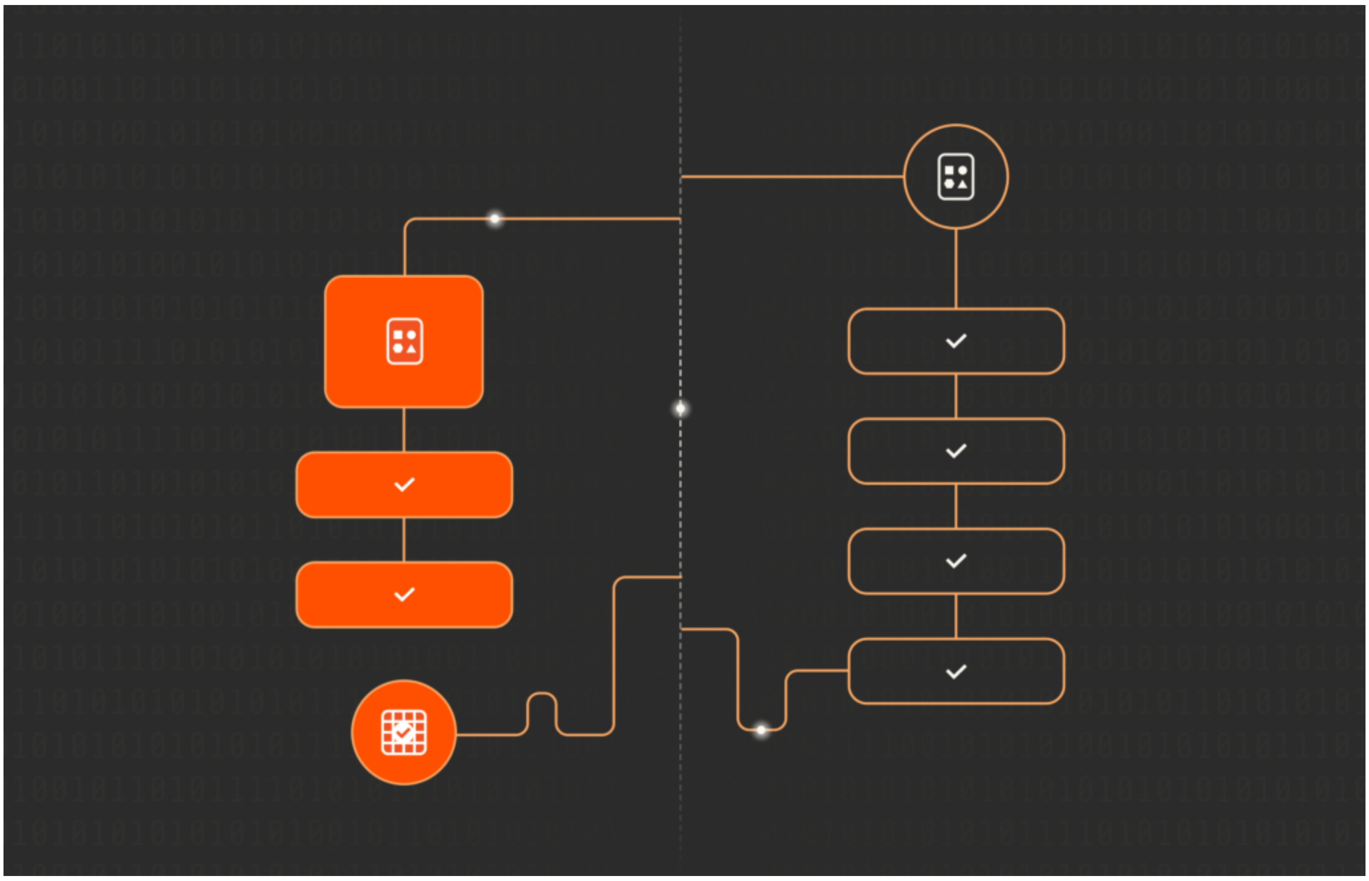
PURESTORAGE®

# An Exploration of Files and Objects for Data Storage



File and object data sets continue to expand in the data center. As a result, a number of trends pose new challenges to the storage infrastructure:

- Growth of machine-generated data
- Continued adoption of AI and machine learning
- Emergence of object storage as primary storage
- Tightening of backup recovery SLAs driven by ransomware attacks

These trends lead to workloads with demanding performance characteristics, often requiring small files, random access, and large numbers of files in one namespace. Also, these workloads increasingly utilize files, as well as objects, with high-performance expectations on both.

Since files and objects are both important, widely used forms of data representation, supporting both in one system is an attractive goal. At the same time, there are deep differences between files and objects, so supporting both in a single system presents an interesting challenge.

In this three-part blog series, we'll focus on one key design decision:

*How should a single unified design represent files, directories, objects, and buckets?*

This first post will cover the background knowledge for the subsequent discussion of design options.

**Protocol Considerations**

To make the example more concrete, we'd look to support the following protocols:

| | |
|---|---|
| **File:** | NFS, SMB, and/or POSIX |
| **Object:** | S3, Azure blob, and/or GCS blob |

**While this may be useful context for some readers, you don't need to know the details of these protocols to follow along.**

## Files and Directories, Objects and Buckets

Before diving into the design options, there's some conceptual background knowledge worth noting. To start, what are files, directories, buckets, and objects?

Files and objects solve a similar problem—providing a way to organize and address data. The two models approach the same problem in fundamentally different ways.

## Files and Directories

If you’ve used a computer, you’ve likely used files and directories. They look something like this:

```
/
├── Pictures
|   └── Spring-2019
|       ├── 0001.JPG
|       └── 0002.JPG
└── Videos
    └── Spring-2019
        └── 0001.MP4
```

A directory is an unordered container that holds files and subdirectories. The result is a hierarchical system of organizing files, rooted in a single top-level directory. A directory is typically not that big: 100s to 100,000s of entries. Some systems, such as Pure Storage® FlashBlade®, support really big directories. But even if you have a file system that works fine with large directories, you still typically don’t want to put millions or billions of entries into a single directory. Ultimately, applications, tools, and libraries tend to assume that directories are moderately sized. For example, the standard “ls” tool will typically load all entries into memory and sort them before showing any output.

So, think of directories as hierarchical, unordered, and moderately sized.

## Objects and Buckets

Like a file, an object is a named blob of data. However, unlike a file, an object is typically immutable, meaning that it can’t be modified once written.

A bucket is a flat collection of objects, usually many of them—think billions. Unlike a directory, a bucket can’t contain other buckets.

While buckets can’t nest, you can approximate a hierarchical structure through a naming convention. For example, I can put these objects in one bucket:

Pictures:Spring-2020:0001.JPG
Pictures:Spring-2020:0002.JPG
Videos:Spring-2020:0001.MP4

To help with this, an object store exposes a list operation to enumerate the objects in a bucket. The list operation supports prefix-based filtering: For example, I can list objects with the prefix Pictures:Spring-2020: and get back a list of pictures from Spring-2020.

Three more things to note:

- **There's no special delimiter.** I used a colon (:) in this example, but that's arbitrary. The user can pick a convention using any delimiter, or choose not to use any nesting convention at all. Also, the prefix doesn't need to end with a delimiter: I could list all objects that start with Pictures:Spr and the object store needs to answer. Using the forward slash (/) as the delimiter is the convention that makes a bucket look the most like a file system, at least at first glance.
- **Buckets are ordered.** While I don't know the internals of every object store out there, efficient support of the list operation basically requires something like an ordered index on the bucket.
- **Buckets get big.** Due to the lack of hierarchy, a bucket can get massive. For example, in a typical file system with 1 billion files, each individual directory will still often be relatively small. In an object store, if you have 1 billion objects in a bucket, they're all in the same flat namespace.

So, think of buckets as flat, ordered, and often massive.

## Transactionality

One underappreciated property of file systems is just how transactional they are. For example, let's say that you delete a file:

**~$ ls -ld Pictures/Spring-2019/**

```
drwxrwxr-x 2 igor igor 4096 Jun 15 01:21 Pictures/Spring-2019/
~$ rm Pictures/Spring-2019/0002.JPG
~$ ls -ld Pictures/Spring-2019/
drwxrwxr-x 2 igor igor 4096 Jun 15 01:25 Pictures/Spring-2019/
```

Notice that the modification time of the directory changed, as shown in red. Deleting the file modified the directory, and so the modification time must change as well. Clients and applications expect the entire transaction to have ACID semantics, like in a transactional database, or else various caches stop working correctly.

**EXAMPLE**

When you delete a file, the file system must transactionally persist multiple changes. In a typical file system, deleting a file results in these modifications:

- **Directory inode:** Modification time is set to current time.
- **Directory entry table:** Entry is removed.
- **File inode:** Link count is decreased by 1. If it reaches zero—there are no other hard links to the file—inode is freed.
- **File data blocks:** The actual file blocks are freed, if link count 0 was reached.

This is a simplified version and the real transaction may be substantially more complicated.

In a file system, if you delete files /Pictures/0001.JPG and /Pictures/0002.JPG, the two operations contend on the same directory /Pictures. They must both correctly and transactionally modify the directory, including the modification time and other attributes, with ACID semantics.

Object stores don't have these strong transactional requirements. If you delete objects Pictures:0001.JPG and Pictures:0002.JPG, the operations are entirely independent. There's no common directory that needs to be carefully managed because Pictures is just a naming convention. No entity named Pictures actually exists.

The transactional requirements of file systems are quite constraining, especially in distributed environments. This is a big part of why public cloud providers chose object over file as the primary storage model.

**EXAMPLE**

File systems are transactional in nature. The file system must ensure that the user won't be able to create a cycle by moving a directory into its own subdirectory. For distributed systems, this isn't necessarily trivial because the entire directory graph isn't known to any one node. So, in the presence of potentially many concurrent modifying operations, the file system must ensure that a user won't manage to create a cycle of directories. These types of problems simply don't exist in an object store.

## Snapshots vs. Versions

Since the file and object storage models are fundamentally quite different, features that have been built in the two worlds also differ.

For example, a common way to protect a file system against accidental data destruction is by taking point-in-time consistent snapshots of the entire file system or a top-level directory. Since a file system has an interdependent hierarchical structure, you typically want to be able to recover back to a consistent point in time. Due to operations like a directory move, an inconsistent view of a file system could have a cycle or unreachable clusters of directories.

In object stores, the individual objects are completely independent. A common way to protect objects is through object versions—by keeping old versions of replaced or deleted objects.

Furthermore, as you get into advanced features for file and object use cases, there's further divergence between the file and object worlds. For example, the expectations around replication and geo-distribution are quite different in each.

## Let’s Summarize

The goal is to design a system that can store two types of entities with different properties:

| | Directory | Bucket |
|---|---|---|
| **Organization** | Hierarchical | Flat |
| **Ordering** | Unordered | Ordered |
| **Entry Count** | Moderate | Massive |
| **Modifications** | Transactional | Independent |
| **Protection** | Snapshots | Versions |

So, while the file and object worlds have some similarities, there are plenty of deep differences. And our goal is to design a system that can handle both! *Read part two of the series:* ***Designing a System to Store Both Files and Objects***

## Expanding Snapshot Monitoring with Modern Pure Storage Innovations

As organizations continue to prioritize efficient storage management, Pure Storage has evolved its product offerings to meet modern data demands. Building on the principles outlined in monitoring snapshot space consumption, let’s explore some of the latest advancements in Pure Storage’s portfolio.

### Unified Block and File Storage with FlashArray//C

The Pure Storage FlashArray//C, designed for cost-efficient capacity-optimized workloads, offers enhanced capabilities that align perfectly with snapshot-based data management. With unified block and file storage, users can consolidate multiple workloads on a single platform while maintaining predictable performance. The integration of file storage capabilities extends snapshot management to file-based workflows, enabling seamless space consumption monitoring across different data types.

Some key features of FlashArray//C include:

- **Cost-optimized Storage**: Built for secondary workloads, FlashArray//C provides high-density storage at a lower total cost.
- **Data Reduction**: Industry-leading data reduction technologies ensure minimal snapshot space usage.
- **Always-On Efficiency**: Inline deduplication and compression work together to optimize storage space, making snapshot retention cost-effective.

By leveraging FlashArray//C, organizations can simplify storage management and enhance their snapshot monitoring strategies, particularly for hybrid or secondary storage environments.

## Comprehensive Product Suite for End-to-End Data Management

Pure Storage's product lineup offers a robust foundation for addressing diverse storage needs, from mission-critical workloads to scalable, cost-effective solutions. The portfolio includes:

- **FlashArray//X**: High-performance all-flash arrays designed for latency-sensitive applications.
- **FlashArray//C**: Optimized for capacity and efficiency, as described above.
- **FlashBlade®**: Purpose-built for unstructured data workloads, providing scalability and performance.
- **Pure1®**: A cloud-based management and monitoring platform that extends visibility into snapshot space consumption, enabling proactive management.

These offerings underscore Pure Storage's commitment to simplifying storage operations while delivering performance and efficiency. By integrating these technologies with your existing monitoring practices, you can gain deeper insights into storage usage and make data-driven decisions to optimize your environment.

## Innovations with FlashBlade//S

FlashBlade//S is the next evolution in unstructured data storage, designed to address the ever-growing demands of modern data workloads. This powerful platform provides unmatched scalability and performance for both file and object storage, making it an essential component of any organization's data strategy.

Key features of FlashBlade//S include:

- **Scalable Architecture**: Delivers modular scalability, allowing you to expand capacity and performance independently as your data grows.
- **High Performance**: Optimized for high-throughput and low-latency workloads, including AI/ML training, data analytics, and backup and recovery.
- **Unified File and Object Storage**: Seamlessly handles both file and object data with a single system, reducing complexity and operational overhead.
- **Sustainability**: Engineered with energy efficiency in mind, FlashBlade//S reduces power and cooling requirements while delivering unparalleled performance.

Organizations leveraging FlashBlade//S can take their unstructured data strategies to new heights, ensuring seamless scalability and efficiency while maintaining simplicity in operations.

## Conclusion

Pure's technology enables better monitoring and management of snapshot space consumption across diverse workloads. As you adopt new storage solutions like FlashArray//C or explore unified block and file storage, ensure that your monitoring practices are updated to reflect these innovations. With tools like Pure1®, you're equipped to handle the demands of modern data management with confidence.