MINIO

Erasure Coding Primer

WHITE PAPER

Erasure coding is a key data protection method for distributed storage systems. This paper is divided into three sections. The first section explains how erasure coding aligns with enterprise requirements for data protection. The second section describes the implementation of erasure coding in MinIO. The third and final section focuses on the advantages of using erasure code to protect data in cloud-native environments.

Erasure Coding Explained

Data protection is essential in any enterprise environment because hardware failure is common. Drive failure, including SSD or HDD, is one of the most common hardware failures to occur. Backblaze publishes hard drive failure rates on an annual basis, and in 2021 (the most recent data) the company had 1.01% of HDD fail. In another study, the storage provider estimated that the annualized failure rate between April 2013 and June 2021 for SSD was 1.05% and for HDD was 6.41%. The value of data protection is underscored by results of a survey of 3000 IT decision makers by Veeam published in 2022 that estimated downtime costs $1,467 per minute.

In a storage system where data is stored on a large number of drives, there will be failures, and the chances of a single drive failing increases with the number of drives in use. Storage systems typically seek to offset this risk by redundantly storing data such that when a number of drives fail, data can still be accessed from other disks.

Failure for enterprise and web-scale object storage takes many forms. Failure isn't a corrupt file or a corrupt drive or even a failed transaction - it is employees, partners and customers getting 404 or 503 errors while multiple applications and the lines of business they support are put at risk. The business implications of offline or corrupted storage in a real time 24/7/365 world are tremendous, so much so that they have spawned an entire range of data protection technologies.

Traditionally, different types of RAID technologies or mirroring/replication were used to provide hardware fault tolerance. Mirroring and replication rely on one or more complete redundant copies of data - this is a costly way to consume storage. More complex technologies such as RAID5 and RAID6 provide the same fault tolerance while reducing storage overhead. RAID is a good solution for data protection on a single node, but fails to scale due to time consuming rebuild operations required to bring failed drives back online.

Erasure coding is the modern approach to data protection because it is resilient and efficient. It provides data protection by splitting data files into data and parity blocks and encoding it so that the primary data is recoverable even if part of the encoded data is not available. As each parity fragment is written, the erasure coding algorithm calculates the parity's value based on the original data fragments. Horizontally scalable distributed storage systems rely on erasure coding to provide data protection by saving encoded data across multiple drives and nodes. If a drive or node fails or data becomes corrupted, the original data can be reconstructed from the blocks saved on other drives and nodes.

In 1960, I. Reed and G. Solomon developed the "block code" coding technique called Reed-Solomon coding. Today, Reed-Solomon codes remain popular due to standards compliance and efficient implementations across many hardware and software formats. MinIO uses the Reed-Solomon algorithm for erasure coding.

Erasure coding protects data saved to distributed systems, making it possible to access data even if part of the encoded data is unavailable. In a distributed system there are multiple independent underlying storage systems and erasure coding is used to encode and distribute the data across the independent storage systems (JBOD) so that the data is recoverable in the event of hardware failure on some of the drives.

Mirroring, replication and RAID are not efficient uses of storage capacity, plus they incur additional bandwidth due to overhead - and inefficiencies are magnified as the storage environment grows. For example, with N-way replication, N replicas are stored on N different drives, and the system is able to tolerate at most N-1 drive failures. Moreover, N-way replication can only achieve a storage efficiency of 1/N (at its theoretical best case, not accounting for operational overhead).

Many distributed systems use 3-way replication for data protection, where the original data is written in full to 3 different drives and any one drive is able to repair or access the original data. Not only is replication inefficient in terms of storage utilization, it is also operationally inefficient when it recovers from failure. When a drive fails, the system will place itself in read-only mode at reduced performance while it fully copies an intact drive onto a new drive to replace the failed drive.

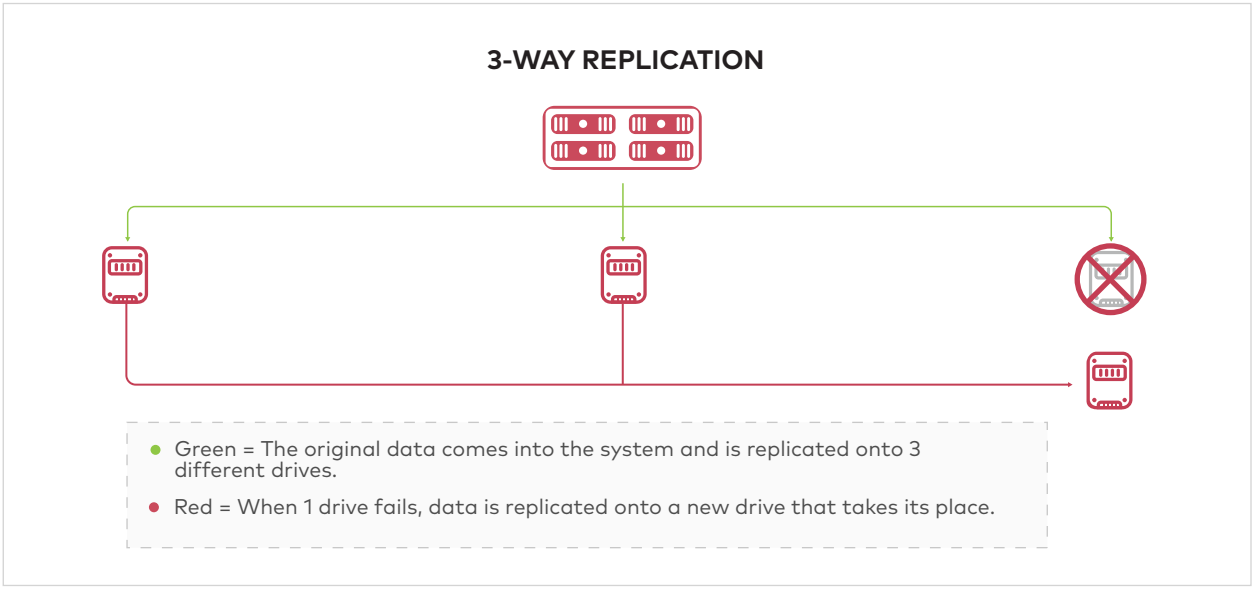

Figure 1. Three Way Replication

Erasure coding, in contrast, is able to tolerate the same number of drive failures with much better efficiency, by striping data across nodes and drives. There are many different erasure coding algorithms, and Maximum Distance Separable (MDS) codes such as Reed-Solomon achieve the greatest storage efficiency.

In object storage, the unit of data to be protected is an object. An object can be stored onto n drives. In erasure coding, we use a number k to indicate potential failure. Therefore, k < n, and with MDS codes the system can guarantee to tolerate n - k drive failures, meaning that k drives are sufficient to access any object.

Considering an object that is M bytes in size, the size of each coded object is M / k (ignoring the size of metadata). Compared to the N-way replication shown above, with erasure coding configured for n = 5 and k = 3, a distributed storage system could tolerate the loss of 2 drives, while improving storage efficiency by 80%. For example, for 10 PB of data replication would require more than 30 PB of storage, whereas object storage would require 15-20 PB to securely store and protect the same data using erasure coding. Erasure coding can be configured for different ratios of data to parity blocks, resulting in a range of storage efficiency. MinIO maintains a helpful erasure code calculator to help determine requirements in your environment.

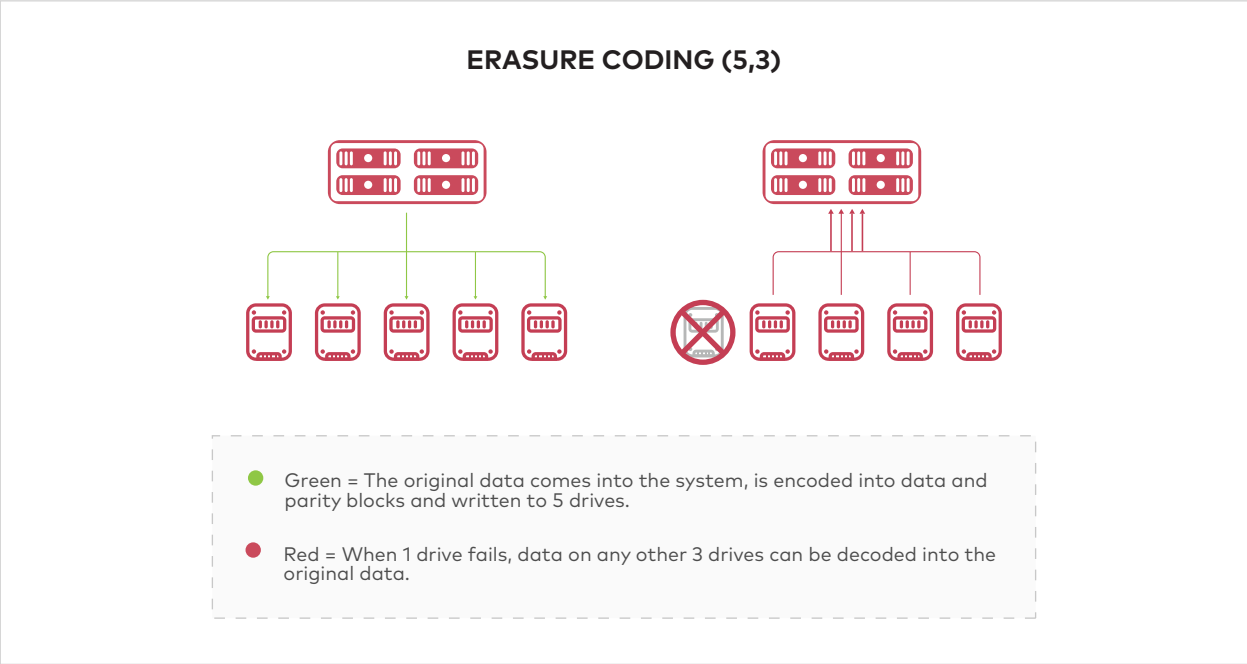

Figure 2. Erasure coding stripes data and parity across drives.

In such a system, accessing the data object requires the system to access k different coded blocks (saved on k different drives) to read the original object using the decoding algorithm of the MDS code that encoded the object. The data object is made up of coded data and parity blocks. In the event of drive corruption or failure, all the system needs is a very small piece of the data object in order to repair it.

In the past decade, the design objectives of erasure codes for cloud storage have matured from a focus on data integrity to a focus on resource overhead. Erasure coding algorithms are typically evaluated based on multiple repair-related factors such as the amount of bandwidth required for a repair, the amount of I/O required for repair, access latency of erasure coded data compared to access latency of the original data, and storage efficiency - with the last being of critical importance when designing web-scale distributed object storage systems.

Overview of MinIO Erasure Coding

MinIO protects data with per-object, inline erasure coding (official MinIO documentation for reference) which is written in assembly code to deliver the highest performance possible. MinIO makes use of Intel AVX512 instructions to fully leverage host CPU resources across multiple nodes for fast erasure coding. A standard CPU, fast NVMe drives and a 100 Gbps network support writing erasure coded objects at near wire speed.

MinIO uses Reed-Solomon code to stripe objects into data and parity blocks that can be configured to any desired redundancy level. This means that in a 16 drive setup with 8 parity configuration, an object is striped across as 8 data and 8 parity blocks. Even if you lose as many as 7 ((n/2)–1) drives, be it parity or data, you can still reconstruct the data reliably from the remaining drives. MinIO's implementation ensures that objects can be read or new objects written even if multiple devices are lost or unavailable.

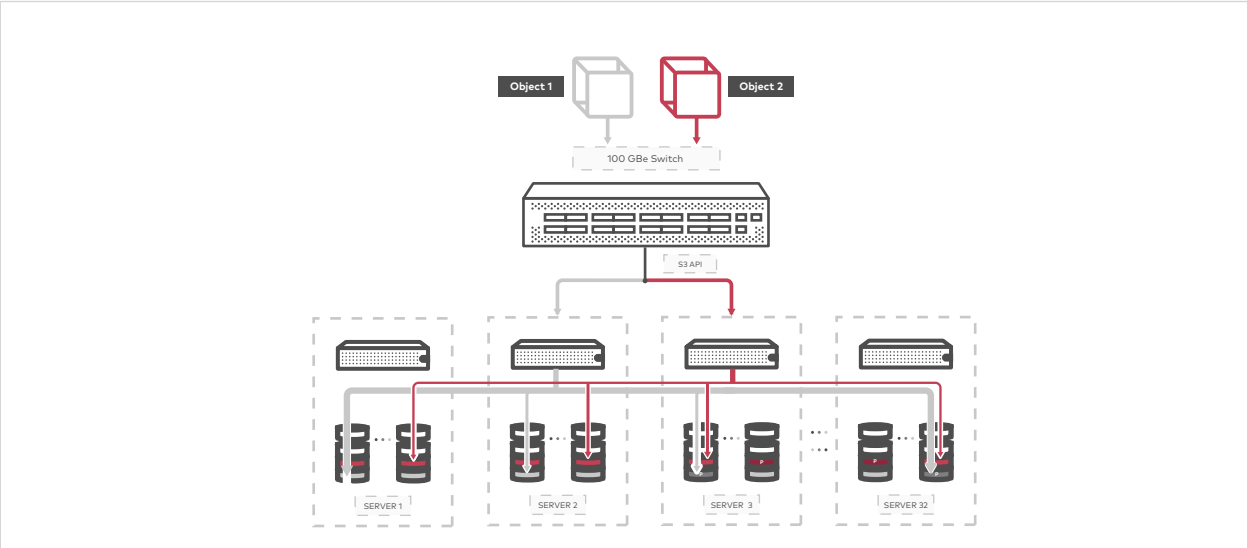

Figure 3. Erasure code protects data without the overhead associated with alternative approaches.

MinIO uses a Reed-Solomon algorithm to split objects into data and parity blocks based on the size of the erasure set, then randomly and uniformly distributes the data and parity blocks across the drives in a set such that each drive contains no more than one block per object. While a drive may contain both data and parity blocks for multiple objects, a single object has no more than one block per drive, as long as there is a sufficient number of drives in the system. For versioned objects, MinIO selects the same drives for data and parity storage while maintaining zero overlap on any one drive.

We use the EC:N notation to refer to the number of parity blocks (N) in an erasure set. The number of parity blocks controls the degree of data redundancy. Higher levels of parity allow for greater fault tolerance at the cost of total available storage. The default for 16 drives is EC:4 which results in 12 data blocks and 4 parity blocks per object. We chose the default stripe size to balance protection and performance.

Total Drives (n)	Data Drives (d)	Parity Drives (p)	Storage Usage Ratio
16	8	8	2.00
16	9	7	1.79
16	10	6	1.60
16	11	5	1.45
16	12	4	1.34
16	13	3	1.23
16	14	2	1.14

Figure 4. MinIO Erasure Coding: Configurable Data and Parity Options and Their Storage Usage Ratios

The back end layout of MinIO is actually quite simple. Every object that comes in is assigned an erasure set. An erasure set is basically a set of drives, and a cluster consists of one or more erasure sets, determined by the amount of total disks.

Let's take a look at a simple example to understand the format and the layout used in MinIO.

It's important to note that the format is about the ratio of the data drives to parity drives - whether we have four nodes with a single drive each or four nodes with 100 drives each (MinIO is frequently deployed in dense JBOD configurations).

We can configure our four nodes with 100 drives each to use an erasure set size of 16, the default. MinIO will group drives into groups of 16 and make them one erasure set. This is the logical layout, and it is part of the definitions of the erasure coding calculations. Every 16 drives is one erasure set made up of 8 data and 8 parity drives. In this case, the erasure set

is based on 400 physical drives, divided equally into data and parity drives, and can tolerate the loss of up to 175 drives. When an object and its parity are written, they are written across the 16 drives by the erasure code algorithm. The erasure code algorithm that was used for each object is recorded along with the content itself to simplify decoding on read.

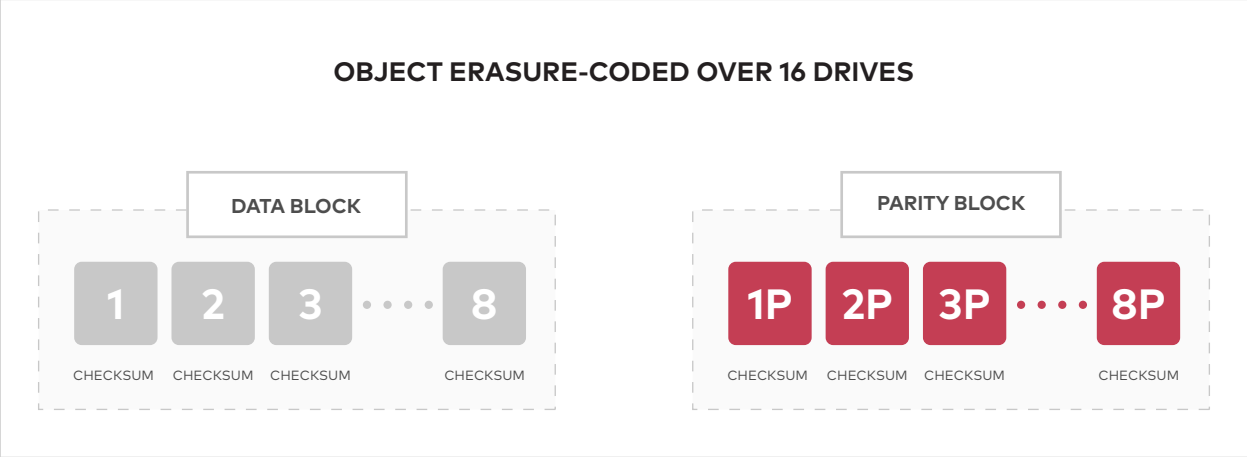

Figure 5. Using MinIO default erasure coding configuration provides the ability to tolerate the loss of up to half of the drives without downtime or data loss.

Once a set of nodes is given an erasure coding configuration, everything written to MinIO adheres to the same configuration. There's no protocol layering and translation to increase complexity and add processing time. To drive the point home, if there were a fire in the data center, an administrator could pull all the drives out, carefully mount them in other servers and read the data. The data is all encrypted and can't be read without the keys or tampered with. MinIO's design is simple, secure and powerful. From a supportability operations point of view, admins will always be able to make sense of which data is held on which drive.

MinIO's XL metadata, written atomically with the object, contains all the information related to that object. There is no other metadata within MinIO. The implications are dramatic - everything is self-contained with the object, keeping it all simple and self-describing. XL metadata indicates the erasure code algorithm, for example two data with two parity, the block size, the checksum. Having the checksum written along with the data itself allows MinIO to be memory optimized while supporting streaming data, providing a clear advantage over systems that hold streaming data in memory, then write it to disk and finally generate a CRC-32 checksum.

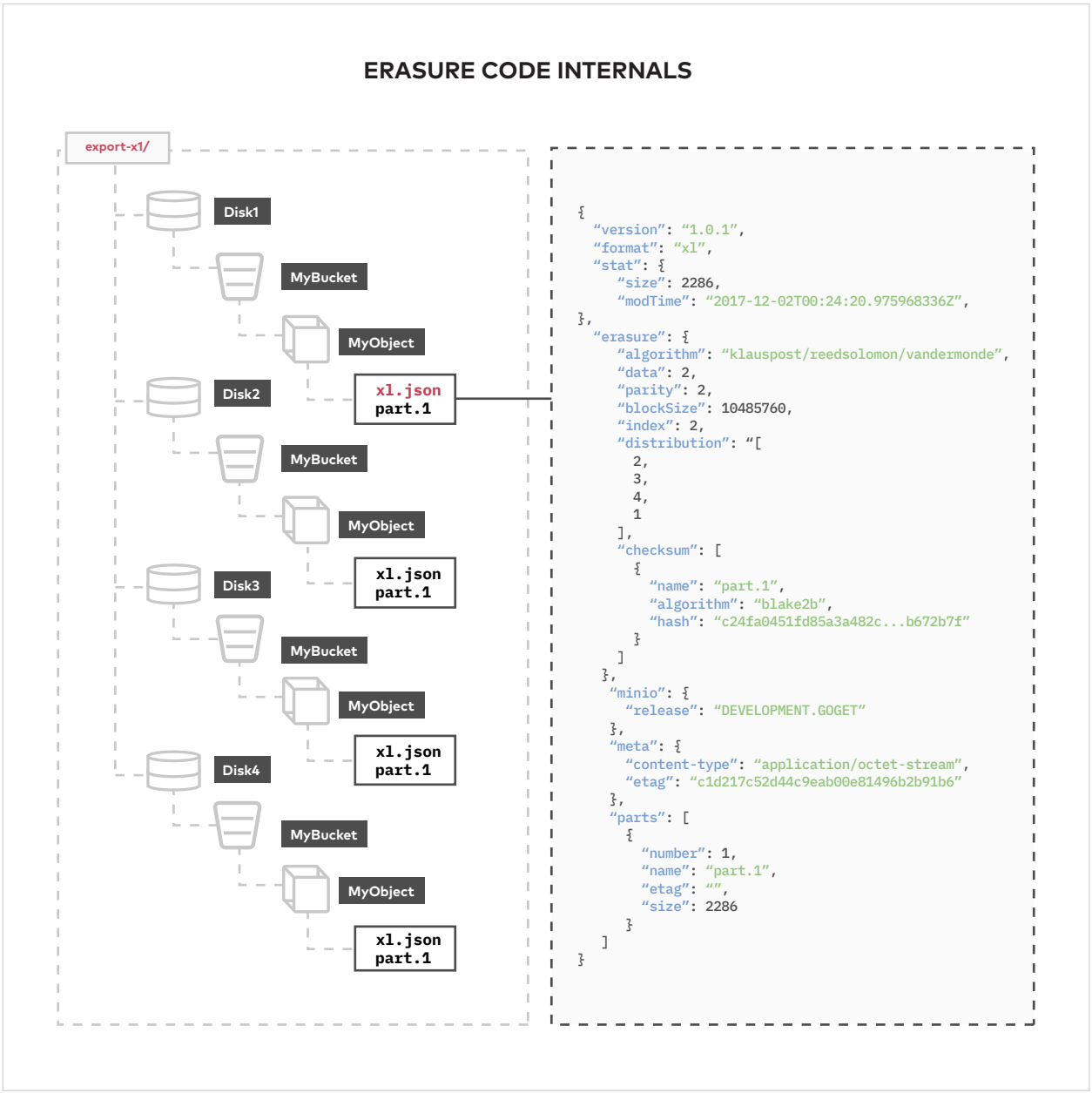

Figure 6. Erasure coded objects are striped across drives as parity and data blocks with self-describing XL metadata.

When a large object, ie. greater than 10 MB, is written to MinIO, the S3 API breaks it into a multipart upload. Part sizes are determined by the client when it uploads. S3 requires each part to be at least 5 MB (except the last part) and no more than 5 GB. An object can have up to 10,000 parts based on the S3 specification. Imagine an object that is 320 MB. If this object is broken into 64 parts, MinIO will write the parts to the drives as part.1, part.2,...up to part.64. The parts are of roughly equal size, for example, the 320 MB object uploaded as multipart would be split into 64 5 MB parts.

Each part that was uploaded is erasure coded across the stripe. Part.1 is the first part of

the object that was uploaded and all of the parts are horizontally striped across the drives. Each part is made up of its data blocks, parity blocks and XL metadata. MinIO rotates writes so the system won't always write data to the same drives and parity to the same drives. Every single object is independently rotated, allowing uniform and efficient use of all drives in the cluster, while also increasing data protection.

For example, in a MinIO cluster running 32 nodes, each with 32 drives, there are 1024 drives. When each object is placed, that object is hashed based on bucket and object names, This hash deterministically places the object in an erasure set. The erasure set is static over a server pool, therefore a hash mode operation will always land in the same erasure set.

MinIO treats all nodes in a cluster the same and all nodes share the same functionality. Combined with metadata being written with the object, the result is that when nodes and/or drives fail, they are simply taken out of service to be replaced when new hardware is available. As a result, MinIO is operationally efficient at scale.

To retrieve an object, MinIO performs a hash calculation to determine where the object was saved, reads the hash and accesses the required erasure set and drives. When the object is read, there are data and parity blocks as described in XL metadata. The default erasure set in MinIO is 12 data and 4 parity, meaning that as long as MinIO can read any 12 drives the object can be served.

In the case of the multipart upload with 64 parts discussed above, the parts will be returned in order from different MinIO nodes - and always be consistent. Every node contains the same logic, the parts are written with their metadata on commit.

BitRot Protection

Silent data corruption or BitRot is a serious problem for drives resulting in the corruption of data without the user's knowledge. As the drives get larger and larger and the data needs to persist for many years, this problem is more common than we imagine. The data bits decay when the magnetic orientation flips and loses polarity. Even solid state drives are prone to this decay when the electrons leak due to insulation defects. There are also other reasons such as wear and tear, voltage spikes, firmware bugs and even cosmic rays.

MinIO's SIMD accelerated implementation of the HighwayHash algorithm ensures that it will never return corrupted data - it captures and heals corrupted objects on the fly. Integrity is ensured from end to end by computing hash on WRITE and verifying it on every READ from the application, across the network and to the memory/drive. The implementation is designed for speed and can achieve hashing speeds over 10 GB/sec per core on Intel CPUs.

MinIO was designed to provide strong data protection using erasure coding and HighwayHash for checksum - and do this at memory speed. There is no performance cost to validating checksums, so MinIO always validates checksums on read to prevent corruption

such as BitRot. Even when an application repeatedly requests the same data, MinIO never blindly trusts drives and always reads and verifies the checksum. This ensures that whatever was written is guaranteed to be exactly the same every single time it is read.

Benefits of Erasure Coding

Erasure coding is better suited for object storage than RAID because objects are immutable blobs that get written once and read many times. Like RAID 5 (parity 1) and RAID 6 (parity 2), MinIO relies on erasure coding (configurable parity between 2 and 8) to protect data from loss and corruption. Erasure coding breaks objects into data and parity blocks, where parity blocks support reconstruction of missing or corrupted data blocks. MinIO distributes both data and parity blocks across nodes and drives in an erasure set. With MinIO's highest level of protection (8 parity or EC:8), you may lose up to half of the total drives and still recover data. MinIO combines multiple erasure sets into a single namespace in order to increase capacity and maintain isolation.

Not only does MinIO erasure coding protect objects against data loss in the event that multiple drives and nodes fail, MinIO also protects and heals at the object level. The ability to heal one object at a time is a dramatic advantage over systems such as RAID that heal at the volume level. A corrupt object could be restored in MinIO in seconds vs. hours in RAID. If a drive goes bad and is replaced, MinIO recognizes the new drive, adds it to the erasure set, and then verifies objects across all drives. More importantly, reads and writes don't affect one another, enabling performance at scale. There are MinIO deployments out there with hundreds of billions of objects across petabytes of storage.

MinIO protects against BitRot, or silent data corruption, which can have many different causes such as power current spikes, bugs in disk firmware and even simply aging drives. MinIO uses the HighwayHash algorithm to compute a hash on read and verify it on write from the application, across the network and to the storage media. This process is highly efficient - it can achieve hashing speeds over 10 GB/sec on a single core on Intel CPUs - and has minimal impact on normal read/write operations across the erasure set.

The erasure code implementation in MinIO results in improved operational efficiency in the datacenter. Unlike RAID and replication, there is no lengthy rebuilding or re-synchronizing data across drives and nodes. It may sound trivial, but moving/copying objects can be very expensive, and a 16TB drive failing and being copied across the datacenter network to another drive places a huge tax on the storage system and network.

About MinIO

MinIO is pioneering high performance, Kubernetes-native object storage for the multi-cloud. The software-defined, Amazon S3-compatible object storage system is used by more than half of the Fortune 500. With 781M+ Docker pulls, MinIO is the fastest-growing cloud object storage company and is consistently ranked by industry analysts as a leader in object storage. Founded in 2014, the company is backed by Intel Capital, Softbank Vision Fund 2, Dell Technologies Capital, Nexus Venture Partners, General Catalyst and key angel investors.

Additional Information
Email: hello@min.io
MinIO Inc.
530B University Avenue,
Palo Alto, CA 94301

Resources
https://min.io
https://docs.min.io/
https://blog.min.io/

MINIO