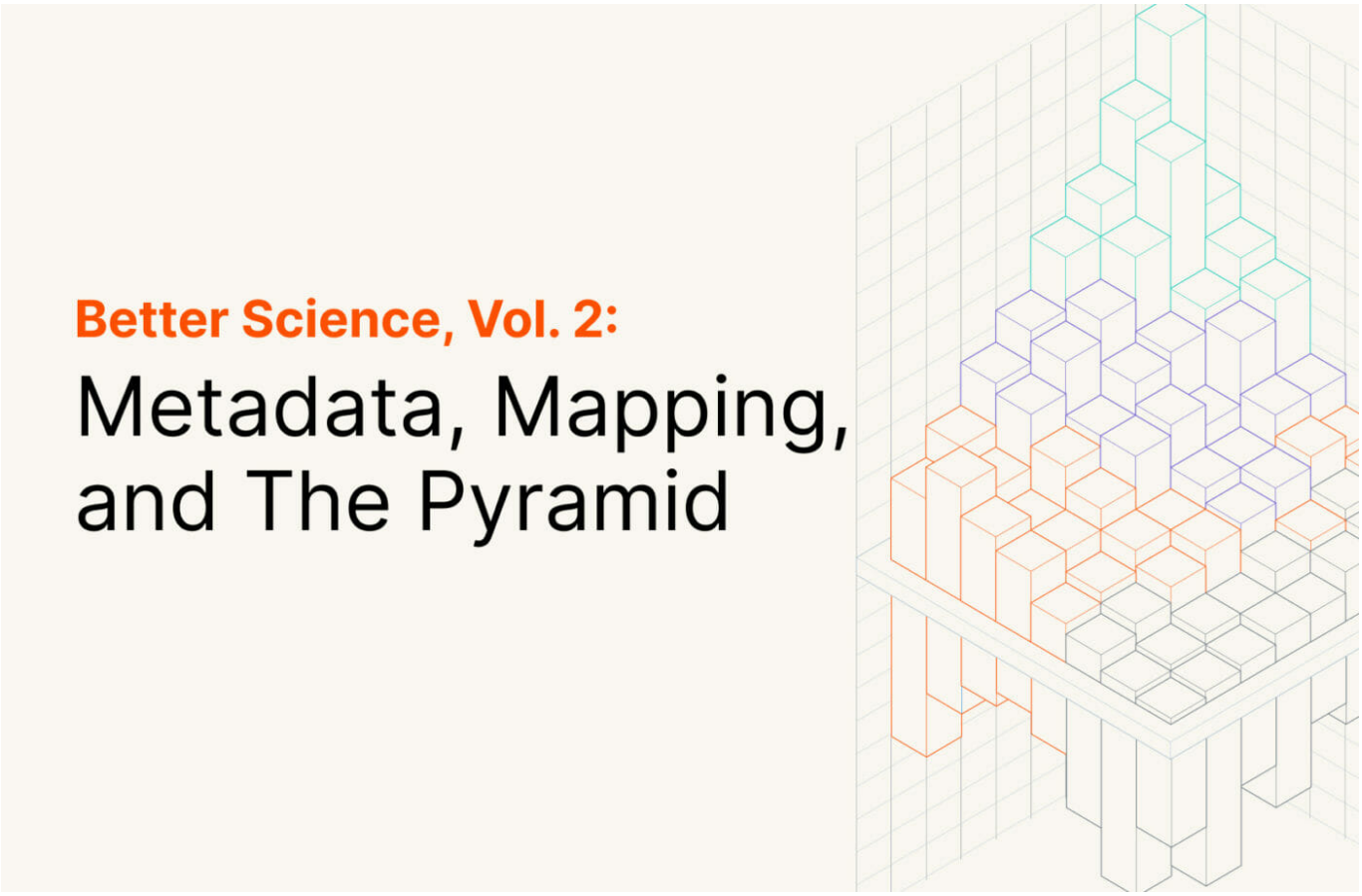
Better Science, Volume 2: Maps, Metadata, and the Pyramid



In our first installment of Better Science, we talked about how Pure Storage® designed a better approach to flash storage systems and the benefits it brings to customers, as well as the environment.

Today, we have a strategy for storing *quadrillions* of bits: petabytes of data, all stored efficiently on the densest flash available, and able to scale even larger in the future. But now that we've established how to store all this metadata, an equally important topic comes to mind:

How do you find anything in this huge ocean of bits?

In this blog post, we're going to cover how the industry at large has approached this problem in the past, talk a little bit about the current state of the art, and how Pure's approach to this mapping problem is uniquely suited to today's modern data landscape.

From Tablets to Maps to Search Bars: A Brief History

As long as there has been data, people have needed ways to sort, collate, and otherwise organize it so that it can be retrieved quickly. In the ancient past, this might have been writing down numbers on clay tablets, or later, on parchment. Once there was enough to fit on one page, you needed a way to organize that information so you knew where to look for what you needed.

[Painting by Jan Gossaert, *The Prosecutor's Study*]

The history of filing is fascinating in its own right. Related documents used to be placed along a string in the order in which they were added, making it easy to flip through and find the piece of parchment that you were looking for. The English verb *to file* even comes from the root latin *filum* (as in filament), which means string or thread. And while these strategies worked for a long time, eventually much more complex systems of organization became necessary.

Our current data challenges are best represented by the evolution of maps. At the dawn of cartography, complex maps took years to create. This meant that, once made, they would remain unchanged for many decades. Even in this "read-only" state, however, they were still incredibly valuable.

Maps also weren't very versatile. Primitive maps (and even some more modern ones) were often useful only for a single purpose. Even as late as the 20th century, a road map was not interchangeable with a terrain relief map or a map showing seismic fault lines. A map of the whole world wasn't terribly useful when you wanted to figure out how to get to the next town over.

Bottom line: When you wanted to update these maps, or perhaps needed a map for a different purpose, you'd have to make an entirely new one. But today, maps are *dynamic.* You can pull out your smartphone and instantly get the most up-to-date map of your immediate surroundings. You can view different layers of information, change the scale on which you're viewing, and get real-time information on traffic or weather. To the map makers of the previous millennium, this would seem like magic.

With these modern digital maps, we don't just build a new map every time something changes, we make *updates*. And in many systems, we make thousands of these updates every second.

How Data Systems Have Evolved to Seek Data Faster

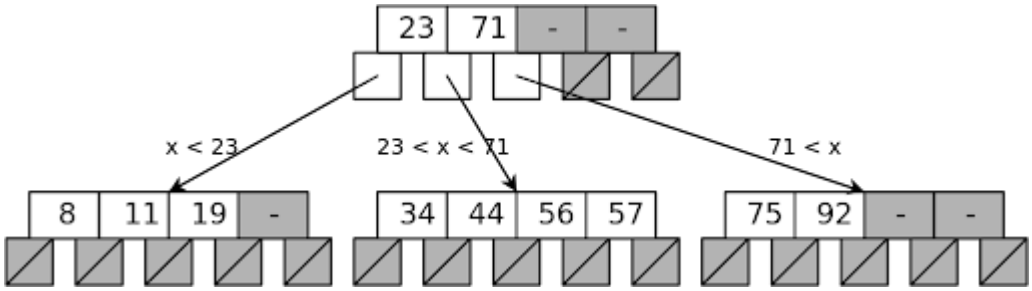
It was these kinds of problems that, in the early part of this century with "Web 2.0," led to a revolution in

how we stored information.

Prior to this major change, data was more often *read* than *written.* Most systems were biased in favor of read performance. As the hard-disk drive (HDD) had been the dominant storage medium since the 1970s, and disks had a much harder time seeking to where metadata was stored rather than just reading and writing it, many optimizations in data structures were made on how to work around this fundamental limitation of disks. The fewer seeks you did made overall performance significantly better—and sequential reads or writes were relatively painless by comparison.

One of these early optimizations was the B-tree, which was outlined in a paper first published in 1970. It's an idea so obvious—even self-evident, in hindsight—that it's often said it was *discovered* rather than invented.

What Is the B-tree?



[Source: https://commons.wikimedia.org/wiki/File:B-tree_example.svg]

The B-tree is "a self-balancing tree data structure that keeps metadata sorted and allows searches, sequential access, insertions, and deletions in logarithmic time." The key idea behind the B-tree is that it lets you store indices of keys with associated values and that you could minimize the number of seek operations when searching for that key's value. The tree would be made of nodes, and these nodes would include pointers not just to information but to other nodes. These keys could be anything sortable, and the values could be pointers or other reference information. By walking down the tree, you could quickly arrive at the index key you were searching for.

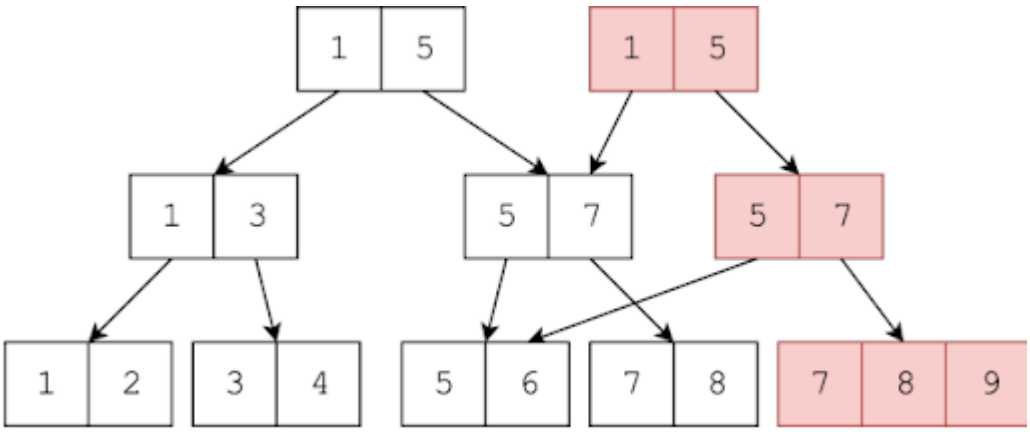
B-trees had tremendous advantages for many different use cases. You could use them for keeping indexes of file locations in a file system or for storing values in a database. And the structure was extremely flexible: You could have an arbitrarily tall tree to improve intra-node searches, or you could have a shorter tree with wider nodes to minimize the number of seeks at the expense of longer intra-node search times or an increased need for memory. This meant one could tune the tree structure for different needs.

Over the years, many other optimizations to this basic structure were made, such as **caching** frequently accessed nodes in memory or generating an index *of* the index to speed up the initial search. Fundamentally, however, the data structure has remained relatively unchanged and yet still very relevant for decades.

Challenges and Innovations of the B-tree

But B-trees have some drawbacks. The first is the concept that nodes within a B-tree are *immutable*. They cannot be changed, and so any time you're doing an update (rather than just appending new information), you need to completely rewrite the node and change the pointers from its parent node accordingly. This is

called *copy-on-write*. It's important in a storage context because ideally, you never want to overwrite existing data until you know your new data is fully written and safe. This is called *atomicity* in the database world—you want your operation to fully complete or not at all.



The second challenge is that over time, your tree can become lopsided and imbalanced. As a result, you'll hit a threshold where you'll need to rewrite large portions of your tree in order to rebalance it or the tree will—metaphorically—"fall over" and cause much longer read times for some data than others. Both of these activities involve re-writing data that is already written. In a storage medium where write endurance is effectively infinite, these trade-offs make perfect sense. (Keep that in mind as we go forward.)

One of the late-stage innovations in the B-tree was **buffering bursts of writes** into a time-ordered log. Rather than doing an insert into the tree every time new information was entered, you could buffer many of these write operations together and then perform these operations in a batch. This reduced the number of tree-rebalance and node-copy actions significantly, and it's one of the major innovations that came to file systems in the early 2000s. This helped a lot with short bursts of writes, as it would amortize the cost of tree restructuring out over time to help smooth out those perceived bumps. However, it didn't help with workloads that were extremely update-intensive. While the dam could help keep the waters calm downstream, it would eventually fail if it was overrun by the river's flow.

A New Kind of Tree Emerges

Also in the early 2000s, the internet was gaining momentum and having major impacts on society and how we stored and accessed information. Many of the new companies that were at the forefront of this change (so-called "Web 2.0," including Yahoo, Google, and Facebook) began to deal with a huge influx of data. Suddenly, updates were more and more frequent.

Existing data systems had to be rethought. In academia, this had been a subject of much research, and a new type of data structure, first proposed in 1991, and later dubbed the Log-Structured Merge-Tree (LSM-tree), was designed to handle high-insert data streams. Outside of this academic setting, it had not been utilized at scale in most systems. That was until these future internet giants started to implement them in service of the ever-increasing flow of data they were ingesting. The most famous early example of this was Google's Bigtable.

In order to understand how the LSM-tree works, let's perform a thought experiment.

The best system for doing lots of writes would be a system where new writes are *always* appended. Nothing is ever deleted, inserted, or overwritten. Instead, new data is always written at the end of an ever-running log. **Think of this like a bank ledger or a visitor log at an office building.** Deletions or updates are just new entries in the log, which invalidate the prior entry. This is by far the most efficient way to write—but it has two major problems:

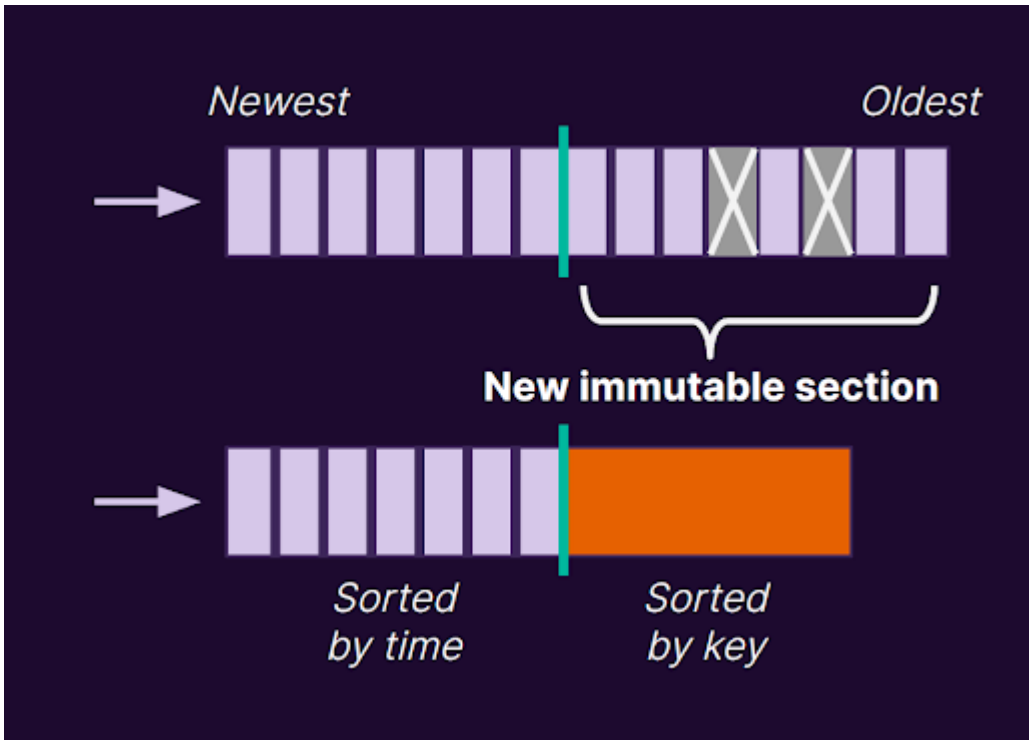
- The capacity required is always growing (because nothing is ever deleted).
- Reads can be catastrophically bad because you have to read backwards from the end of the log until you find the data you're looking for.

If the metadata was written a long time ago, you might end up reading through almost the entire log.

Clearly, we need to make some improvements. A system that uses infinite space and has terrible read performance isn't a good trade-off for write performance. The first improvement we can make is to implement ***garbage collection***, where we periodically go through and delete bits of data that have been overwritten or invalidated. This should sound familiar if you've read Volume 1. This solves our infinite space problem. As data is overwritten or deleted, we can actually reclaim space, but at the expense of a background process that periodically performs these clean-up duties.

We still haven't solved our read performance problem, though, because we still have a temporally ordered but otherwise unsorted ledger of updates.

The solution to this problem is to periodically take the oldest data and perform a **merge and sort operation**, taking all this unsorted data from a period of time or an *interval* and reorganizing it into a sorted list and writing it to a new immutable segment. This process essentially defers the insertion/sorting process to later, doing it in batched operations similar to how the write buffer in front of a B-tree could defer the tree rebalancing penalty for later. But instead of doing this only in the beginning before inserting it into the B-tree, we're doing this all the time.



The effect of this is that as data ages and the tree gets "flattened," it gets organized further and further down the tree. Newer, hot data will be in more recently written segments higher in the tree, making it faster to find as you descend. We also avoid rewriting cold data as much as possible. Because we only rewrite data when these merge sort operations occur, we have a lot of control over how much data gets written and when.

Throw in optimizations like caching of frequently accessed keys, fence pointers (an index of the key ranges of each layer's segments), and you've got a system that provides an excellent balance of read, write, and update performance, where you can tune various parameters of your tree to prioritize each of those operations differently. This is one reason why the LSM-tree has become **the de facto standard for modern data storage platforms**.

Where Trees and Flash Meet

When LSM-trees had their big moment in the early 2000s, there was another major change just starting to germinate in the technology world, and that was the transition from disk to flash. In the mid-2000s, few people thought that flash would eventually obsolete hard disks (although there are some famous exceptions), but flash had begun displacing disk in more and more places in the consumer electronics industry and anywhere that space and power were at a premium.

But NAND flash had a major challenge: finite endurance. As we discussed extensively in Volume 1, this problem was only going to get worse as flash became more and more dense. This presents a major conundrum:

Modern storage systems need to handle many writes and frequent updates, but flash has a finite number of write cycles.

Thus, using data structures (like the B-tree), which dramatically increases write amplification (due to its copy-on-write and tree-balancing activities), exacerbates this problem. We're taking a system designed around avoiding disk seeks and applying it to a storage medium that has little to no seek time, while

simultaneously accelerating the wear of that medium; it's just no longer fit for purpose. It is, in fact, the **exact opposite** of what you'd want your storage system to do.

Therefore, storage solution design should have a new north star: **reduce write amplification.** Unnecessary writes are the new disk seeks—something to be avoided whenever possible. The LSM-tree allows us to build a system that is organized around this principle.

Pure Storage did not invent the LSM-tree, but we were one of the first companies to make a few key insights:

- LSM-trees are uniquely suited to NAND flash media.
- A system utilizing LSM-trees on top of NAND flash could be tuned for many different varieties of flash and many different types of workloads.
- A key-value store database implemented this way would be incredibly scalable and could be used as a foundation for any storage protocol, in both scale-up and scale-out architectures.

Based on these insights, and on our deep understanding of NAND flash, we've been incrementally improving our LSM-tree implementation for the last 12 years, and it's what we internally call…

The Pyramid

Purity's Approach to Log-structured Mapping

To explain what makes the Pyramid so special, let's break down the three points above.

1. The Pyramid is designed from the beginning to be aligned to the particular characteristics of NAND flash.

As we discussed in Volume 1, flash is fundamentally different from spinning disk: It has no seek time and it can be highly parallel, but it has a particular endurance which means you want to **avoid unnecessary writes**.

By deeply integrating the Pyramid with the underlying storage medium, we can eliminate lots of redundant work and leverage much of the synergies between an LSM-tree and the way that flash works. LSM-trees require garbage collection of older data; so does flash. What if we collected garbage *once*, collapsing the garbage collection of both the data structure and the media into one step?

Implementing a B-tree on top of flash would create write amplification every time you do a copy-on-write or you rebalance the tree. By contrast, if we buffer the stream of writes and coalesce them in batches by keeping the buffer in NVRAM and the lower layers in flash, we only write sorted and merged data down to flash. The only time data gets rewritten is when a layer of the Pyramid gets flattened, and this happens less and less as these immutable segments age.

When we talk about how Pure's products are designed for flash and don't have any legacy HDD code, this is one of the key things we're talking about. Unlike other solutions which may have been designed in an era when drive seeks were the enemy and built their fundamental data structures around those realities, Pure's solution is built at its most basic level around the particular characteristics of flash.

2. The Pyramid's design is flexible and it allows us to use one single architectural family to satisfy the needs of different performance and capacity designs.

The same fundamental data structure has scaled from arrays with tens of terabytes to arrays with many petabytes, and across many generations of NAND flash and controller architectures. This flexibility allows us to tune the parameters of the Pyramid as we refine our code or modify our strategies and trade-offs when using different NAND types or designing our system for different performance or latency targets. We even leverage different Pyramid configurations within the same product for different purposes.

3. The Pyramid is extensible, which has made it last across numerous system generations, even as we've added new features and redesigned portions of it.

It's allowed us to metaphorically change the tires on a car while it's still on the road and is the key enabling technology for what we call Evergreen. There are Pure arrays out there right now with segments that were written *years* ago in prior versions of Purity that have been unchanged through controller upgrades and storage expansions.

This is why Evergreen is more than just a buying program: Evergreen is only possible when the underlying technology enables it. Building a brand new product that is fundamentally incompatible with the old product and offering migration services and upgrade discounts is *not Evergreen.* Evergreen is not something that we cooked up in marketing, but it's something that is **core to the design of our products**, going back more than a decade. It's not something that you can tack on later—it has to be something you build in from the start, and when it was founded, Pure Storage had the vision and foresight to make it a reality.

Today, we have customers with arrays that have extended the same Evergreen™ subscription for **10 years**. Ninety-seven percent of Pure arrays purchased six years ago with Evergreen are still in service today. How many storage migrations have been avoided in that time for our customers? How many perfectly good components have avoided the e-waste bin thanks to Evergreen?

The Pyramid Is Better Science

The Pyramid is an incredible enabler for Pure Storage's portfolio of products, and it's a great example of how approaching problems in new, innovative ways can have game-changing and long-lasting impacts. At Pure, we're all about *Better Science:* doing things the right way, not the easy way. The easy way to introduce flash is to take what you've been doing for a couple decades in an existing system with hard drives and just swap out the spinning drives for solid-state ones. But the right way is designing a system that isn't built on old assumptions and constraints and reimagines what's possible with the new technology at hand.

It's the difference between ripping the internal combustion power train out of a car and shoving batteries and an electric motor into it, versus building an electric car platform from scratch. In the long run, it's pretty clear what the better approach is.

Listen to the Pure Report podcast to learn more!

Post Likes 15

Color orange–gradient

Color orange–gradient

Color orange–gradient

Color orange–gradient

Color orange–gradient