Inside an Apache Ozone Upgrade

Ethan Rose
Senior Engineer @ Cloudera
Agenda

- Introduction to Ozone
- How an Ozone upgrade works
- Ozone’s upgrade compatibility guarantees
- Managing disk compatibility
- Managing client compatibility
Introduction to Ozone

● Distributed storage system
● Strongly consistent
● Supports HDFS and S3 protocols
● Improvements from HDFS:
  ○ Metadata stored in RocksDB, only working set is kept in memory
    ■ Scales to billions of objects
  ○ Decoupled namespace and block space
    ■ Handles many small files
Ozone’s Upgrade Guarantees

- **Non-rolling upgrades**
  - All server side components are stopped in the old version, then started in the new version

- **Downgrade support**
  - Also non-rolling
  - Data written in newer version should still be readable

- **Client cross compatibility**
  - All clients released since Ozone 1.0.0 (GA) should work with all server versions released since Ozone 1.0.0
  - An Ozone server may see clients both older and newer than itself
Visualizing an Ozone Upgrade

Data written by 1.2.0 should still be readable by 1.1.0

Client compatibility

Disk compatibility

Client 1.0.0  Client 1.1.0  Client 1.2.0

Ozone 1.1.0
What Affects Compatibility?

- New Ozone features in 2022:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Affects Disk Layout?</th>
<th>Affects Client Protocol?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 RocksDB per Datanode Volume</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>S3 Gateway Persistent Connections</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>S3 Multi-Tenancy</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Bucket Layouts</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Erasure Coding (EC)</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Disk Compatibility

Managing upgrades and downgrades across features with incompatible disk changes
Managing Disk Compatibility

● Finalization
  ○ Concept borrowed from HDFS

● **Pre-Finalized**: New features cannot be used, but downgrade is allowed.

● **Finalized**: New features can be used, but downgrade is not allowed.
Implementing Finalization

- **Layout Feature**: Feature with disk compatibility concerns that is assigned a layout version
- **Software Layout Version (SLV)**: Hard coded layout version
- **Metadata Layout Version (MLV)**: Layout version written to disk

<table>
<thead>
<tr>
<th>Layout Versions</th>
<th>State</th>
<th>New Layout Features Usable?</th>
<th>Downgrade Allowed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLV &lt; SLV</td>
<td>Pre-finalized</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>MLV == SLV</td>
<td>Finalized</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>MLV &gt; SLV</td>
<td>Error, startup fails</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Ozone Upgrade/Downgrade Flow

Start
Ozone 1.1.0
MLV = 0
SLV = 0

Upgrade
Pre-finalized
Ozone 1.2.0
MLV = 0
SLV = 3

Finalization
Finalized
Ozone 1.2.0
MLV = 3
SLV = 3

Downgrade
Representing Layout Features

```java
public enum OMLayoutFeature implements LayoutFeature {
    INITIAL_VERSION(0, "Initial Layout Version"),
    ERASURE_CODED_STORAGE_SUPPORT(1, "Ozone version with built in support for" + " Erasure Coded block data storage."),
    BUCKET_LAYOUT_SUPPORT(2, "Ozone version supporting bucket " + "layouts and introducing the FILE_SYSTEM_OPTIMIZED and OBJECT_STORE " + "bucket layout types."),
    MULTITENANCY_SCHEMA(3, "Multi-Tenancy Schema");
}
```
Developer Requirements

 ➔ Block incompatible requests while pre-finalized

Without a dedicated framework...

✗ No clear guidance for devs to handle upgrade concerns
✗ Every feature’s upgrade concerns are mixed with unrelated code
✗ Difficult to locate code that handles upgrade concerns
✗ Higher chance of error if devs are not familiar with upgrade/downgrade process

➔ Separation of Concerns: Separate compatibility handling from other business logic
Blocking a New Request While Pre-Finalized

@DisallowedUntilLayoutVersion(MULTITENANCY_SCHEMA)
public OMRequest preExecute(OzoneManager ozoneManager) throws IOException {
    ...
}

✓ Easier for devs to use
✓ Upgrade logic separated from business logic
Blocking a New Variation of an Existing Request While Pre-Finalized

Example: Only block bucket creation with EC replication type

```java
@RequestFeatureValidator(
    conditions = ValidationCondition.CLUSTER_NEEDS_FINALIZATION,
    processingPhase = RequestProcessingPhase.PRE_PROCESS,
    requestType = Type.CreateBucket)

public static OMRequest disallowCreateBucketWithECReplicationConfig(OMRequest req, ValidationContext ctx) throws OMException {
    ...
}
```

✓ Easier for devs to use
✓ Upgrade logic separated from business logic
Client Compatibility

Managing client/server protocol changes while maintaining cross compatibility
Client Compatibility Requirements

- Cross compatibility between all clients and servers since Ozone 1.0.0 (GA)
- Client/server protocol should function at the level of the oldest component
- Incompatible requests fail cleanly
Client Compatibility Implementation

- Each component has:
  - Its own version
  - The latest versions of other components it knows about
- Components can choose actions based on the versions they receive.
- The newest component must handle compatibility concerns
Client and Server are the Same Version

```
$ ozone sh bucket create vol/ecbucket
   --type EC --replication rs-3-2-1024k
```
Client is Newer Than Server

- **Client** compares the server’s version with its known server versions
- **Client** should block user requests the server will not understand
- Prevents unnecessary requests resulting in messy protocol errors being returned to the user
public enum OzoneManagerVersion implements ComponentVersion {
    DEFAULT_VERSION(0, "Initial version"),
    S3G_PERSISTENT_CONNECTIONS(1,
        "New S3G persistent connection support is present in OM.")",
    ERASURE_CODED_STORAGE_SUPPORT(2, "OzoneManager version that supports
        + "ECReplicationConfig"),
    FUTURE_VERSION(-1, "Used internally in the client when the server side is
        + " newer and an unknown server version has arrived to the client.");

    //////////////////////////////////////////////////////////////////////////////////
...}

public enum OzoneManagerVersion implements ComponentVersion {
    DEFAULT_VERSION(0, "Initial version"),
    FUTURE_VERSION(-1, "Used internally in the client when the server side is
        + " newer and an unknown server version has arrived to the client.");

    //////////////////////////////////////////////////////////////////////////////////
...}
$ ozone sh bucket create vol/ecbucket
   --type EC --replication rs-3-2-1024k

Can not set the default replication of the bucket to Erasure Coded replication, as OzoneManager does not support Erasure Coded replication.
Server is Newer than Client

- **Server** compares the client’s version with its known client versions
- **Server** should block the client from reading data it cannot understand
  - This could happen if the data was written by a new client using a new format like erasure coding
- Prevents unnecessary requests resulting in messy protocol errors being returned to the user
public enum ClientVersion implements ComponentVersion {
  DEFAULT_VERSION(0, "Initial version"),
  VERSION_HANDLES_UNKNOWN_DN_PORTS(1,
    "Client version that handles the REPLICATION port in DatanodeDetails."),
  ERASURE_CODING_SUPPORT(2, "This client version has support for Erasure Coding."),
  BUCKET_LAYOUT_SUPPORT(3, "This client version has support for Object Store and File System Optimized Bucket Layouts."),
  FUTURE_VERSION(-1, "Used internally when the server side is older and an unknown client version has arrived from the client.");
}

Server is Newer than Client

Key is a key with Erasure Coded replication, which the client can not understand. Please upgrade the client before trying to read the key: vol/ecbucket/k1

```
$ ozone sh key get
vol/ecbucket/k1 /tmp/k1
```

---

$ ozone sh key get
vol/ecbucket/k1 /tmp/k1

Get service info

server version = 2

Read key k1

Client version = 1

Error

k1 is erasure coded. Client version must be $\geq$ 2 for EC support
Where should the server block a read request?

1. Server receives the request to read a key
2. Server retrieves the metadata
3. Server uses the metadata to determine client compatibility

∴ Server must block the read after processing the request

@RequestFeatureValidator(
    conditions = ValidationCondition.OLDER_CLIENT_REQUESTS,
    processingPhase = RequestProcessingPhase.POST_PROCESS,
    requestType = Type.LookupKey
)

public static OMResponse disallowLookupKeyResponseWithECReplicationConfig(
    OMRequest req, OMResponse resp, ValidationContext ctx)
throws ServiceException {
...}
Summary

- Ozone supports non-rolling upgrades and downgrades. This requires:
  - Disk compatibility for downgrades
  - Client/server cross compatibility
- Downgrade disk compatibility is supported using finalization
- Client/server cross compatibility is supported as much as possible, but some specific requests must be blocked
- Ozone uses a developer friendly framework to handle compatibility of new features
  - Feature devs can easily plug in to the upgrade and request flows as needed
  - Separation of concerns and standardized implementations improve readability
Future Work

- Extend client compatibility annotations to other components
  - Currently only implemented on OM
  - Refactor client, SCM, datanode to use annotations as well
- More robust request annotations
  - Handle client/server and layout version checking in the annotations
- Monitor how the framework meets Ozone’s evolving needs
Special Thanks

Special thanks to Aravindan Vijayan, Istvan Fajth, and everyone else involved in Ozone’s upgrade compatibility efforts in the last two years.
Thanks for Attending

Q&A

erose@apache.org | erose@cloudera.com
Github: errose28