issues. But the company goes beyond case-by-case reaction to problems *after* they occur. Pure1 servers continuously search the historical database for configuration and usage scenarios with *fingerprints* similar to those of arrays in which problematic issues have occurred, and selectively alert users, suggesting preemptive actions that can be taken.

For example, if an FA-450 array (specific model) using iSCSI (specific storage network technology) that is approaching the maximum number of supported snapshots (specific usage) reports a write throttling alert due to low available memory, Pure1 may issue alerts for arrays with similar configurations and usage so they can avoid the condition. For alerts traced to software or firmware issues solved by updates, Pure Storage can issue advisories for just those arrays for which the updates are relevant.

Highlights

- Automated real-time problem reporting is an expected storage array feature
- Arrays continuously report configuration, status, and performance information to Pure1 servers
- When a problem is reported to Pure1, Pure Storage typically develops a fingerprint to describe it.
- Pure1 Support continuously searches its database to identify configurations and usage patterns that match the fingerprints of known issues.
-
- The fingerprint library expands as issues are identified, so the ability to preempt problems improves with time.

Data collected by Pure1 enables Pure Storage to identify potential issues before they occur and recommend preemptive action to users who might be affected.

The key advantage is selectivity. As the FlashArray installed base grows into the tens of thousands, stability and consistency become increasingly important. But a software/firmware update that alleviates, say, an iSCSI problem, may result in unforeseen issues in Fibre Channel arrays. Knowledge that pinpoints the arrays that are likely to be affected by a given issue enables Pure Storage to deal preemptively with them and only them.

An analogy may clarify the value of preemptive support. Imagine that a strong correlation is discovered between children and older people who begin to sneeze and have elevated body temperatures, and a particular strain of influenza virus, but that the correlation is much weaker among working-age adults. With this information, public health authorities can target vaccination programs to where they are likely to be most effective, and also minimize undesirable side effects such as working-age adult allergies to the particular vaccine.

Analyzing Pure1 data to develop fingerprints for common problems is necessarily inexact. Obviously, fingerprint accuracy improves as more instances are encountered, and as root causes are identified. Moreover, the FlashArray installed base is constantly expanding with new array models and capabilities. Thus, using fingerprints to identify potential issues is inherently aspirational. Nevertheless, avoiding issues based on experiences elsewhere in the installed base elevates the FlashArray user experience above those of conventional array users.

15: Evergreen Storage

Storage technology evolves rapidly, but as it does, user requirements for more performance, more capacity, greater robustness, and advanced capabilities seem to evolve along with it. Typically, a combination of technology improvements and maintenance cost make it uneconomical to operate an array for longer than 3-5 years. As a result, data center managers have historically been faced with periodic forklift upgrades of storage array hardware and software. Replacing storage arrays is undesirable because:

It costs a lot:
Users effectively have to re-buy something they already own (storage hardware and software), and dispose of old storage, which typically has minimal residual value.

It entails data migration:
Copying (*migrating*) data from old arrays to new ones requires extensive planning, and typically entails some level of downtime. The more an organization depends its online data for success, the greater the potential impact. Bulk migrations can take months if everything goes well, but there is always a risk that errors and/or unanticipated effects will result in additional (unplanned) downtime. In extreme cases, arrays can be 15-20% through their useful lives before the ones they replace are finally retired.

Despite all of this, users have historically accepted forklift storage upgrades every 3-5 years as an unpleasant fact of data center life.

Highlights

- Users typically replace storage arrays on 3-5 year refresh cycles
- Replacing storage arrays is expensive, time-consuming, and risky, especially in terms of the downtime it entails
- The Pure Storage Evergreen storage model eliminates forklift upgrades—arrays can be upgraded as needed in five independent dimensions: performance, capacity, density, connectivity, and function
- Users can safely plan on decade-long storage life cycles without massive data migration

Forever Flash, with **free every three** *upgrades,* **flat and fair** *service pricing, and a* **forever maintenance** *replacement guarantee, enables users to track technology evolution and meet growing storage requirements without paying again for what they already own.*

The Evergreen Storage Model

The Pure Storage *Evergreen™ storage model* completely changes this paradigm by eliminating forklift upgrades entirely. With the Evergreen model, users can adopt FlashArray storage with reasonable expectations of 10+ year useful array lifetimes. During an array's lifetime, some, or even all, of its hardware and software components may change, but the changes (a) do not require the user to re-purchase what is already owned, (b) do not entail bulk data migrations, and (c) do not disrupt I/O service to clients.

Based on the FlashArray *software-defined storage architecture*, the Evergreen model makes it possible for users to evolve arrays in five independent dimensions:

- Performance, by upgrading controllers
- Capacity, by adding or replacing storage shelves
- Density (space efficiency), by replacing array components with more compact ones
- Client connectivity, by adding storage network interface ports to controllers
- Features, by upgrading array software as new capabilities become available.

The software-defined architecture allows users to respond separately to demands for increases capacity, performance, connectivity, and so forth as need arises or as technology becomes available. Because FlashArray

upgrades are non-disruptive, there is no downtime when capacity is increased, when software is upgraded to new releases, or when controllers are replaced with more capable models. Because arrays are never replaced in their entirety, there is no data migration. Users can plan on downtime-free, risk-free *rapid online upgrade cycles* for the life of an array.

Figure 15.1 illustrates the main features of the Evergreen storage model. The key points of the figure are that (a) all upgrades are non-disruptive and do not entail data migration, and (b) the upgrade dimensions are independent of each other, and any or all can be performed as needed.

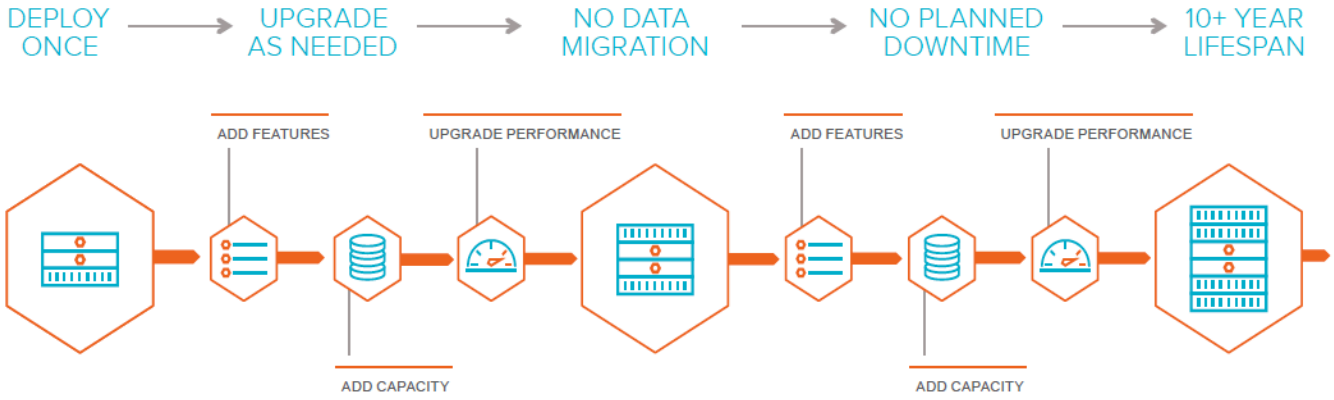

Figure 15.1: The Evergreen Storage Model

Realizing Evergreen Storage

Pure Storage implements Evergreen storage via the *Forever Flash* program. FlashArray owners become part of the program by maintaining on-going service agreements. The three primary features of Flash Forever are:

Flat and Fair **maintenance pricing**: [5]
Maintenance pricing remains constant (inflation-adjusted) for the life of an array; there are no large price increases when contracts are renewed

Free Every Three **controller technology refresh**:
Array maintenance contracts include upgrades to the latest equivalent controller technology every three years; technology stays up-to-date for the lifespan of an array with no service disruption, data migration, or incremental cost

Forever Maintenance **replacement guarantee**:
Maintenance contracts guarantee replacement of any failed array component, including SSDs, for the lifetime of an array.

In addition, for Forever Flash subscribers who require performance or connectivity beyond what was originally purchased, Pure Storage offers *Flex Product Bundles* that replace controllers with more capable ones, at a cost based on the difference between the replaced equipment and the equipment that replaces it.

When additional storage capacity is needed, users can add storage shelves to an array, up to the maximum capacity supported by its controllers. Add-on shelves can be populated with SSDs of the most appropriate capacity and technology at the time they are added. There is no requirement for all of an array's SSDs to have identical capacities.

Similarly, an array's storage network connectivity can be increased by powering off one of its controllers (causing failover to the other), installing an interface card, connecting the new ports to the network, powering the controller back on, and repeating the steps with the other controller. Controllers automatically recognize and use any

[5] The current Pure Storage End User Agreement (part number 40-0001) is the definitive source for *Forever Maintenance* hardware replacement and other terms and conditions, including *Flat and Fair* maintenance pricing.

additional storage network ports when they start up. (At present, interface card installation is a non-disruptive procedure, but is performed by trained technicians.)

Finally, as array capabilities evolve through the addition of new software features and refinement of existing ones, users can take advantage of enhancements by upgrading their array software. Like other upgrades, software upgrades are performed controller by controller while an array is operating.

Because of the *active/active front end, active/passive back end* design (Section 6), not only are hardware and software upgrades non-disruptive, but the effect on array performance is typically negligible. Based on analysis of Pure1 data, FlashArrays achieve more than 99.999% uptime across the installed base.

The Evergreen storage model, backed by Flash Forever, is a completely new way for data center managers to plan, implement, and operate storage. Users can plan for decade+ storage life cycles without disruption or massive data migration. Performance, capacity, density, connectivity, and functionality can all be upgraded independently of each other as needs arise and as technology becomes available. *Flat and Fair* maintenance pricing eliminates the risk that downstream maintenance will be unaffordable. And *Forever Maintenance* means no worries about disruption or data loss due to flash wearout.

In Summary…

This paper describes fifteen ways in which Pure Storage FlashArrays are a breed apart from conventional disk, hybrid, and flash arrays. In a nutshell, these are:

Self-everything:
FlashArrays eliminate most administrative tasks related to physical configuration by making decisions dynamically according to conditions within the array.

Scaling: up vs. out:
FlashArray scale-up architecture is inherently more reliable and cost-effective as capacity requirements increase. Any potential performance advantage of scale-out is seldom utilized in practice, but the cost is always present.

SLC vs. eMLC vs. cMLC:
The FlashArray architecture is designed to utilize low-cost cMLC devices. An advanced media layout and software design optimizes performance and endurance to achieve parity with more costly technologies.

Data reduction: all the data, all the time:
FlashArray data reduction is always on—administrators cannot throttle or disable it. Arrays reduce data in two stages: as it enters the array, and as they garbage collect to retrieve storage occupied by obsolete data. Two-stage reduction minimizes the impact on client response as well as the resources expended to reduce short-lived data.

Stateless controllers:
All of an array's data and metadata, including configuration information, is stored on SSDs or in NVRAM. This makes non-disruptive controller replacement and upgrade possible. With Forever Flash, forklift upgrades are obsolete; over time, the only constant in a Forever Flash storage installation is the data.

The second controller:
The FlashArray architecture is *active/active front end, active/passive back end*. Both controllers receive and respond to client commands, but all processing is done by an array's primary controller. There is no appreciable performance degradation when a controller fails.

Streamlined code paths:
FlashArray software minimizes code complexity and promotes robustness by grouping errors into broad classes with common recovery procedures, and by incorporating procedures such as RAID rebuilding, that are commonly considered error recovery into main-line array operation.

Really useful snapshots:
FlashArray snapshots are immutable images of administrator-defined protection groups of volumes from which any number of clone volumes can be created. Snapshot creation can be ad hoc or scheduled. When a snapshot is destroyed, there is a 24-hour grace period during which it can be recovered.

Realistic space accounting:
FlashArray space reporting distinguishes sharply between **Data Reduction** (space occupied by representations of client data) and **Total Reduction** (the ratio of total exported capacity to occupied space). Space occupied by data shared among multiple volumes and/or snapshots is explicitly reported as **Shared**, allowing administrators to determine approximately how much space can be reclaimed by destroying a given object.

RAID for flash storage:
FlashArray RAID-3D protects all persistent data and metadata completely automatically. It protects individual segments rather than entire devices, and is optimized for flash mamory properties through the inclusion of enhanced detection of page read and mis-positioning errors.

Data security:

FlashArrays use AES-256 to encrypt all data on SSDs and NVRAMs all the time. In addition, SSDs power on locked, so data cannot be accessed without a key. Arrays regenerate and partition keys every 24 hours and distribute the partitions for security. For physically insecure environments, optional Rapid Data Locking hardware allows the key regeneration mechanism to be physically separated from the array.

Metadata: the secret sauce:

Metadata design is a key enabler for much of the arrays' robustness, performance, space efficiency, and functionality. Unlike arrays that utilize direct mapping, FlashArray metadata size is not directly proportional to the amount of storage in an array. This property makes it possible for developers to balance array cost against performance for different models.

Toward the DIY storage array

Simplicity is a fundamental Pure Storage design philosophy. Many of the tasks required to administer conventional arrays do not exist with FlashArrays. At present, upgrades and installations are typically Pure Storage Technical Support or partner-assisted. The company's goal is to simplify these as well, to the point where it becomes practical for end users to perform them routinely.

Fixing it before it breaks:

In addition to the usual automatic problem identification and vendor notification, the company uses configuration, performance, and health data continuously collected by Pure1 servers to identify arrays in which known issues might be prone to occur, and alerts owners and partners so that preemptive action can be taken.

The Evergreen storage model:

FlashArray users can independently upgrade array performance, capacity, density, client connectivity, and functionality, all non-disruptively. Decade-long array life cycles are entirely feasible—over time, all of an array's hardware and software components may change without ever disrupting service. With Forever Flash, users no longer pay again for what they already own.

.

Appendix A: FlashArray Media Structure

The FlashArray SSD media structure and data storage scheme is useful for understanding how arrays represent snapshots and replicas on SSD storage. This appendix is an overview of FlashArray storage organization and the process by which arrays reduce and store client data.

The "Apartment"

Internally, Pure Storage refers to an array's complement of SSDs as its *apartment*. Upon discovering a properly formatted SSD, A controller automatically initializes it and adds it to the apartment. Similarly, controllers eject SSDs from the apartment when they become non-responsive.

An array's SSDs house array configuration information as well as data. They can be moved from one pair of controllers to another with no loss of data or configuration information. FlashArray controllers are therefore *stateless*—they contain no persistent information about an array's configuration or about the data stored in it.

Write Groups

An array manages its storage as a single pool, but it constrains the locations from which it allocates the *segments* in which it writes data. An array's SSDs are organized as *write groups*. When SSD chassis are fully-populated:

- **SH0** and **SH1** of FA-400 series arrays contain two groups of 11 SSDs
- FA-400 series shelves **SH2**-**SH5** and FlashArray//m external shelves contain two groups of 12 SSDs
- A FlashArray//m integrated shelf contains two groups of 10 SSDs.

Allocation Units

Arrays organize storage as 8-megabyte *allocation units* that are boundary-aligned with SSD erase blocks for optimal write performance and endurance. When allocating a segment, an array chooses the allocation units that comprise it from SSDs in a single write group to limit the scope of RAID-3D data protection.

Superficially, SSDs resemble disks in that they present individually addressable 512 byte sectors of storage. Internally, however, each model organizes and addresses pages of flash cells differently. FlashArray software includes a unique *flash personality layer* (FPL) module for each SSD model that Pure Storage qualifies and supports. Each FPL module uses a unique algorithm to map disk-like *logical block addresses* (LBAs) to SSD sectors in a way that optimizes read and write performance for the particular SSD type. This enables the rest of the array's software to treat all qualified SSDs as vectors of logical blocks. As a consequence, it is not necessary that all of an array's SSDs be identical, or even have identical capacities.

Segments

Arrays allocate SSD storage for writing data and metadata dynamically, in units called segments. A segment consists of one allocation unit on each of several of the SSDs in a write group. Arrays choose the SSDs to contribute space to a segment randomly, with a bias that favors newer and less-populated devices to equalize media wear and balance device I/O load. The space in a segment is organized as illustrated in Figure A.1.

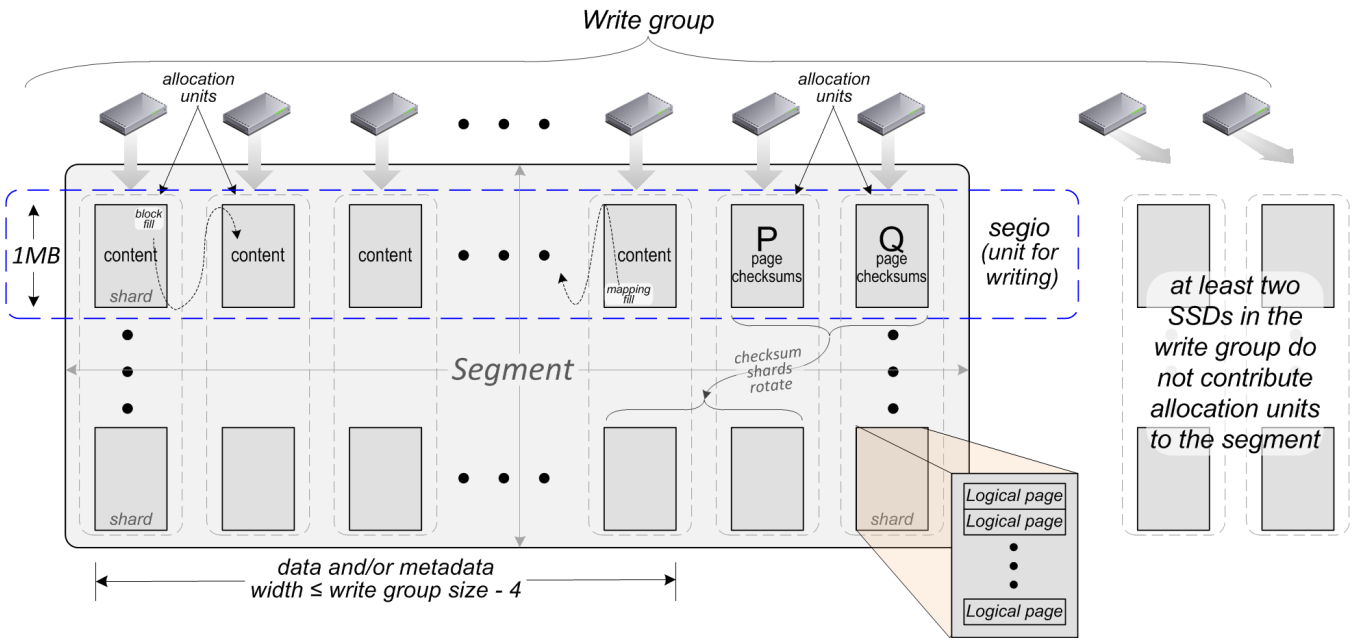

Figure A.1: FlashArray Segment Layout

Inside the Segment

Arrays treat each allocation unit in a segment as a column of eight 1 megabyte *shards*. The corresponding shards in each of a segment's allocation units collectively make up a *segio*, the unit in which arrays buffer and write data.

Arrays treat each shard as a column of 4 kilobyte *logical pages.* Because allocation units are aligned with SSDs' internal organization, each logical page corresponds to one or more SSD *flash pages* (the unit in which SSDs read data and apply ECC protection). As an array fills write buffers with data and/or metadata, it computes and stores a checksum in each logical page. Additionally, it uses two shards in each segio to store RAID *P*+*Q* (parity and Reed-Solomon) checksums that it computes over the contents of corresponding logical pages in the other shards. Arrays initialize logical page checksums with segment and offset identifiers so that they verify not only page contents, but also that the correct page has been read.

> *Arrays use logical page checksums to validate every logical page's contents before using it. Checksums detect both content and position errors that may elude SSD detection and correction mechanisms.*

Appendix B: FlashArray Writing and Reading

Client Write Command Processing

Figure B.1 illustrates the path of data written by a client from the time it enters the array until it is stored on SSD.

An array receives clients' write commands and blocks of data from storage network interface drivers. For each block, it computes a checksum (❶ in Figure B.1) that it will use for end-to-end validation when the block is read. At this stage, it also replaces repeating patterns with descriptors and performs lightweight compression.

The array then writes the blocks' representations to two NVRAMs (❷) for recovery in case of controller failure. Block representations remain in NVRAM until the blocks have been written to SSD media. Once data is in NVRAM, the array sends a completion notification to the client (❸).

Next, the array computes a *signature* for each sector in the block, and compares them to signatures of recently written and known duplicate sectors to identify possible duplicate sequences (❹).

Arrays only deduplicate multi-sector sequences, but per-sector signatures make it possible to detect duplicates (a) of different lengths and (b) of different alignments in volume/LBA space. When an array identifies a possible duplicate sequence, it reads the stored data and compares it to the new to verify that the two are indeed identical.

The array then places the data's representation (patterned block descriptor, compressed non-duplicate data or mappings of duplicates) in write buffers along with metadata that describes it (❺), and calculates RAID-3D checksums over the logical pages it occupies.

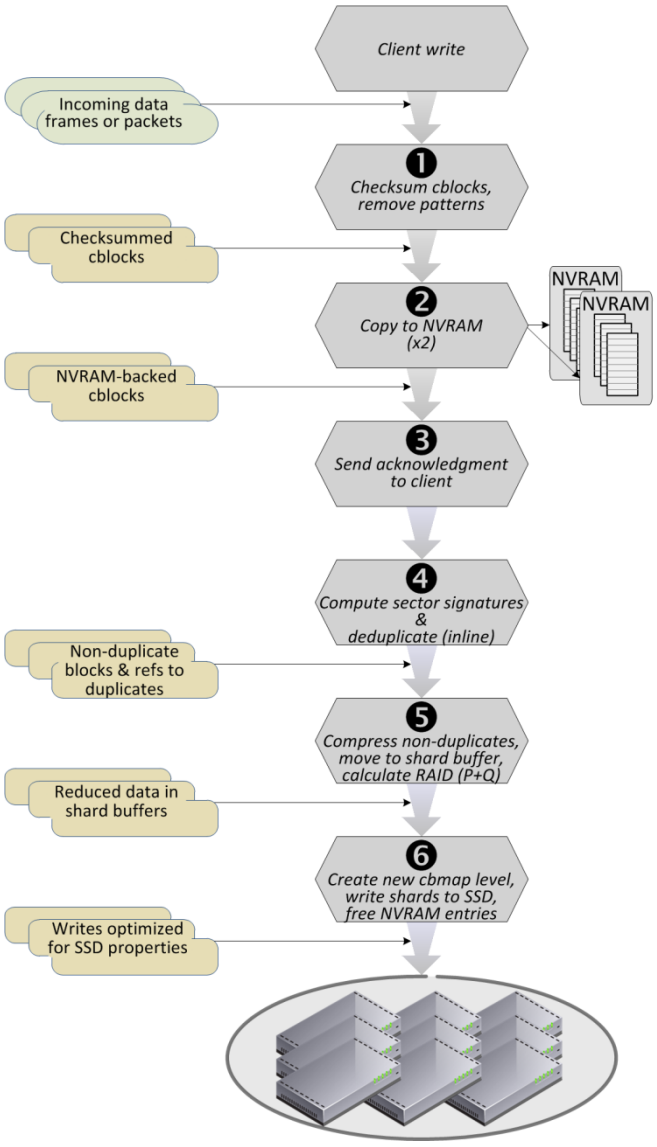

Figure B.1: Client Write Command Processing

When the buffers for a segio fill, the array writes its shards to SSD storage (❻). Arrays *stagger* shard writes (perform one or two at a time) so that as many SSDs as possible remain available to service reads while data is being written.

As it creates each segio, the array adds a new level to the array's cbmap. The new level contains entries that point to the SSD locations of the cblocks in the segio.

When all of a segio's shard buffers have been written, the array frees the NVRAM entries that represent the data in them.

Client Read Command Processing

Figure B.2 illustrates client read command processing.

The first step in processing a clent's read command is locating the requested data:

- Arrays locate cblocks on SSD media by traversing medium extents and the cbmap (❶ in Figure B.2)
- Arrays *lookaside* to locate cached cblocks and buffered cblocks that are awaiting writing to SSD, and process them directly from controller memory (input to ❹ in Figure B.2).

For cblocks located on SSD, the array:

- Reads the logical pages that contain the requested data (❷)
- Verifies logical page checksums to detect any read errors not corrected by SSD mechanisms and unpacks the requested blocks from the pages (❸).

Next, the array expands the data to its original form (❹), and validates the cblock checksum(s) (❺). Finally, the expanded cblocks are handed off to storage network interface drivers for packetization and delivery to the client (❻).

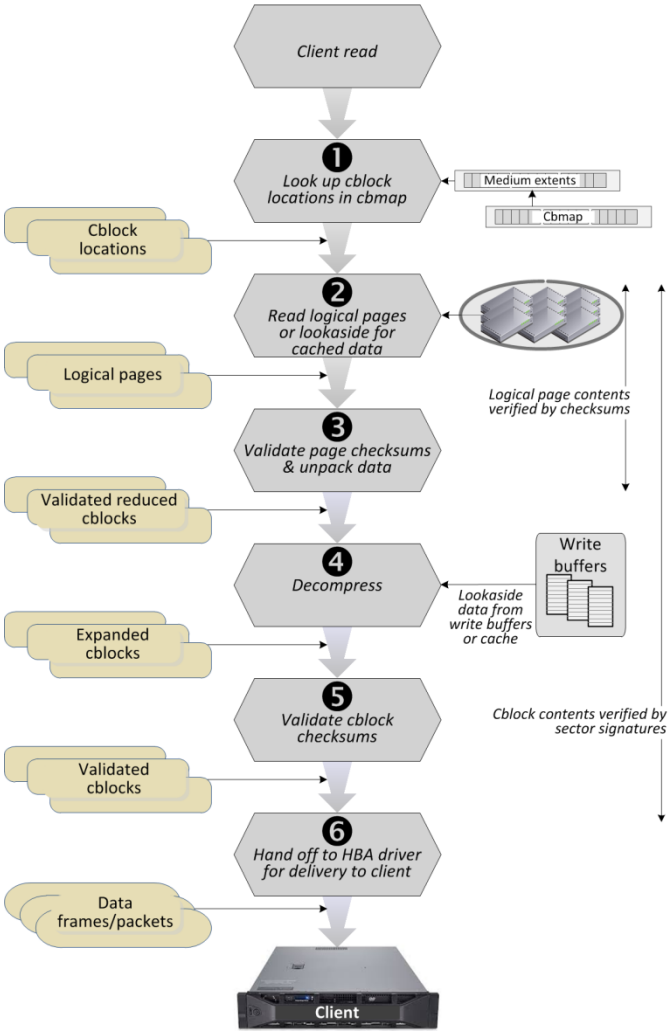

Figure B.2: Client Read Command Processing

Pure Storage, Inc.
Twitter: @purestorage

650 Castro Street, Suite 260
Mountain View, CA 94041
800-379-7873

Sales: sales@purestorage.com
Support: support@purestorage.com
Media: pr@purestorage.com
General: info@purestorage.com