Strong Data Integrity Checks in Erasure Coding (EC) Storage Systems

Uma Maheswara Rao Gangumalla
Sr. Engineering Manager - Storage, Cloudera Inc

Aswin Shakil Balasubramanian
Software Engineer II, Cloudera Inc

Swaminathan Balachandran
Senior Software Engineer - Storage, Cloudera Inc
Speakers Intro

**Uma Maheswara Rao G**
- Sr. Engineering Manager at Cloudera
- ASF Member. A Committer and PMC member on Apache Hadoop, Ozone, Incubator, Ratis…etc
- Mentor @ Incubator
- ApacheCon/CoC Track Chair - 2021, 2022, 2023

**Aswin Shakil Balasubramanian**
- Software Engineer II at Cloudera
- Apache Ozone Contributor
- Apache Ozone Project Committer

**Swaminathan Balachandran**
- Senior Software Engineer at Cloudera
- Apache Ozone Contributor
- Apache Ozone Project Committer
Agenda

- Quick Overview - How EC Works
  - Write
  - Read
  - Offline Recovery
- Reconstruction time corruption possibilities
  - Detection techniques & challenges
- Proposed solution
- Question and Answers
Terminology and Implementation Considerations

- Block Group: a block presents in a container group.
- Each data(d) + parity(p) number of chunks written to block group to finish ‘Stripe’.
- Parity generated at the client.
Placement policies pick the required number of nodes for EC.

Nodes set will be chosen based on EC schema (3(d):2(p), 6:3, 10:4).

After finish writing \( d^* \) block_size data, client requests masters to get new node set.
Write: Striping

- **Stripe**: One round of data + parity chunks called as full stripe.
- **Chunks** would be written in round robin fashion to data nodes.
- **Parity Generation**: After every data number of chunks written, parity will be generated and send to remaining nodes in group.
- If stripe write fails, the current block group will be closed and rewrite the failed stripe to new block group.
- **Client** keep track of bytes written and check for failures.
Write: Partial Stripe with Padding

Client uses padding data for generating parity chunks if stripe is not full.

**Partial Stripe:**
- chunk2 and chunk 3 assumed as padding data and len is 1MB.
- chunk3 assumed as padding data and len is 1MB.
- Full Stripe: No padding needed.
Write: Striping

- If stripe write fails, the current block group will be closed
  - Failed stripe will be rewritten to the new block group.
- Client keep track of bytes written and check for failures.
- After all data writes finishes, then parity writes.
Read

- Reads in the same order in which order writes done. Order will be based on replica indexes.

- Client stitches the data back to original order and serves to user.
Reconstructional Reads

➢ First read will attempt to read data blocks.

➢ When node failed while reading, client will switch to reconstructional read and read from parity and reconstruct the lost data transparently.

➢ Reconstruction read will have overhead due to ec decode operation.

➢ To avoid the degraded reads, we need to recover the lost replicas offline.
Recovery Reads

EC Block Group : 1

N1

N2

N3

N4

N5

Reconstruct and Read stripe-1

Reconstruct and Read stripe-2

Read File

EC Client

1MB - chunk1
1MB - chunk2
1MB - chunk3
1MB - chunk4
1MB - chunk5
1MB - chunk6

blockGrpID:1

1MB - chunk1
1MB - chunk2
1MB - chunk3
1MB - chunk4
1MB - chunk5
1MB - chunk6

1MB parity1 (c1, c2, c3)
1MB parity2 (c4, c5, c6)

1MB parity1 (c4, c5, c6)
1MB parity2 (c4, c5, c6)
Offline Recovery

What is the Offline Recovery?

➢ When a node/Disk lost, we will lose the containers which are residing in that node/disk.
➢ We need a mechanism to recover that lost containers in the background.
➢ We call this process of background recovery as “Offline Recovery”.
➢ This is very critical background task similar re-replication on node/disk failures.
Offline Recovery

ReconstructECContainersCommand:
1. containerID
2. Source replicas: DN1, DN2, DN3
3. Target DNs - DN6, DN7
4. Missing indexes
5. ECReplicationConfig

- Find Blocks
- List
- Loop for recover Block for all Block
- Recover Block
- EC decode and recover lost index chunks
- Write chunk to target container
- Transfer chunk to the target container
- Close the containers in target DNs

Target DNs sending ICR

Container Recovery Done
Reconstruction Time Corruption Possibilities
Offline Recovery

ReconstructECContainersCommand
1. containerID
2. Source replicas: DN1, DN2, DN3
3. Target DNs - DN6, DN7
4. Missing indexes
5. ECReplicationConfig

Target DNs sending ICR

Wrongly generated data leads to wrong checksums here
Offline Recovery

ReconstructECContainersCommand
1. containerID
2. Source replicas: DN1, DN2, DN3
3. Target DNs - DN6, DN7
4. Missing indexes
5. ECReplicationConfig

1. containerID
2. Source replicas: DN1, DN2, DN3
3. Target DNs - DN6, DN7
4. Missing indexes
5. ECReplicationConfig

Create recovering state containers in target DNs
List blocks
Find Blocks
List
Loop for recover Block for all Block
Recover Block

How do we detect?

EC decode and recover lost index chunks
Write chunk to target container
Transfer chunk to the target container
Close the containers in target DNs

Target DNs sending ICR

Container Recovery Done
Corruption Detection & Challenges

Here is one example corruption situation we dealt with.

- Some of blocks content turned out with all zeros.
  - Found the issue when compared with source data checksums.

- Need to find the blocks with all zeros as evidence of corruption:
  https://github.com/umamaheswararao/ec-corruption-analyzer

- Need additional confirmation to make sure corruption is real:
  https://github.com/sodonnel/hdfs-ec-validator

- Fixing of such corruption is hard, need to delete the blocks which are impacted in a block group.

- System will fix with right content automatically.

- If impacted blocks are more than d number of blocks, it is not possible to fix automatically unless we have source data available.
Proposed Solution
Solution (WIP)

➢ Checksum bytes calculated for full stripe at client.
➢ Checksum bytes are splittable at chunk boundaries. Meaning when checksum bytes are 4 bytes for chunk, full stripe checksum bytes are 5 * 4 bytes (Note: in the interest of space picture only showing 2 bits checksum length for chunk)
So, we can find specific chunk position checksum bytes from full stripe checksum bytes.
➢ We can store these checksum bytes only at P + 1 nodes.
➢ Why P+1 nodes to store checksums?
  ○ At any point of time, reconstruction needs d nodes.
  ○ So, at least one node of P+1 will participate in reconstruction.
Pros

➢ getCheckSum API can make sure of these precalculated stripe checksum
➢ When node lost, recovery node can use these checksum bytes to validate

Cons

➢ Little(negligible) overhead of storing these addition stripe checksum bytes at DNs
Recovery Time Validation

N2, N3, N5 are used for reconstruction.

On recovery of chunks, recovery node can use N5’s original stripe checksum to validate with newly generated chunk checksum.

If mathes, chunk reconstruction is successful.

If does not matches, reconstruction is wrong and recovery would fail.
EC supported in a scalable Object Storage: Apache Ozone

- Github repo: https://github.com/apache/ozone
- Looking to contribute to the Apache Ozone EC project?
  - Start with https://github.com/apache/ozone/blob/master/CONTRIBUTING.md
- Bug reporting is at: https://issues.apache.org/jira/projects/HDDS
Want to join a Storage Meetup?

➢ There is a storage meetup happening in Santa Clara on Oct 25.
➢ Scan the below QR code for details
Thanks

Q&A

umamahesh@apache.org
aswin@apache.com
sbalachandran@cloudera.com