Apache Ozone Best Practices at Didi

Overview

As Didi's volume of unstructured data continues to grow, the number of stored files has increased significantly, putting mounting pressure on the HDFS metadata service. The deployment of the Ozone service has effectively addressed this pain point. Ozone has been running in production at Didi for over two years. As of now, it manages hundreds of PB of storage and tens of billions of files. The scale of the Ozone cluster is illustrated in Figure 1. This article introduces the evolution of Ozone at Didi.

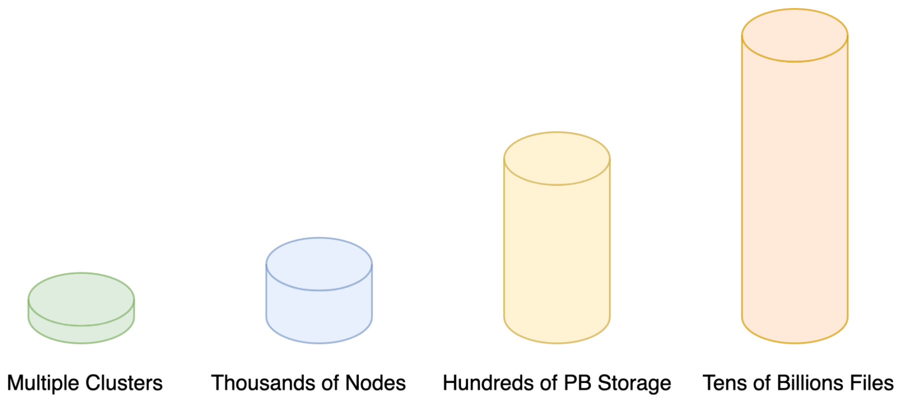


Figure 1. Ozone Cluster Scale

Why Ozone

Apache Ozone is a distributed object storage system and the next-generation bigdata storage engine in the Hadoop ecosystem. Ozone not only inherits the excellent design features of HDFS, such as high reliability and scalability, but also addresses the file count limitation issue in HDFS. Compared to HDFS, Ozone stores metadata in RocksDB, which avoids memory bottlenecks. Furthermore, Ozone separates its metadata services into OM (Ozone Manager) and SCM (Storage Container Manager), improving overall service performance. The data management granularity is upgraded from blocks to containers, greatly reducing the pressure caused by block report storms. Ozone is especially suitable for storing small files. A comparison of Ozone and HDFS architectures is shown in Figure 2.

	HDFS	Ozone
Metadata	In memory	In RocksDB
Max File Count	Hundreds of millions	Tens of billions
Redundancy Mechanism	Replication / EC	Replication / EC

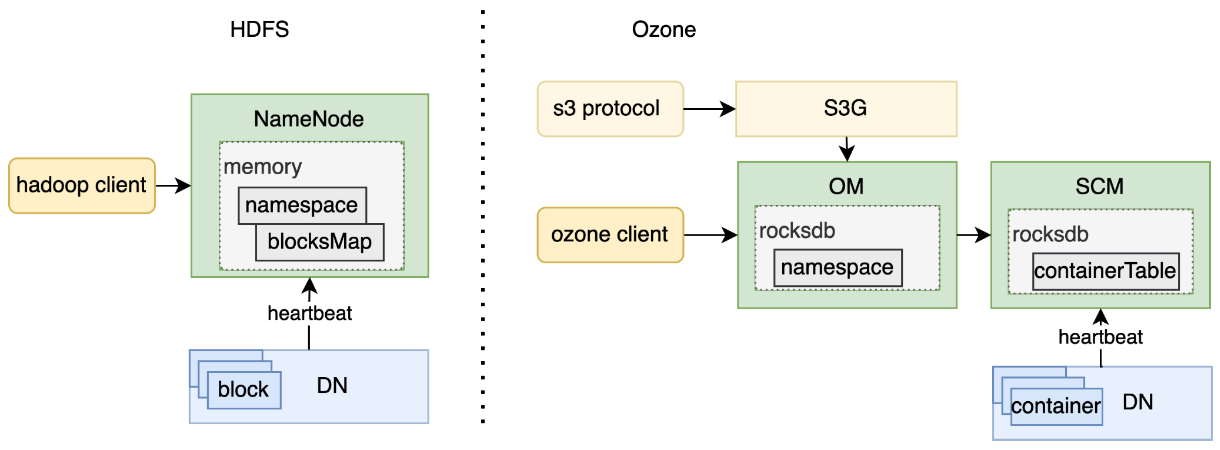


Figure 2. Architecture Comparison between Ozone and HDFS

Practice

1. Support for Multi-Cluster Routing

Ozone stores metadata on disk, resulting in little pressure when storing small files. In practice, we found that as the number of files and access frequency surged, the OM faced challenges related to SSD storage capacity and RPC congestion. To address these issues, we drew inspiration from HDFS ViewFs and implemented multi-cluster routing on the client side. By maintaining a mapping between paths and clusters, requests with different characteristics are routed to their corresponding target paths in the appropriate clusters. For example:

```
vol/bucket/prefix1 --> cluster1(vol/bucketA/prefixX)
vol/bucket/prefix2 --> cluster2(vol/bucketB/prefixY)
```

Figure 3 illustrates the overall architecture. After rolling out this solution, we gradually added multiple clusters and kept the number of files in each cluster under 5 billion. This effectively alleviated RPC pressure and enabled elastic scaling of storage resources.

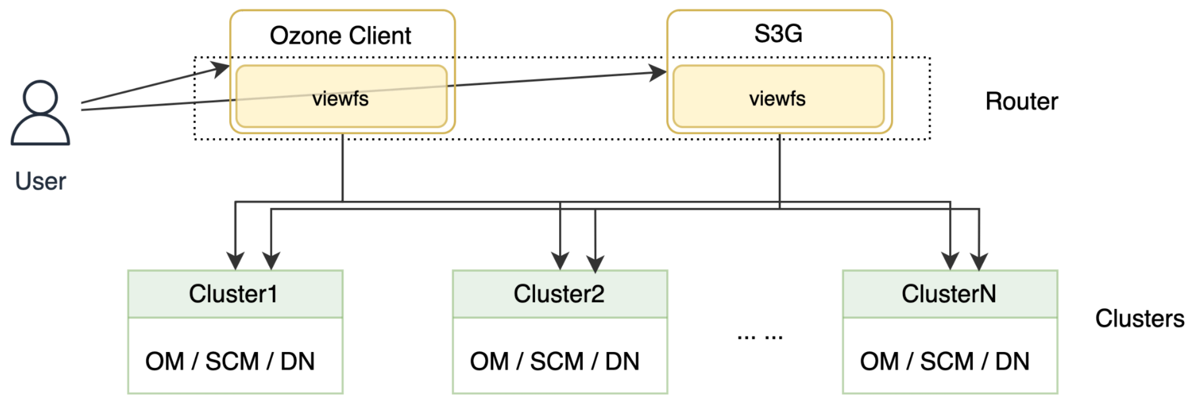


Figure 3. Ozone ViewFs Support Architecture

2. OM Follower Read for S3G in Internal Scenarios

The latency and throughput of OM's RPC processing are critical to the overall performance of the system. In the current OM HA implementation, all RPC requests are handled exclusively by the Leader OM, which may become a performance bottleneck and result in higher metadata access latency.

OM HA uses Apache Ratis (a Java implementation of the Raft consensus algorithm) for state synchronization. By default, OM in HA mode consists of one Leader and several Follower nodes. Write operations are handled by the Leader and replicated to the Followers, while read operations must also go through the Leader to ensure linearizability.

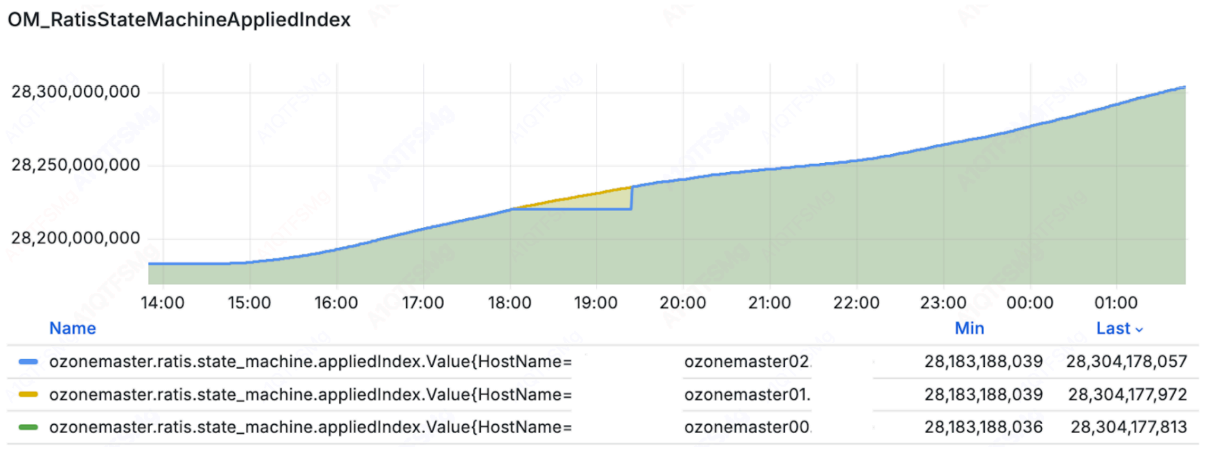
To improve read performance, we explored enabling Followers to serve read requests. The Raft protocol supports linearizable reads through either the ReadIndex or Lease mechanism (already implemented in Ratis, see RATIS-1557). Based on this, we proposed two approaches:

- **Directly reading from Follower nodes**: This approach does not guarantee linearizability and carries the risk of returning stale data.
- **Submitting read requests through Ratis**: This ensures linearizability via the Raft protocol, but in our current testing, its performance did not meet internal latency requirements.

Given internal business requirements, where S3G download latency is critical but data consistency can be relaxed, we optimized both S3G and OM. In S3G, in addition to the original ozoneClient for RPC communication with OM, we introduced a dedicated read client that establishes connections with all OM nodes and prioritizes the best available connection. To select the optimal client, we introduced a heartbeat thread (the `probeTask`, executed every 3 seconds by default) that evaluates the following two factors in real time:

- **Lowest OM latency**: A new API was added to retrieve OM latency. The `probeTask` calls this API to query all OM nodes and select the one with the lowest latency.
- **Freshness of data**: Another new API retrieves the `lastAppliedIndex` of each OM node. The `probeTask` uses this API to check the `lastAppliedIndex` of all OM nodes. If an OM's index falls behind a predefined threshold, its data is considered stale, and that OM will no longer handle read requests. The client will prioritize reading from the OM with the most up-to-date index. As shown in Figure 4, when the appliedIndex of the master02 OM node falls significantly behind, this Follower node will no longer participate in read operations during that period.

```
2025-04-15 10:54:36,876 INFO org.apache.hadoop.ozone.s3.RefreshFollowerClientService:
Sorted clients:
{
FOLLOWER:om2:master02:9862/(0.280119ms)/index:42769979562/count:3791226
FOLLOWER:om1:master01:9862/(0.320800ms)/index:42769979562/count:3791226
LEADER:om0:master00:9862/(0.477696ms)/index:42769979563/count:3791226
}
```



Name		Min	Last ⌄
ozonemaster.ratis.state_machine.appliedIndex.Value{HostName=	ozonemaster02	28,183,188,039	28,304,178,057
ozonemaster.ratis.state_machine.appliedIndex.Value{HostName=	ozonemaster01.	28,183,188,039	28,304,177,972
ozonemaster.ratis.state_machine.appliedIndex.Value{HostName=	ozonemaster00	28,183,188,036	28,304,177,813

Figure 4. The Follower node's index lags behind in a short period

After the above logic, the reference to the persistent read client will continuously update to the optimal client, as shown in Figure 5.

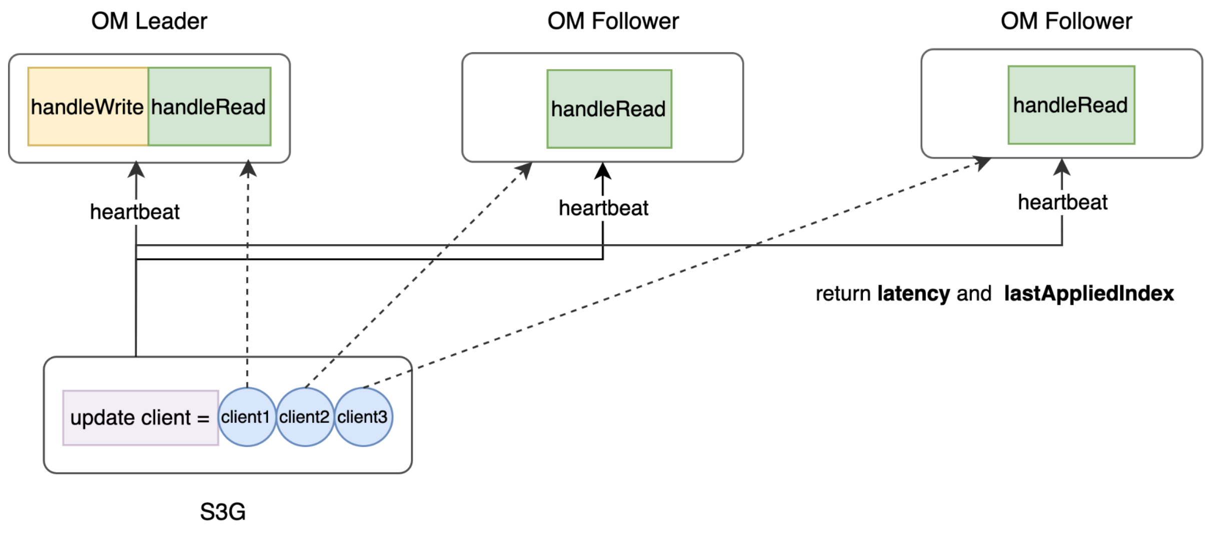
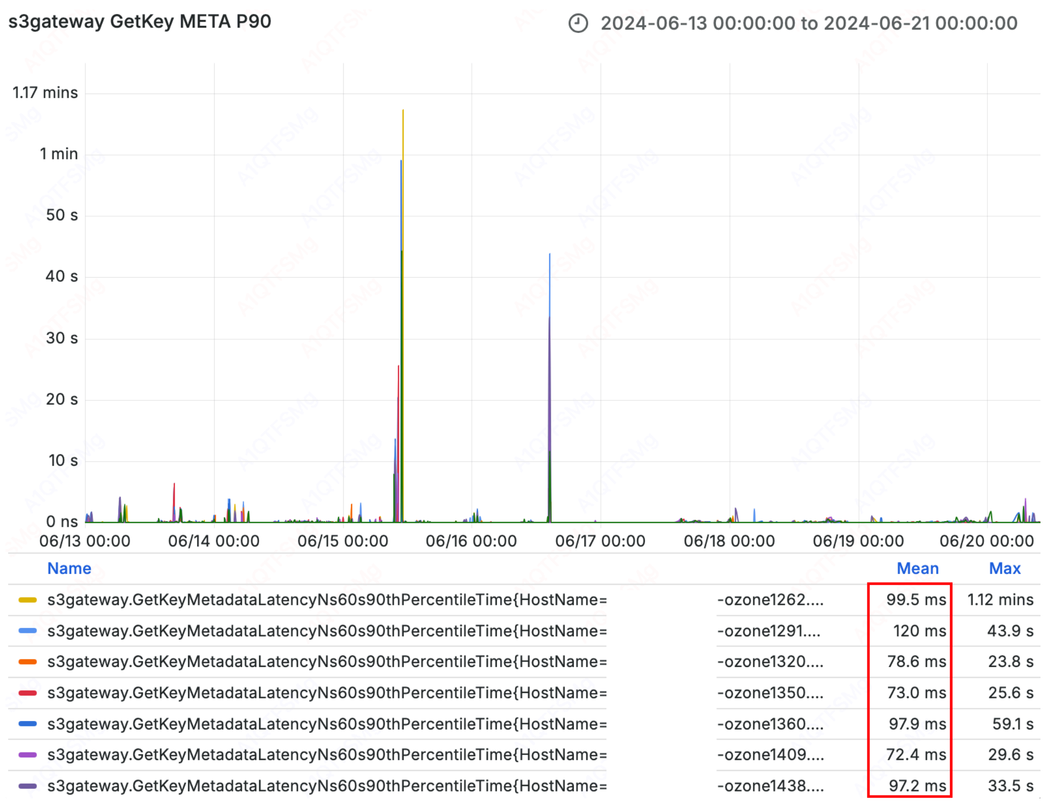


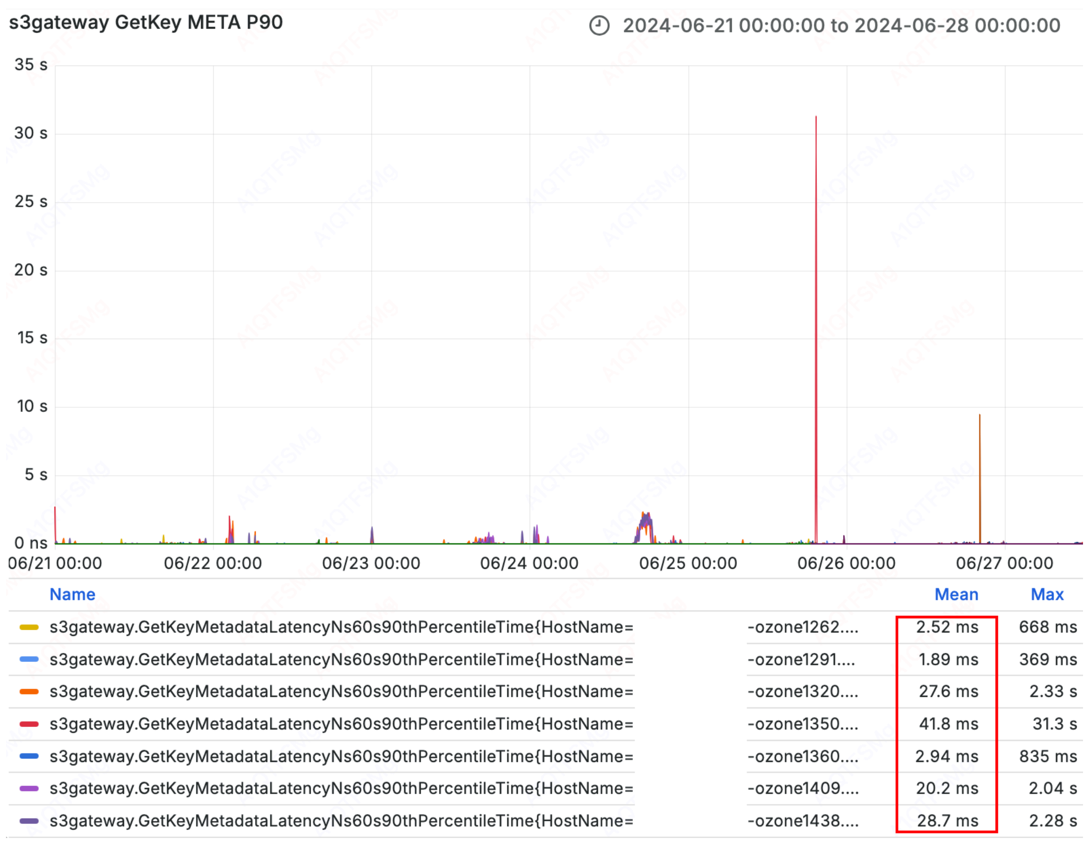
Figure 5. S3G update client continuously

After going online, the S3G download latency was significantly reduced, and downloading data was no longer affected by the RPC pressure from the OM Leader. The p90 latency (GetMetaLatency) of the upgraded S3G decreased from a weekly average of 90ms to 17ms. In the best cases, it dropped from several tens of milliseconds to less than 3ms. The S3G latency monitoring before and after the update is shown in Figure 6-1 and Figure 6-2.



	Name		Mean	Max
🟨	s3gateway.GetKeyMetadataLatencyNs60s90thPercentileTime{HostName=	-ozone1262....	99.5 ms	1.12 mins
🟦	s3gateway.GetKeyMetadataLatencyNs60s90thPercentileTime{HostName=	-ozone1291....	120 ms	43.9 s
🟧	s3gateway.GetKeyMetadataLatencyNs60s90thPercentileTime{HostName=	-ozone1320....	78.6 ms	23.8 s
🟥	s3gateway.GetKeyMetadataLatencyNs60s90thPercentileTime{HostName=	-ozone1350....	73.0 ms	25.6 s
🟦	s3gateway.GetKeyMetadataLatencyNs60s90thPercentileTime{HostName=	-ozone1360....	97.9 ms	59.1 s
🟪	s3gateway.GetKeyMetadataLatencyNs60s90thPercentileTime{HostName=	-ozone1409....	72.4 ms	29.6 s
🟪	s3gateway.GetKeyMetadataLatencyNs60s90thPercentileTime{HostName=	-ozone1438....	97.2 ms	33.5 s

Figure 6-1. S3G Download Latency Monitoring Before Going Online



	Name		Mean	Max
🟨	s3gateway.GetKeyMetadataLatencyNs60s90thPercentileTime{HostName=	-ozone1262....	2.52 ms	668 ms
🟦	s3gateway.GetKeyMetadataLatencyNs60s90thPercentileTime{HostName=	-ozone1291....	1.89 ms	369 ms
🟧	s3gateway.GetKeyMetadataLatencyNs60s90thPercentileTime{HostName=	-ozone1320....	27.6 ms	2.33 s
🟥	s3gateway.GetKeyMetadataLatencyNs60s90thPercentileTime{HostName=	-ozone1350....	41.8 ms	31.3 s
🟦	s3gateway.GetKeyMetadataLatencyNs60s90thPercentileTime{HostName=	-ozone1360....	2.94 ms	835 ms
🟪	s3gateway.GetKeyMetadataLatencyNs60s90thPercentileTime{HostName=	-ozone1409....	20.2 ms	2.04 s
🟪	s3gateway.GetKeyMetadataLatencyNs60s90thPercentileTime{HostName=	-ozone1438....	28.7 ms	2.28 s

Figure 6-2. S3G Download Latency Monitoring After FollowerRead Goes Online

3. Read Performance Optimization

Performance optimization for read and write operations is a challenge that every storage system must address. Read performance optimization covers a wide range of aspects. In general, performance optimization focuses on two key objectives: reducing latency and increasing throughput. In practice, optimizing performance often involves making trade-offs between minimizing latency and maximizing throughput to achieve the best overall system efficiency.

Typical Internal Read/Write Scenarios

Our internal workload is a typical read-heavy / write-light scenario. Users are generally not sensitive to write latency, but are extremely sensitive to read latency, especially in first-frame read scenarios. In a first-frame read scenario, only parts of the file, such as the header and footer are read, without downloading the entire file, as shown below.

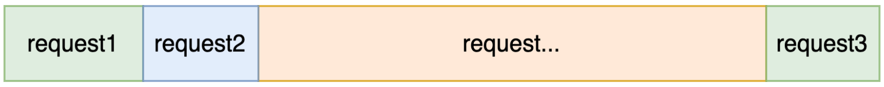


Figure 7. Illustration of a First-Frame Request

This process requires five IO requests, each reading roughly 1MB of data. For a 7200 RPM HDD, a single 1MB access typically takes around 30–50 ms, resulting in a total of roughly 150–250 ms for all five accesses. Our goal is to guarantee the performance of the whole link (including I/O latency, network latency, and RPC overhead, etc.) of the first-frame read, with a target P97 latency of less than 700 milliseconds.

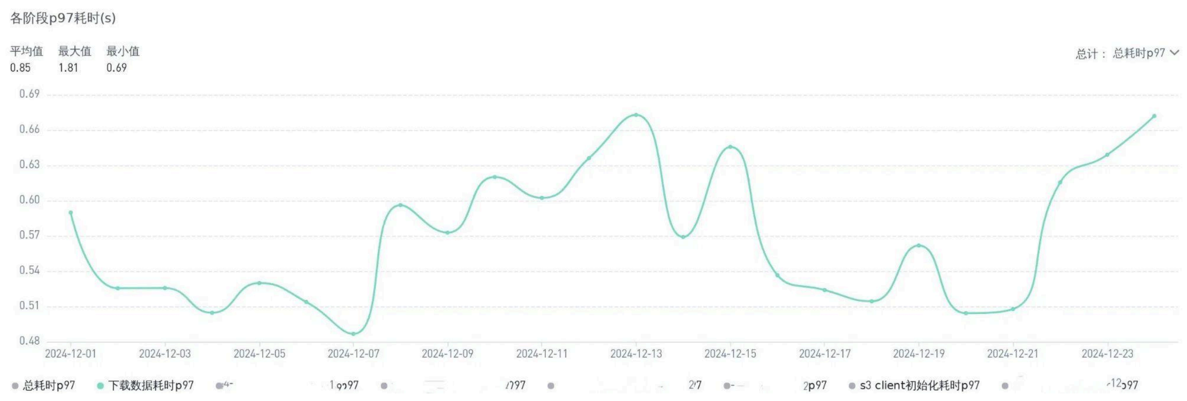


Figure 8. P97 read latency Illustration

In the internal practice process, we focus on the optimization and exploration of the following key aspects based on our own data access characteristics, which will be explained in detail in the following chapters.

3.1 Cache Optimization

Ozone is deployed on HDD-based servers, where disk latency and limited IOPS often lead to access jitter under heavy load, making it difficult to meet first-frame performance requirements. To address this, we designed and implemented a heterogeneous storage-based caching system that improves overall performance while keeping costs under control.

This caching solution enhances read and write performance by optimizing the I/O path and using NVMe drives as a cache layer for HDDs, significantly accelerating data reads and reducing response latency.

3.1.1 Selecting an Appropriate Caching Medium

Memory is the ideal caching medium, such as in HBase's BlockCache. However, our storage-oriented servers have limited memory and off-heap capacity. Relying solely on memory leads to frequent cache evictions, reduced hit rates, and degraded performance, especially with large datasets.

To address this, we use heterogeneous servers with 10 HDDs and 2 NVMe SSDs. HDDs offer large capacity for regular data, while NVMe provides high-speed access for hot data. By distributing data appropriately between HDD and NVMe, we improve cache efficiency and overall system performance.

3.1.2 Cache Granularity and Cached File Selection

In our system, the stored data exceeds several hundred PB, with tens of billions of files. The majority of the data is stored in EC (Erasure Coding) format, and the average size of each file is around tens of MB. We cannot cache all file content, which presents a significant challenge: how to optimize data access performance within limited cache space, especially to improve first-frame access speed.

Considering the above, we decided to implement a strategy of caching the first Chunk of each Block. Each Chunk is 1MB, and we cache all data from the first stripe of the EC block. For example, with EC-6-3-1024K, the caching strategy covers 9MB of data, including 6MB of actual data and 3MB of parity. The 6MB of actual data is sufficient to meet most first-frame access requirements, significantly reducing disk access latency and improving first-frame loading speed.

It is worth noting that this caching strategy provides strong fault tolerance. Even if data is lost on certain DataNodes (DN), the cached data stored on faster NVMe drives can accelerate the reconstruction and recovery process during data read.

Through this first-Chunk-based caching optimization strategy, we achieve efficient data access within limited cache space and ensure rapid recovery during system failures or data loss, maintaining high data availability.

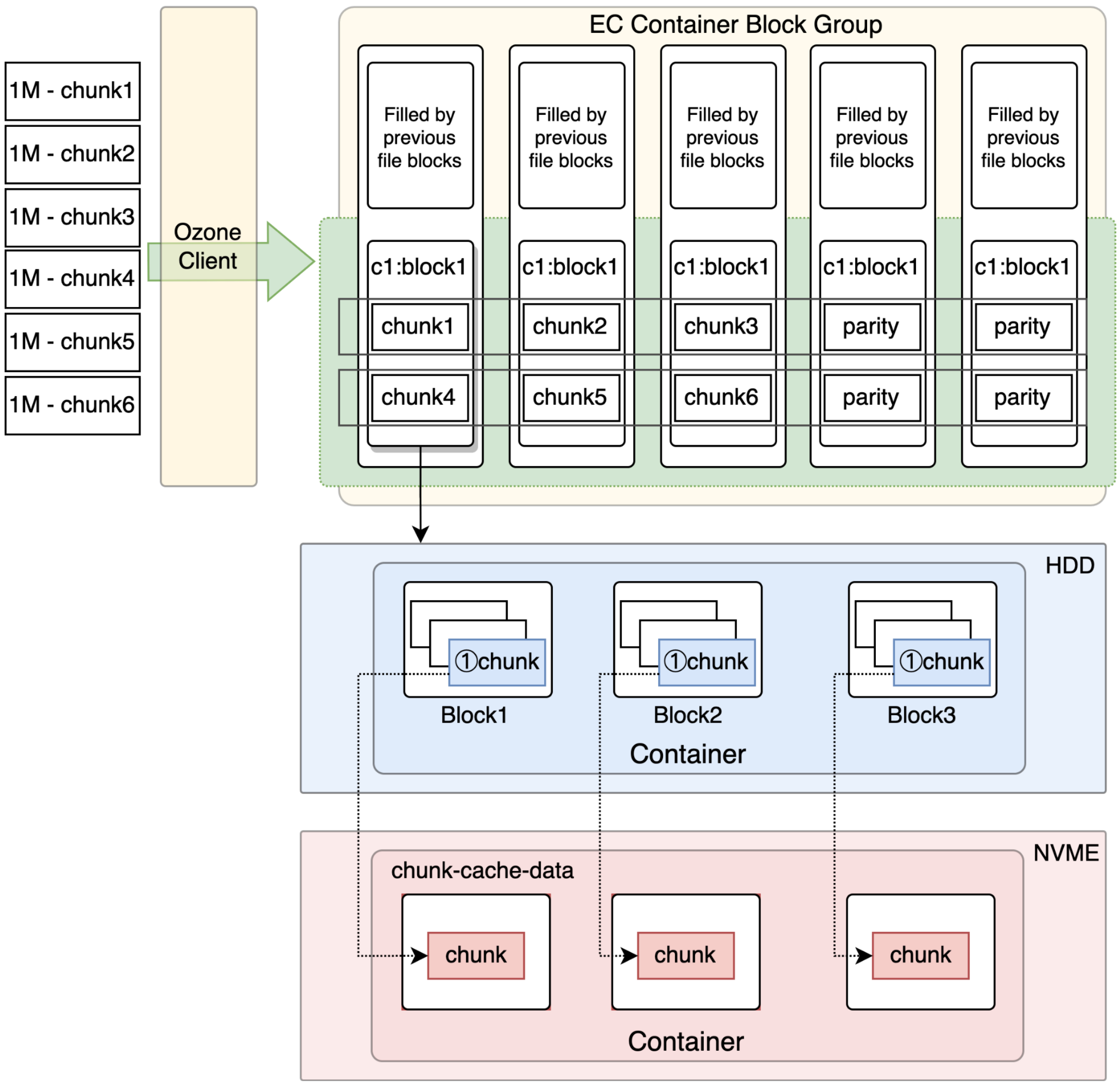


Figure 9. NVMe Cache Architecture

3.1.3 Cache Lifecycle and Read/Write Strategy

The caching strategy covers both read and write operations, with the specific process shown in Figure 10.

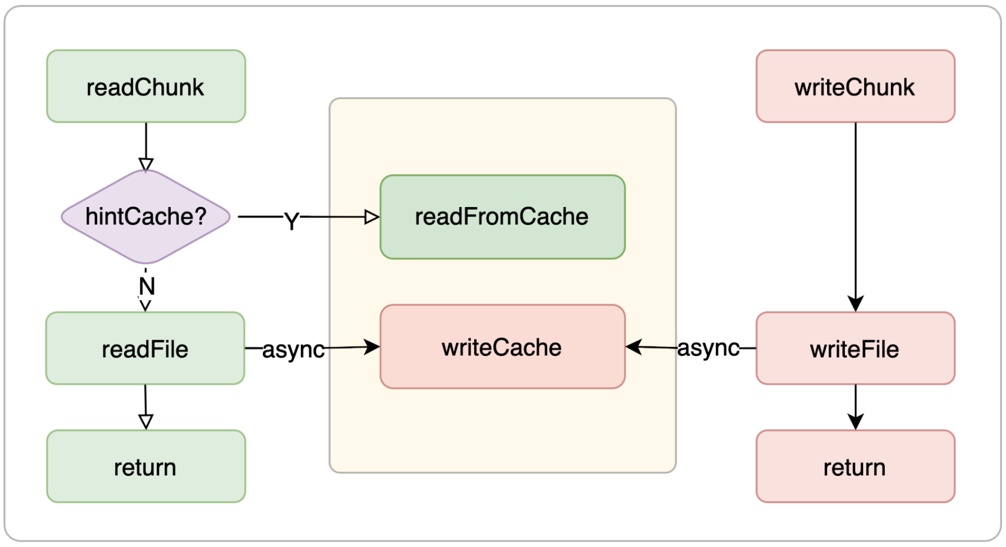


Figure 10. Cache Read/Write Architecture Diagram

- Meta Information Management

Meta information is a critical part of the caching system, and it is stored in a key-value (KV) format to manage the location of each Chunk. The unique identifier for each Chunk (ChunkName) consists of containerID-blockID-chunkNum. Using Meta information, we can quickly query the disk location of each Chunk, enabling efficient cache access.

- Read-Cache

When processing a read request from the user, the system first checks the Meta cache to determine if the requested Chunk is already cached. If the Meta information records the cache location, the system directly reads the corresponding Chunk file from the NVMe drive, greatly improving read speed. If there is no cache hit, the system follows the normal read logic, fetching data from the disk, and asynchronously writes the Chunk to the cache after the read is complete, updating the Meta information.

- Write-Cache

To further improve cache hit rates, we also update the cache during the file write process. When a user performs a write operation, the system not only writes the data to the regular disk storage (e.g., HDD) but also asynchronously writes the first Chunk of each Block to the cache. This mechanism ensures that the relevant data being written is quickly reflected in the cache, improving the hit rate for subsequent accesses to this data.

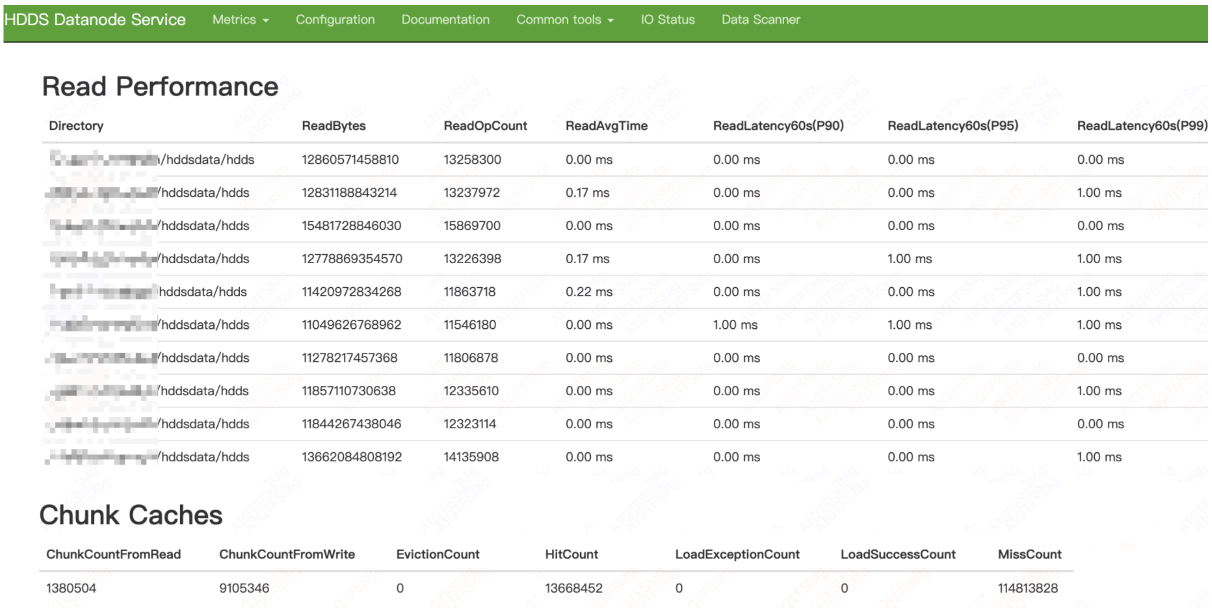
- Cache Management and Eviction Strategy

To maintain efficient cache usage within limited storage space, we use LRU strategy. LRU ensures that when cache space is tight, data with the lowest usage frequency is evicted first, allowing high-frequency accessed data to remain in the cache for extended periods, thereby improving data access performance. When the cache reaches its limit, the system automatically cleans up cache entries based on the LRU rule, freeing up space for new hot data.

- DN-side Cache Monitoring Function

To better understand and manage cache usage, we have developed a dedicated cache monitoring function on the DataNode (DN) side. This monitoring capability can track and record cache hit rates, cache usage, and the health status of the cache in real time, helping us get a comprehensive view of the cache system's performance.

With the cache monitoring feature, we can monitor the cache hit rate in real time, including the number of cache hits, misses, and dynamic changes in the hit rate. This provides us with an intuitive understanding of cache efficiency and helps identify whether cache space is being fully utilized. Monitoring data not only assists in optimizing cache strategies but also helps us identify potential performance bottlenecks.



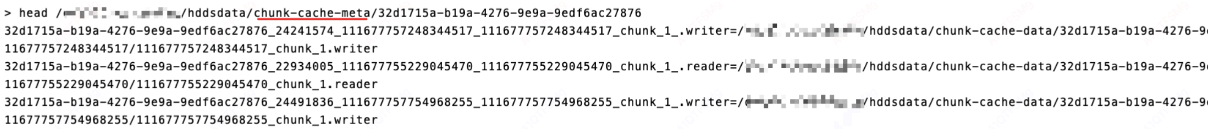
Read Performance

Directory	ReadBytes	ReadOpCount	ReadAvgTime	ReadLatency60s(P90)	ReadLatency60s(P95)	ReadLatency60s(P99)
[redacted]/hddsdata/hdds	12860571458810	13258300	0.00 ms	0.00 ms	0.00 ms	0.00 ms
[redacted]/hddsdata/hdds	12831188843214	13237972	0.17 ms	0.00 ms	0.00 ms	1.00 ms
[redacted]/hddsdata/hdds	15481728846030	15869700	0.00 ms	0.00 ms	0.00 ms	0.00 ms
[redacted]/hddsdata/hdds	12778869354570	13226398	0.17 ms	0.00 ms	1.00 ms	1.00 ms
[redacted]hddsdata/hdds	11420972834268	11863718	0.22 ms	0.00 ms	0.00 ms	1.00 ms
[redacted]/hddsdata/hdds	11049626768962	11546180	0.00 ms	1.00 ms	1.00 ms	1.00 ms
[redacted]/hddsdata/hdds	11278217457368	11806878	0.00 ms	0.00 ms	0.00 ms	0.00 ms
[redacted]/hddsdata/hdds	11857110730638	12335610	0.00 ms	0.00 ms	0.00 ms	1.00 ms
[redacted]/hddsdata/hdds	11844267438046	12323114	0.00 ms	0.00 ms	0.00 ms	0.00 ms
[redacted]/hddsdata/hdds	13662084808192	14135908	0.00 ms	0.00 ms	0.00 ms	1.00 ms

Chunk Caches

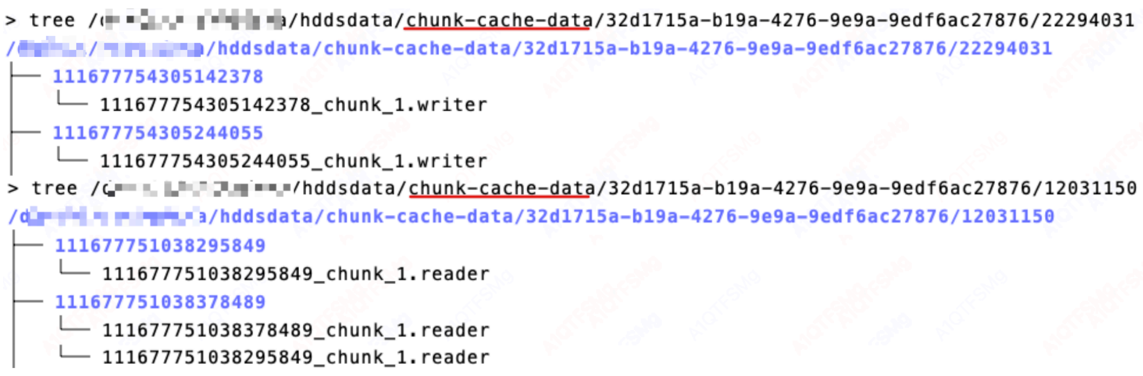
ChunkCountFromRead	ChunkCountFromWrite	EvictionCount	HitCount	LoadExceptionCount	LoadSuccessCount	MissCount
1380504	9105346	0	13668452	0	0	114813828

Figure 11. Cache and Read Performance Overview



```
> head /[redacted]/hddsdata/chunk-cache-meta/32d1715a-b19a-4276-9e9a-9edf6ac27876
32d1715a-b19a-4276-9e9a-9edf6ac27876_24241574_111677757248344517_111677757248344517_chunk_1_.writer=/[redacted]/hddsdata/chunk-cache-data/32d1715a-b19a-4276-9
111677757248344517/111677757248344517_chunk_1.writer
32d1715a-b19a-4276-9e9a-9edf6ac27876_22934005_111677755229045470_111677755229045470_chunk_1_.reader=/[redacted]/hddsdata/chunk-cache-data/32d1715a-b19a-4276-9
111677755229045470/111677755229045470_chunk_1.reader
32d1715a-b19a-4276-9e9a-9edf6ac27876_24491836_111677757754968255_111677757754968255_chunk_1_.writer=/[redacted]/hddsdata/chunk-cache-data/32d1715a-b19a-4276-9
111677757754968255/111677757754968255_chunk_1.writer
```

Figure 12-1. Illustration of Cache Metadata Content



```
> tree /[redacted]a/hddsdata/chunk-cache-data/32d1715a-b19a-4276-9e9a-9edf6ac27876/22294031
/[redacted]/[redacted]a/hddsdata/chunk-cache-data/32d1715a-b19a-4276-9e9a-9edf6ac27876/22294031
├── 111677754305142378
│   └── 111677754305142378_chunk_1.writer
├── 111677754305244055
│   └── 111677754305244055_chunk_1.writer
> tree /[redacted]/hddsdata/chunk-cache-data/32d1715a-b19a-4276-9e9a-9edf6ac27876/12031150
/[redacted]a/hddsdata/chunk-cache-data/32d1715a-b19a-4276-9e9a-9edf6ac27876/12031150
├── 111677751038295849
│   └── 111677751038295849_chunk_1.reader
├── 111677751038378489
│   └── 111677751038378489_chunk_1.reader
│   └── 111677751038295849_chunk_1.reader
```

Figure 12-2. Illustration of Cached Chunk

This improvement brought us at least a 100 ms performance gain.

3.2 Concurrency Optimization

Optimizing the granularity of locks reduces contention and enhances system performance in multi-threaded scenarios.

In practice, we observed latency issues even when repeatedly reading the same data block. We noticed that this was related to the spin lock in ChunkUtils#processFileExclusive, as addressed by the community patch HDDS-11281.

The core changes introduced in this patch are as follows:

Before improvement:

```
  static <T> T processFileExclusively(Path path, Supplier<T> op)
      throws InterruptedException {
    long period = 1;
    for (;;) {
      if (LOCKS.add(path)) {
        break;
      } else {
        Thread.sleep(period);
        // exponentially backoff until the sleep time is over 1 second.
        if (period < 1000) {
          period *= 2;
        }
      }
    }
    try {
      return op.get();
    } finally {
      LOCKS.remove(path);
    }
  }
```

After improvement:

```
private static Striped<ReadWriteLock> fileStripedLock =
    Striped.readWriteLock(DEFAULT_FILE_LOCK_STRIPED_SIZE);

try (AutoCloseableLock ignoredLock = getFileWriteLock(path)) {
      FileChannel channel = null;
      try {
        channel = open(path, WRITE_OPTIONS, NO_ATTRIBUTES);
        try (FileLock ignored = channel.lock()) {
          return writeDataToChannel(channel, data, offset);
        }
      } catch (IOException e) {
        throw new UncheckedIOException(e);
      } finally {
        closeFile(channel, sync);
      }
    }
```

The improvements mainly focus on the following two aspects:

- Lock Retry Mechanism

In the original implementation, the code repeatedly checks LOCKS.add(path) to determine whether the lock can be acquired. If not, the thread sleeps using Thread.sleep(period) with exponential backoff until the lock is successfully obtained. This loop-based retry mechanism causes excessive thread context switching and resource waste.

After improvement, AutoCloseableLock is used to manage the lock, making the retry logic more concise and avoiding the complexity of manual lock handling.

- Lock Granularity and Performance

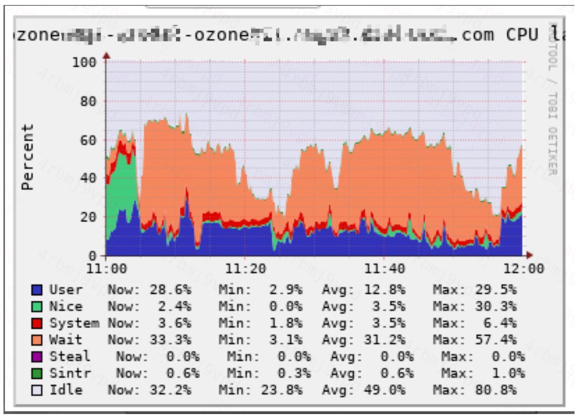
The original code uses a global LOCKS set to manage locks for file paths. This approach results in coarse-grained locking, which can lead to performance bottlenecks when multiple threads frequently access the same file path.

After improvement, the lock granularity and management are more flexible, reducing lock contention and improving concurrency and overall system performance.

This optimization led to a performance gain of at least 50ms.

3.3 Disk I/O Optimization

Disk I/O issues are a major challenge in system performance optimization, especially in HDD-based deployment environments, where I/O wait (IOWait) problems are particularly prominent. Prolonged I/O waits lead to a significant decrease in throughput and an increase in latency, severely affecting system performance.

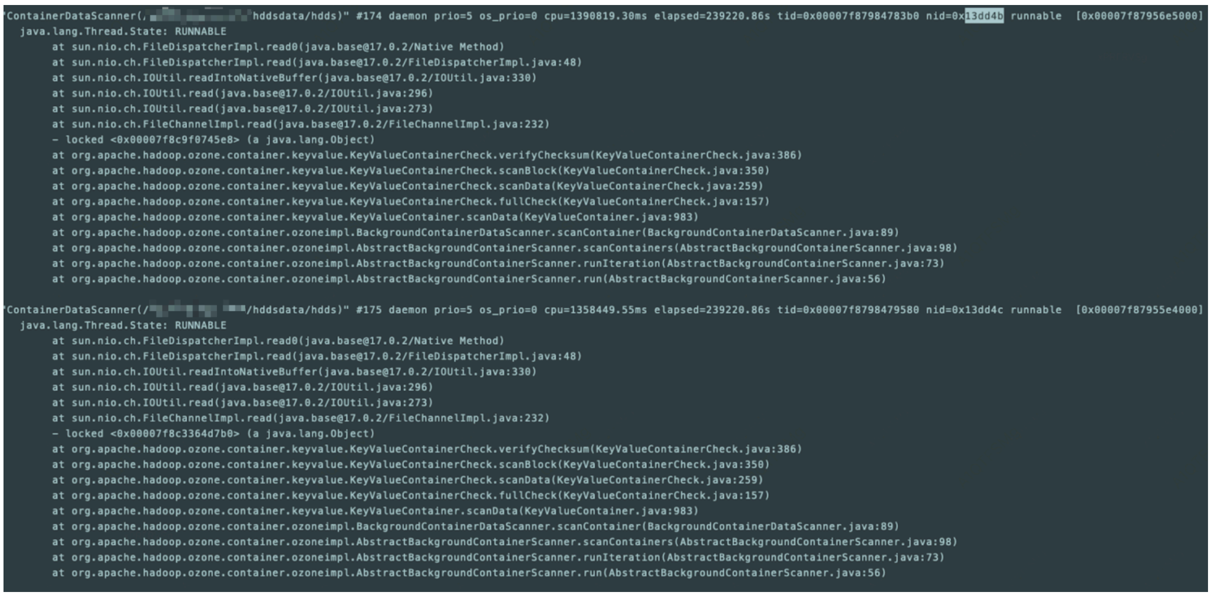


```
■ User   Now: 28.6%   Min:  2.9%   Avg: 12.8%   Max: 29.5%
■ Nice   Now:  2.4%   Min:  0.0%   Avg:  3.5%   Max: 30.3%
■ System Now:  3.6%   Min:  1.8%   Avg:  3.5%   Max:  6.4%
■ Wait   Now: 33.3%   Min:  3.1%   Avg: 31.2%   Max: 57.4%
■ Steal   Now:  0.0%   Min:  0.0%   Avg:  0.0%   Max:  0.0%
■ Sintr   Now:  0.6%   Min:  0.3%   Avg:  0.6%   Max:  1.0%
□ Idle   Now: 32.2%   Min: 23.8%   Avg: 49.0%   Max: 80.8%
```

Figure 13. Ozone IOWait Illustration

To optimize I/O performance, we rely on system monitoring tools such as pidstat, ps, top, atop, iotop, and readlink to analyze I/O usage and identify bottlenecks. During the analysis, we found that many I/O operations are triggered by management actions, such as data deletion, EC data reconstruction, and decommissioning machines. These unordered I/O requests consume a large amount of resources, with the impact being especially noticeable under high load.

The figure below is a sample jstack from a DataNode, showing that the periodic scans performed by the BackgroundContainerDataScanner are the main cause of the high IOWait observed on the server.



```
"ContainerDataScanner(/[redacted]hddsdata/hdds)" #174 daemon prio=5 os_prio=0 cpu=1390819.30ms elapsed=239220.86s tid=0x00007f87984783b0 nid=0x13dd4b runnable  [0x00007f87956e5000]
   java.lang.Thread.State: RUNNABLE
        at sun.nio.ch.FileDispatcherImpl.read0(java.base@17.0.2/Native Method)
        at sun.nio.ch.FileDispatcherImpl.read(java.base@17.0.2/FileDispatcherImpl.java:48)
        at sun.nio.ch.IOUtil.readIntoNativeBuffer(java.base@17.0.2/IOUtil.java:330)
        at sun.nio.ch.IOUtil.read(java.base@17.0.2/IOUtil.java:296)
        at sun.nio.ch.IOUtil.read(java.base@17.0.2/IOUtil.java:273)
        at sun.nio.ch.FileChannelImpl.read(java.base@17.0.2/FileChannelImpl.java:232)
        - locked <0x00007f8c9f0745e8> (a java.lang.Object)
        at org.apache.hadoop.ozone.container.keyvalue.KeyValueContainerCheck.verifyChecksum(KeyValueContainerCheck.java:386)
        at org.apache.hadoop.ozone.container.keyvalue.KeyValueContainerCheck.scanBlock(KeyValueContainerCheck.java:350)
        at org.apache.hadoop.ozone.container.keyvalue.KeyValueContainerCheck.scanData(KeyValueContainerCheck.java:259)
        at org.apache.hadoop.ozone.container.keyvalue.KeyValueContainerCheck.fullCheck(KeyValueContainerCheck.java:157)
        at org.apache.hadoop.ozone.container.keyvalue.KeyValueContainer.scanData(KeyValueContainer.java:983)
        at org.apache.hadoop.ozone.container.ozoneimpl.BackgroundContainerDataScanner.scanContainer(BackgroundContainerDataScanner.java:89)
        at org.apache.hadoop.ozone.container.ozoneimpl.AbstractBackgroundContainerScanner.scanContainers(AbstractBackgroundContainerScanner.java:98)
        at org.apache.hadoop.ozone.container.ozoneimpl.AbstractBackgroundContainerScanner.runIteration(AbstractBackgroundContainerScanner.java:73)
        at org.apache.hadoop.ozone.container.ozoneimpl.AbstractBackgroundContainerScanner.run(AbstractBackgroundContainerScanner.java:56)

"ContainerDataScanner(/[redacted]/hddsdata/hdds)" #175 daemon prio=5 os_prio=0 cpu=1358449.55ms elapsed=239220.86s tid=0x00007f8798479580 nid=0x13dd4c runnable  [0x00007f87955e4000]
   java.lang.Thread.State: RUNNABLE
        at sun.nio.ch.FileDispatcherImpl.read0(java.base@17.0.2/Native Method)
        at sun.nio.ch.FileDispatcherImpl.read(java.base@17.0.2/FileDispatcherImpl.java:48)
        at sun.nio.ch.IOUtil.readIntoNativeBuffer(java.base@17.0.2/IOUtil.java:330)
        at sun.nio.ch.IOUtil.read(java.base@17.0.2/IOUtil.java:296)
        at sun.nio.ch.IOUtil.read(java.base@17.0.2/IOUtil.java:273)
        at sun.nio.ch.FileChannelImpl.read(java.base@17.0.2/FileChannelImpl.java:232)
        - locked <0x00007f8c3364d7b0> (a java.lang.Object)
        at org.apache.hadoop.ozone.container.keyvalue.KeyValueContainerCheck.verifyChecksum(KeyValueContainerCheck.java:386)
        at org.apache.hadoop.ozone.container.keyvalue.KeyValueContainerCheck.scanBlock(KeyValueContainerCheck.java:350)
        at org.apache.hadoop.ozone.container.keyvalue.KeyValueContainerCheck.scanData(KeyValueContainerCheck.java:259)
        at org.apache.hadoop.ozone.container.keyvalue.KeyValueContainerCheck.fullCheck(KeyValueContainerCheck.java:157)
        at org.apache.hadoop.ozone.container.keyvalue.KeyValueContainer.scanData(KeyValueContainer.java:983)
        at org.apache.hadoop.ozone.container.ozoneimpl.BackgroundContainerDataScanner.scanContainer(BackgroundContainerDataScanner.java:89)
        at org.apache.hadoop.ozone.container.ozoneimpl.AbstractBackgroundContainerScanner.scanContainers(AbstractBackgroundContainerScanner.java:98)
        at org.apache.hadoop.ozone.container.ozoneimpl.AbstractBackgroundContainerScanner.runIteration(AbstractBackgroundContainerScanner.java:73)
        at org.apache.hadoop.ozone.container.ozoneimpl.AbstractBackgroundContainerScanner.run(AbstractBackgroundContainerScanner.java:56)
```

Figure 14. Ozone BackgroundContainerDataScanner jstack

To address the issue, we implemented several optimization measures.

3.3.1 Throttling Container DataScanner

The DataNode's background service, BackgroundContainerDataScanner, launches a separate thread for scanning containers on each volume. While this scanning is critical for monitoring the health and status of containers, it may occur at any time, potentially causing I/O fluctuations and increasing system IOWait.

To mitigate this, we restrict scanning to off-peak hours—specifically between midnight and 5 AM—to avoid impacting business workloads during peak periods. Additionally, we limit the amount of data processed in each scan iteration to reduce system load and ensure that other critical operations remain unaffected.

Below are some parameter configurations used in practice:

```xml
<property>
   <name>hdds.datanode.read.chunk.threads.per.volume</name>
   <value>20</value>
</property>
<property>
```

```
    <name>hdds.container.scrub.enabled</name>
    <value>true</value>
</property>
<property>
    <name>hdds.container.scrub.volume.bytes.per.second</name>
    <value>2097152</value>
</property>
<property>
    <name>hdds.container.scrub.on.demand.volume.bytes.per.second</name>
    <value>2097152</value>
</property>
<property>
    <name>hdds.container.scrub.off-peak.hour</name>
    <value>0,1,2,3,4,5</value>
</property>
<property>
    <name>hdds.container.scrub.off-peak.ratio</name>
    <value>3</value>
</property>
```

New Parameter Descriptions:

- `hdds.container.scrub.off-peak.hour`: Defines the specific time range considered as off-peak hours.
- `hdds.container.scrub.off-peak.ratio`: Specifies the speed ratio of scrub operations between off-peak and peak hours.

For example, setting this ratio to 3 means that the scrub speed during off-peak hours is three times faster than during peak hours. This configuration ensures efficient scrubbing during low-traffic periods while throttling it during peak times to avoid adding unnecessary I/O pressure on the DataNode.

3.3.2 Dynamically Planned Data Deletion

Ozone's data deletion process consists of three stages:

- Stage 1: The OM selects the containers and blocks to be deleted and sends the requests to the SCM.
- Stage 2: The SCM issues delete commands to the corresponding DataNodes based on the replica status and tracks the deletion progress.
- Stage 3: The DNs execute the deletion operations.

To minimize the performance impact of deletion operations, we introduced a dynamic parameter tuning mechanism that adjusts deletion intervals and batch sizes based on system load. During peak hours, the number of deletion commands is reduced to lower I/O pressure and ensure stable read/write performance. During off-peak hours, the system increases deletion throughput to leverage idle resources and accelerate the cleanup process.

This dynamic adjustment strategy effectively avoids performance degradation during high load and improves overall system efficiency when resources are sufficient.

3.3.3 Point-to-Point Data Transfer

In large-scale server environments, issues such as rack power overload or switch failures often require some machines to be taken offline and decommissioned. These decommission operations can trigger large-scale data transfers and replications within the Ozone system, resulting in a high volume of random I/O requests and degraded service performance.

To mitigate this impact, we introduced a point-to-point data transfer mechanism. When a machine needs to be decommissioned, the system selects an idle server from another rack to directly perform data migration with the target machine, avoiding involvement of other nodes. This approach effectively reduces the scope of random I/O requests, eases system load, and ensures stable business operations.

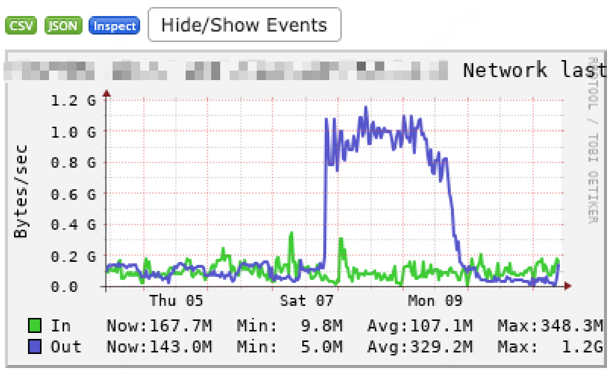


Figure 15. Point-to-Point Data Transfer network usage

3.4 Request Retry and Fault Tolerance

In practice, we observed that a sudden increase in the latency of certain individual requests can severely degrade system performance. This phenomenon is commonly referred to as "Long Request Blocking." To address this issue, we have implemented the following measures:

- Server-side Fast Failure & Client-side Retry

To prevent long request blocking and resource occupation, when a request times out, the server actively disconnects the request and notifies the client to retry. This approach helps reduce a large number of ineffective waiting operations, prevents requests from occupying system resources for extended periods, and ensures the system can continue processing other requests.

- DN-side Quick Retry during readChunk

Considering the relatively high cost of user-initiated retries for data reads, we optimized the retry strategy to minimize unnecessary retries. Noticing that the normal disk response time for reading a 1MB chunk is about 50ms, we set a threshold: if the response time exceeds 100ms, the DN will proactively abort the current read request and initiate a new thread to access the chunk.

This effectively prevents threads from being blocked for extended periods, improving the system's throughput and response speed.

3.5 S3G Read Cache Optimization

In practice, we observed that some slow queries were caused not by slow data reads from the DN, but by delays in S3G receiving the data, which led to increased *first-frame* latency for the application. Code analysis revealed that during data reception from the DN, S3G uses a `byteBuffer` to read data per I/O operation. The default buffer size is 4KB, which can result in multiple network I/O operations. Increasing the buffer size to match the typical request size can help ensure that data transmission is completed in a single network I/O.

In the community-optimized version, the buffer size has already been increased to 4MB. See issue: HDDS-11483.

4. EC Practice

At the beginning, we adopted a 3 replica storage policy. However, as data grew rapidly, we faced significant storage cost pressure. With daily data growth exceeding 1PB and the annual total expected to surpass 500PB, the cost of storage hardware far exceeded our budget. As a result, we decided to revise our storage policy.

Erasure Coding (EC) provides an alternative to traditional replication. We selected the EC-6-3-1024K policy to replace the 3 replica solution. It reduces the replication factor to approximately 1.5, effectively cutting storage usage by half compared to 3 replica. The adoption of EC has delivered substantial cost savings. However, during the practice of EC-based storage, we also encountered several challenges, as outlined below.

4.1 Efficient Deletion

During the data deletion process, we encountered performance bottlenecks, where deletion speed fluctuated significantly. These inconsistencies hindered steady progress and led to a growing backlog of data pending deletion.

Upon analyzing the code, we identified a critical detail. When SCM issues deletion requests for block replicas, it does not dispatch all requests to the DataNodes (DNs) at once. For example, in the case of EC data using the EC-6-3-1024K format, deletion requires removing 9 replicas. However, SCM often only issues deletion requests for 5 or 6 of them. As a result, even if the DNs successfully delete those replicas, the remaining 3 are not marked as deleted. These replicas are not cleared in time and remain until the system times out and re-initiates the deletion process.

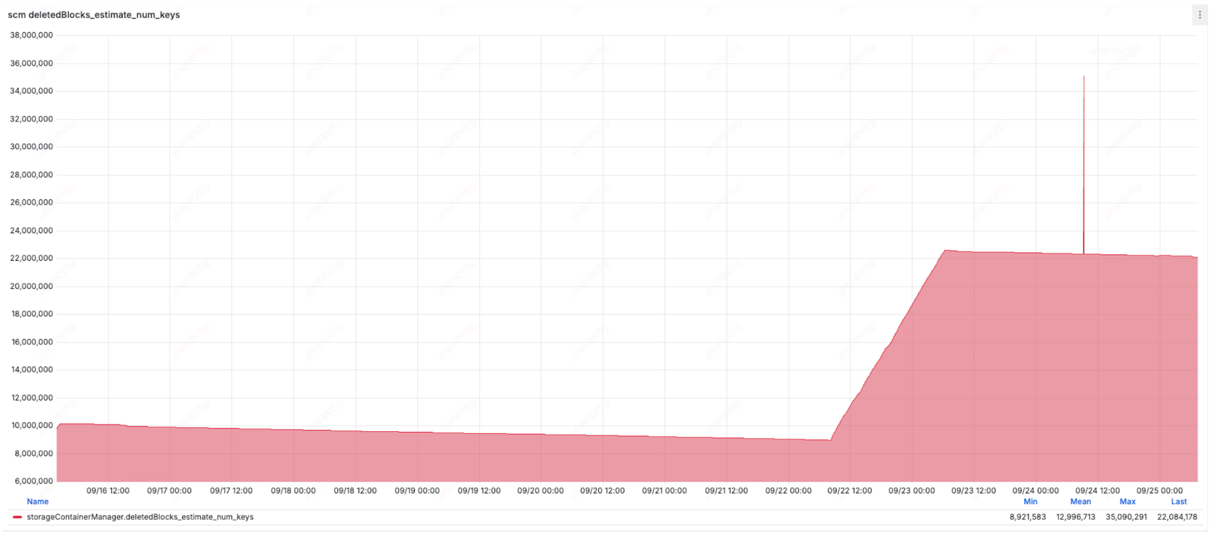


Figure 16-1. Illustration of Deletion Backlog Before Optimization

We submitted HDDS-11498: Improve SCM Deletion Efficiency, and with the support of the community, successfully implemented the enhancement. This optimization significantly improved the efficiency of deletion.

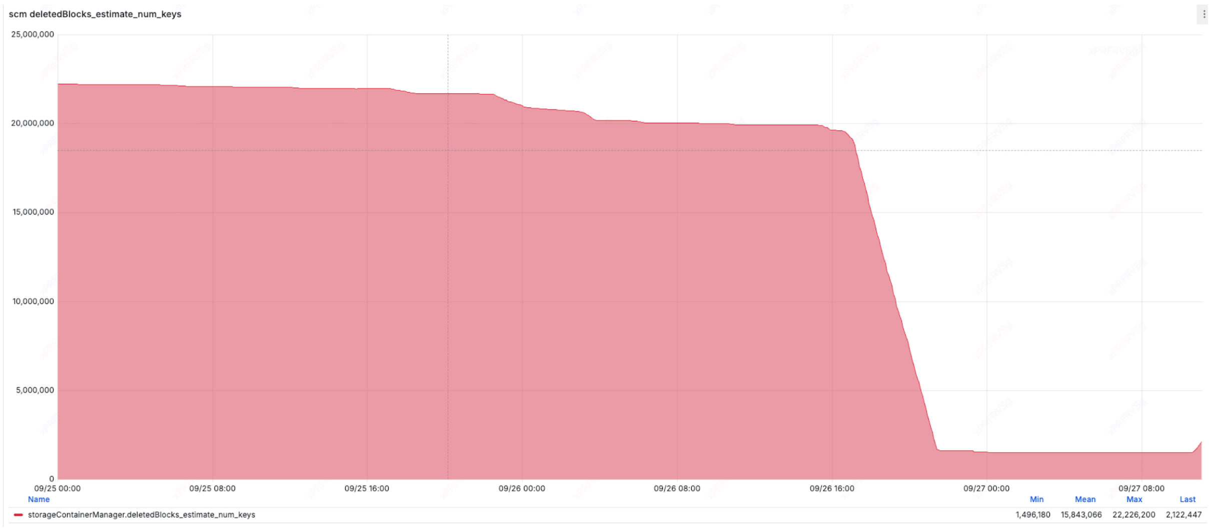


Figure 16-2. Illustration of Deletion Process After Optimization

Deletion parameters used:

```
-- OM
ozone.path.deleting.limit.per.task 150000
ozone.directory.deleting.service.interval 180s
ozone.key.deleting.limit.per.task 150000
ozone.block.deleting.service.interval 180s
-- SCM
hdds.scm.block.deletion.per-interval.max 2000000
hdds.scm.block.deleting.service.interval 300s
```

4.2 EC Replica Insufficient Issues

After restarting the SCM service, we encountered an issue where clients frequently reported errors such as "There are insufficient datanodes to read the EC block," which negatively impacted the stability of the service.

The root cause of the replica shortage issue lay in the design of SCM's Safe Mode. Specifically, Safe Mode did not originally support EC containers. In theory, when SCM is in Safe Mode and encounters an EC container, it should ensure that the container has the minimum required number of replicas. For example, with the EC-6-3-1024K policy, a container should have at least 6 data replicas.

However, earlier versions mistakenly treated EC containers as if they were using a 3 replica policy. As a result, SCM would consider the container "healthy" once it received just 3 replica reports, leading to a premature exit from Safe Mode.

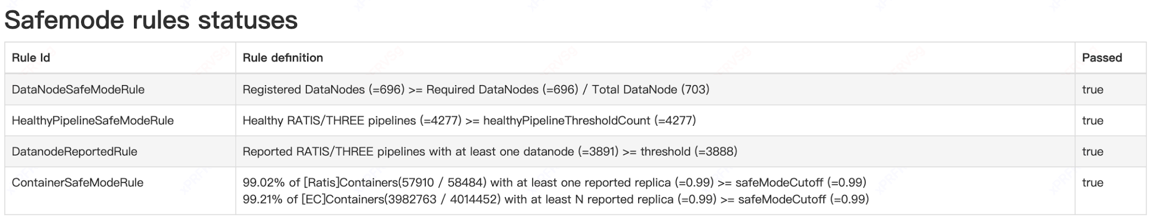
This behavior caused two major issues:

- **Replica Insufficient**: Since the number of EC replicas did not meet the expected 6, clients encountered errors when accessing those containers, significantly affecting user experience and service stability.

- **Incorrect Safe Mode exit:** SCM exited Safe Mode prematurely without verifying that EC containers had sufficient replicas.

To effectively resolve this issue, we contributed two pull requests to the community:

- **HDDS-11209: Avoid insufficient EC pipelines in the container –** This PR addressed the potential issue where OM might cache incomplete EC pipelines, ensuring the pipelines are fully cached.
- **HDDS-11243: SCM SafeModeRule Support EC** – This PR enhanced SCM's Safe Mode logic to fully support EC containers, preventing the replica shortage issue caused by the previous lack of EC handling.


Safemode rules statuses

Rule Id	Rule definition	Passed
DataNodeSafeModeRule	Registered DataNodes (=696) >= Required DataNodes (=696) / Total DataNode (703)	true
HealthyPipelineSafeModeRule	Healthy RATIS/THREE pipelines (=4277) >= healthyPipelineThresholdCount (=4277)	true
DatanodeReportedRule	Reported RATIS/THREE pipelines with at least one datanode (=3891) >= threshold (=3888)	true
ContainerSafeModeRule	99.02% of [Ratis]Containers(57910 / 58484) with at least one reported replica (=0.99) >= safeModeCutoff (=0.99) 99.21% of [EC]Containers(3982763 / 4014452) with at least N reported replica (=0.99) >= safeModeCutoff (=0.99)	true

Figure 17. Safemode rules statues

4.3 Repeated EC Reconstruction Issue

During EC block recovery, a problem was encountered where reconstruction repeatedly failed, triggering retries across many machines and resulting in a large number of abnormal I/O operations. The error message was as follows:

```
java.lang.IllegalArgumentException: The chunk list has 2 entries, but the
```

```
checksum chunks has 3 entries.
They should be equal in size.
```

Due to the limited logging available on the DN, we added additional logs on some DNs to write reconstruction errors into the `audit.log` file. These logs revealed numerous issues with block reconstruction.

```
2024-07-25 07:25:15,830 | ERROR | DNAudit | user=null | ip=null |
op=RECOVER_EC_BLOCK {blockLocationInfo={blockID={conID: 951772 locID:
113750155032021583 bcsId: 0}, length=25165824, offset=0, token=null,
pipeline=Pipeline[ Id: cc205dc9-49a1-4c07-92b9-0c27504268b6, Nodes: ...,
excludedSet: , ReplicationConfig: EC{rs-6-3-1024k}, State:CLOSED, leaderId:,
CreationTimestamp2024-07-25T07:23:12.013617185+08:00[Asia/Shanghai]],
createVersion=0, partNumber=0}} | ret=FAILURE |
java.lang.IllegalArgumentException: The chunk list has 4 entries, but the
checksum chunks has 5 entries. They should be equal in size.
```

Upon further analysis, we traced that the reconstruction failures occurred due to miscalculated checksum chunk selection during recovery.

In EC recovery, the system chooses a smaller `BlockGroupLength` to reconstruct the data. However, during validation, it mistakenly used the chunk size from the corrupted (dirty) data to verify the reconstructed data. To ensure data is correctly validated, the chunk size used for verification should match the smaller `BlockGroupLength` used for reconstruction.

After identifying the correct fix, we submitted a patch via PR **HDDS-10985: EC Reconstruction failed because the size of currentChunks was not equal to checksumBlockDataChunks**, which successfully resolved the issue of repeated EC reconstructions.

4.4 Efficient Conversion from Replica to EC

We initially used Hadoop Distcp for converting historical data to EC storage, but encountered limitations such as unclear migration progress and inflexible error handling. To improve migration efficiency and flexibility, we switched to a self-developed Spark-based migration tool for the following reasons:

- **Ozone is fully compatible with Spark**, enabling efficient processing of large-scale data;
- **Spark offers customizable migration logic,** checksum validation, and fault tolerance, which meets our complex requirements.

Conversion process:

- Parse Ozone Image files to extract metadata for the 3 replica data targeted for conversion;
- Use Spark to read the metadata and download the corresponding data;
- Write the data in EC format using the EC-6-3-1024K configuration;
- Perform checksum and data volume verification to ensure accuracy.

Through this enhancement, we achieved a highly efficient and reliable migration from 3 replica storage to EC-based storage.

5. Availability And Stability Practices

In addition to the Follower Read optimization, read performance improvements, adoption and improvements of EC storage, we have also driven a series of enhancements in other areas, as outlined below:

5.1 Service Monitoring

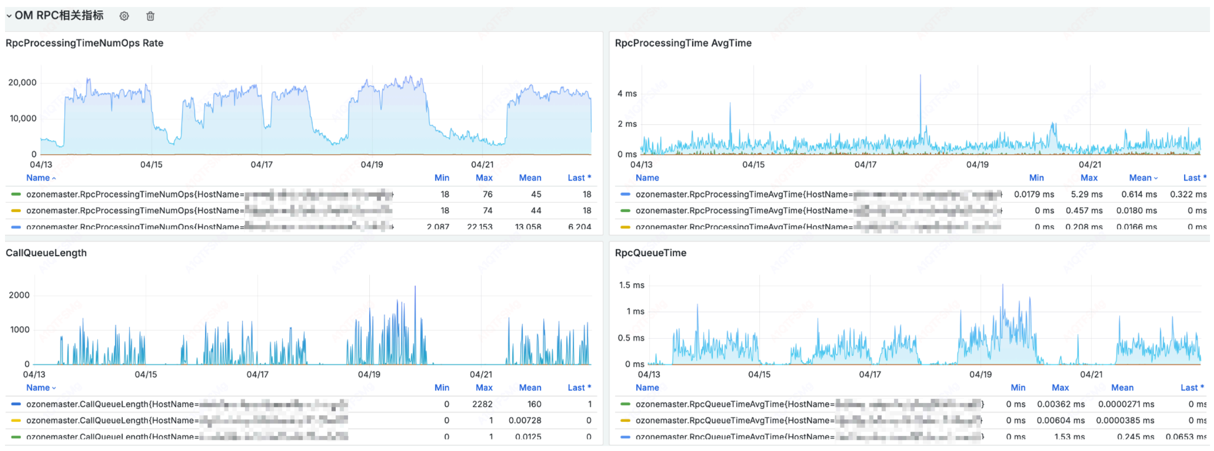
Service monitoring is a key measure to ensure system performance and stability. We continuously track and monitor critical metrics such as OM, SCM, and DN performance to ensure the system remains efficient and stable under high load conditions.

5.1.1 OM/SCM RPC Monitoring

The RPC communication between the client and OM, and between OM and SCM, is a critical component of the Ozone system. Both OM and SCM RPC can become potential bottlenecks. RPC backlogs may block request transmission between these components, leading to degraded service performance or even system crashes.

We have implemented focused monitoring on key metrics related to OM/SCM RPC, primarily covering the following aspects:

- **RPC Request Queue Length**: Monitors the backlog of requests in the queue. If the queue becomes heavily congested, actions such as load balancing or resource scheduling may be required.
- **RPC Response Time**: Tracks the response times of OM and SCM RPCs to ensure that requests are processed within an acceptable time frame. Prolonged response times may lead to request accumulation and impact the execution of other operations.



Name	Min	Max	Mean	Last *
ozonemaster.RpcProcessingTimeNumOps{HostName=[redacted]	18	76	45	18
ozonemaster.RpcProcessingTimeNumOps{HostName=[redacted]	18	74	44	18
ozonemaster.RpcProcessingTimeNumOps{HostName=[redacted]	2,087	22,153	13,058	6,204

Name	Min	Max	Mean	Last *
ozonemaster.RpcProcessingTimeAvgTime{HostName=[redacted]	0.0179 ms	5.29 ms	0.614 ms	0.322 ms
ozonemaster.RpcProcessingTimeAvgTime{HostName=[redacted]	0 ms	0.457 ms	0.0180 ms	0 ms
ozonemaster.RpcProcessingTimeAvgTime{HostName=[redacted]	0 ms	0.208 ms	0.0166 ms	0 ms

Name	Min	Max	Mean	Last *
ozonemaster.CallQueueLength{HostName=[redacted]	0	2282	160	1
ozonemaster.CallQueueLength{HostName=[redacted]	0	1	0.00728	0
ozonemaster.CallQueueLength{HostName=[redacted]	0	1	0.0125	0

Name	Min	Max	Mean	Last *
ozonemaster.RpcQueueTimeAvgTime{HostName=[redacted]}	0 ms	0.00362 ms	0.0000271 ms	0 ms
ozonemaster.RpcQueueTimeAvgTime{HostName=[redacted]}	0 ms	0.00604 ms	0.0000385 ms	0 ms
ozonemaster.RpcQueueTimeAvgTime{HostName=[redacted]}	0 ms	1.53 ms	0.245 ms	0.0653 ms

Figure 18. OM RPC Metrics

5.1.2 OM/SCM Resource Usage Monitoring

OM/SCM, as the metadata node, serves as the control center of the entire Ozone system. If the load on the metadata servers is too high, such as excessive CPU usage, it can lead to slower request processing, increased response times, and even resource exhaustion, causing service unavailability. We monitor the CPU, memory, and disk capacity usage on the master nodes, allowing us to detect and intervene promptly in case of any abnormalities.

For example, we focus on the following key metrics:

- **CPU Utilization**: Real-time monitoring of the OM process's CPU usage to prevent excessive consumption of CPU resources. We have set alert thresholds so that when CPU usage approaches the limit, the system automatically triggers alarms to notify administrators to take action.
- **Per-Core Load Distribution**: Monitors the load distribution across each CPU core to ensure balanced usage and avoid overloading any individual core, which could lead to inefficiencies. Through load balancing and scheduling strategies, we can optimize resource utilization and improve overall system performance.

5.1.3 DN Performance Monitoring

The performance of DN directly impacts data storage and access efficiency. We monitor key performance indicators for DN, primarily including:

- **Disk I/O Performance:** Monitoring the read/write rate and response time of disks to ensure that DN's performance does not degrade due to I/O bottlenecks when processing data.
- **Network Bandwidth Usage**: DN's data transfer speed is closely related to network bandwidth. Bandwidth bottlenecks may cause slow data transfer, affecting the overall performance of the system.
- **Memory Usage**: Monitoring DN's memory usage to avoid memory overflow or excessive consumption, which could lead to system instability.
- **Request Handling Capacity**: Monitoring DN's ability to handle requests, including metrics such as requests per second and average response time. When request handling capacity decreases, it may be necessary to analyze the bottleneck and take corresponding optimization measures.

5.2 JDK 17 Upgrade

To improve system performance and stability, we upgraded to JDK 17. This version introduces a range of enhancements, including improved garbage collection and more efficient compiler optimizations, which collectively contribute to faster response times and better resource utilization.

5.3 Ratis Optimizations

We noticed that DataNodes frequently encountered FullGC issues. The root cause was traced to unreleased off-heap memory. After reviewing community improvements, we backported

RATIS-2065: *Avoid the out-of-heap memory OOM phenomenon of frequent creation and deletion of Raft group scenarios*. This effectively resolved the issue.

5.4 OM HA Stability Optimization

During our early use of Ozone version 1.3, a critical issue was identified where the OM in HA mode would crash and fail to restart. The root cause was a timing inconsistency in write operations under multi-threaded conditions, which resulted in discrepancies between two key internal data structures: applyTransactionMap and double-buffer. This inconsistency subsequently caused failures in the validation logic. Through thorough analysis and with support from the community, the problem was successfully addressed, as documented in HDDS-9342. Since the resolution of this issue, OM HA has demonstrated stable and reliable performance in production.

5.5 Differentiated High-Density storage Practices

We implemented differentiated high-density deployment in storage by creating a high-density cluster. This cluster uses machines with 60 HDDs, each storing about 1PB, which we use to store cold data.

5.6 Exploring Rocksdb Size and File Count

OM maintains the organizational structure of file paths, with the corresponding metadata stored in RocksDB on NVMe SSDs. The capacity of the NVMe SSDs, along with the service's memory configuration, directly affects the scale of metadata that can be stored within a single cluster. Based on our experience, every 100GB of RocksDB size can accommodate approximately 1 billion files. The following data reflects some of our practical observations.

Component	Heap Size (Memory)	SSD Size	om.db Size	File Count
OM	128G	> 1T	400G ~ 500G	≈ 5000 million

Summary

In our practice with Ozone, we have continuously explored and addressed various performance bottlenecks in large-scale data storage. Through a series of optimization measures, we have significantly improved the system's stability and efficiency. Looking ahead, we will continue to optimize I/O performance, with plans to introduce IO_URING technology to enhance read and write efficiency. Additionally, we intend to adopt SPDK technology to further accelerate metadata access performance in OM and SCM, both of which rely on RocksDB. We would also like to express our sincere gratitude to the Apache Ozone open-source community. It is through the community's open discussions, experience sharing, and excellent code contributions that we have greatly benefited. We firmly believe that the development of Apache Ozone will continue to thrive and reach new heights.