HDFS Erasure Coding

v2015.2.6, ~~v2015.2.4~~

Problem Statement

Currently HDFS triplicates each block by default for a number of purposes: 1) protection against *DataNode* failures; 2) better locality for MapReduce tasks; 3) avoidance of overloaded *DataNodes* through choosing among multiple replicas. Replication is **expensive** -- the default triplication scheme has **200%** overhead in storage space and other resources (e.g., *NameNode* memory usage). However, for "warm" datasets with low I/O activities, secondary block replicas are rarely accessed during normal operation -- while consuming the same amount of resources as the primary ones. Therefore, a natural improvement is to use Erasure Coding (EC) in place of replication, which provides the same level of fault tolerance with much less storage space. In typical EC setups the storage overhead is **≤ 50%**.

HDFS archival storage (HDFS-6584) shares a similar motivation. It is designed for clusters provisioned with well-balanced primary and archival storage resources. EC optimizes homogenous storage architectures as well. Moreover, EC can be used on top of archival storage to save storage resources on multiple tiers.

Facebook's solution is to build a RAID layer *on top* of HDFS [Fan09, Sathiamoorthy13]. This eases development but limits its interactions with the HDFS kernel. As a result, it stores parity data as *user-visible files* instead of blocks, which are prone to misoperations. Moreover, it is not a self-contained HDFS feature, with dependencies on MySQL to store metadata and MapReduce to create parity files. The RAID node periodically queries for corrupt files and increases *NameNode* workload.

Use Cases

Use Case: Saving space with sealed / warm data

In Google File System (GFS) and Windows Azure Storage (WAS), datasets are initially triplicated. After a file or block is determined "sealed" (no longer modified), a background task encodes it and then deletes its secondary replicas [Ford10, Huang12]. The following scenarios need to be considered to support this use case under the specific architecture, features, and workloads of HDFS:

Skewed data temperatures

HDFS hosts a wide variety of workloads: from batch-oriented MapReduce jobs to interactive, latency-sensitive queries in Impala and HBase. Therefore we should allow users and admins to specify (or hint at) the "temperature" of a file. Hot datasets will remain replicated even after

they are sealed. This way a reading client can choose from multiple replicas and avoid overloaded *DataNodes*.

Skewed file sizes

One important assumption in the original GFS design was to handle *big files*. This is becoming less valid as big data workloads evolve. Applications like MapReduce and HBase generate a lot of small and medium files [Harter14]. Some files only consist of a few blocks, and some others are smaller than a single block. HDFS-EC should be versatile to handle all reasonable types of file size distributions.

Use Case: High availability and durability

Under the current replication scheme, HDFS block errors are discovered and repaired both online and offline. Erasure-coded blocks should have the same level of protection.

Passive recovery (on the read path)

When a replicated block is read and its primary replica is unavailable, a secondary replica is used to serve the request. The block is also re-replicated in the background. Each erasure-coded block has only one replica. When that only replica is found unavailable during a read request, the EC mechanism should be able to reconstruct the lost block while continue serving the read request without significant delay. Recovery latency is critical to high performance query engines like Impala. It is not acceptable to reconstruct the entire lost block before serving the read request.

Active recovery (in the background)

The HDFS *NameNode* actively discovers lost or corrupt blocks from periodic *block reports*. A replicated block is repaired by copying a healthy replica to a new *DataNode*, so the block is back to “full strength”. Under EC, original blocks should be actively diagnosed to ensure they are ready for I/O; parity blocks should be treated similarly to maintain the “strength” of fault tolerance.

Use Case: Dynamic data access pattern

A file’s access pattern can change on-the-fly. When a cold dataset becomes hot, HDFS should recreate its secondary replicas for faster I/O accesses.

Use Case: Saving I/O bandwidth on the write path

When applied directly on the write path, EC saves network and disk bandwidth by reducing the total amount of data written from the client to the storage servers. QFS [Ovsiannikov13] uses this *online* approach. Note that EC uses more I/O bandwidth than replication in case of block reconstruction.

Use Case: Leveraging multiple spindles

Network bandwidth grows at a much higher rate than hard disk, and data center storage is becoming flat [Nightingale12]. When EC uses a small codec unit (e.g., 64KB as in QFS), a single read/write request from the client can cross multiple storage servers and therefore leverage the aggregate bandwidth of their disk spindles.

Use Case: Geo-distributed disaster recovery

Geo-replication helps datasets survive severe failures that could render an entire datacenter unavailable (HDFS-5442). With this greater fault tolerance comes doubled storage overhead. Under the default replication factor, **6 replicas** of the same block need to be stored. Applying EC in this context has a great potential of saving storage space and WAN bandwidth. Facebook's f4 BLOB storage system uses XOR across 3 sites to achieve this goal [Muralidhar14].

Goals

Substantial space saving

Based on customer requirements we have collected, space saving is by far the most important use case. In WAS, EC has a target storage overhead of 33%, compared to 200% in default 3-way replication. This level of saving is easily achievable with large files and a uniform policy of always encoding sealed data. Our design should efficiently handle hot data and small files so that EC still saves significantly in storage space under most of our workloads.

Flexible policies

Users and admins should be able to tag files as hot or cold and these hints should be reflected in EC actions. This should be integrated or coordinated with storage policies in HDFS archival storage work. The current storage policies only define the desired storage types for a block's replicas. EC needs to extend storage policies to define and enforce the *distribution pattern*.

Moreover, certain EC tasks should be marked as high priority / urgency to support freeing up space when the usage is approaching the quota.

Fast recovery / conversion

We should apply local reconstruction codes (LRC) [Huang12] or its equivalents to reduce network traffic required in recovery. State-of-the-art codec techniques should also be pluggable, such as Jerasure, Intel ISA library, and the optimization in [Khan12].

Saving I/O bandwidth on the write path

Under I/O intensive workloads, EC potentially saves bandwidth usage by reducing the sheer amount of data written to the system.

Low overhead

EC inevitably introduces overhead on the *NameNode* to keep track of parity blocks. We should try to minimize this overhead.

Transparency / compatibility

HDFS users should be able to use all basic or advanced features on erasure-coded data, including caching, snapshots, encryption, appending, truncation, and so forth.

Non-Goals

The following features map to low priority use cases. Considering their development costs, they are not included in the scope of this current stage:

- Geo-distributed EC
    - We should revisit this topic when geo-replication (HDFS-5442) is finalized

Technical Approach

Design Decisions and Architecture

EC and Block Layout

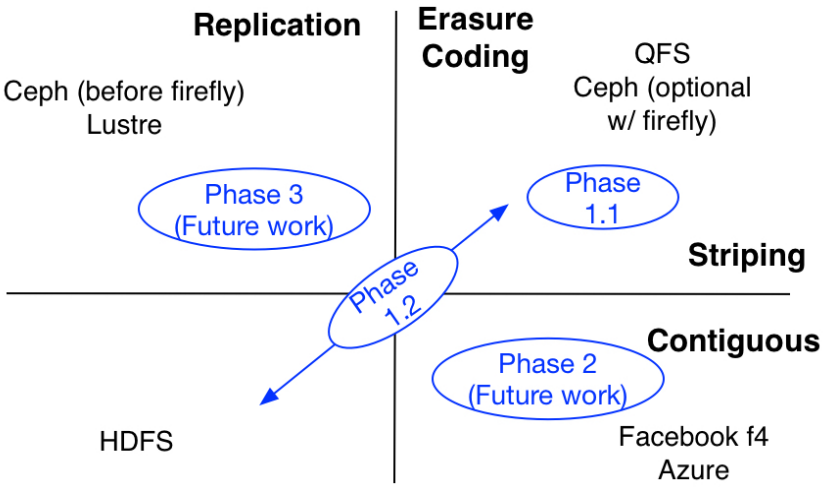


Figure 1: EC and block layout

Currently HDFS stores each block replica *contiguously* on a single *DataNode*. Another widely used layout is to *stripe* data across multiple *DataNodes*, examples of which include QFS,

Colossus, Ceph, and Lustre. In principle, block layout (contiguous vs. striping) and redundancy form (replication vs. erasure coding) are 2 orthogonal dimensions, resulting in 4 possible combinations, as shown in Figure 1.

The pros and cons of EC have been discussed earlier in this document. On the other dimension, striping greatly enhances sequential I/O performance by leveraging multiple disks in parallel; data locality is lost as a cost. In the context of EC, striping has several critical advantages. First, it enables online EC which bypasses the conversion phase and immediately saves storage space; this is especially desirable in clusters with high end networking. Second, it naturally distributes a small file to multiple *DataNodes* and eliminates the need to bundle multiple files into a single coding group. This greatly simplifies file operations such as deletion, quota reporting, and migration between federated namespaces.

Considering these tradeoffs, the objective of the current phase is to support **EC with striping** (Phase 1.1), as well as the conversion **between *{EC+Striping}* and *{Replication+Contiguous}*** forms (Phase 1.2). It leads to the following design decisions:

1. Encoding is done both *online* and *offline*. In offline encoding, blocks are still replicated initially, and converted into erasure-coded form when designated conditions are met.
2. Codec processing is done by both the client and *DataNode*. The client calculates parity data for the initial online encoding, and reconstructs lost original data during read requests. The *DataNode* builds and stores coded blocks in proactive/background recovery, as well as converts from replicated to EC forms.
3. Only *finalized* files are eligible for conversion. Appends are not supported during conversions.

System Architecture

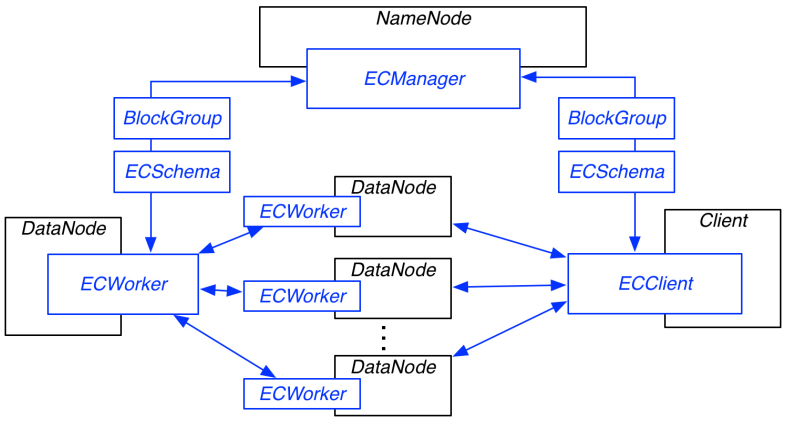


Figure 2: Overall architecture

Figure 2 illustrates the overall architecture of the proposed design. Added components and logics are in blue. **`ECClient`** denotes the extension to the HDFS client which stripes data to and from multiple *DataNodes*. The **`ECManager`** resides on the *NameNode* and manages EC block groups, whose responsibilities include group allocation, placement, health monitoring, and coordination of recovery work.

DataNodes are unaware of EC or striping during normal I/O operations. The added **`ECWorker`** daemon thread listens for *recovery* or *conversion* commands from **`ECManager`**. It serves these requests by pulling data from peer *DataNodes*, carrying out codec calculation, building recovered or converted blocks, and possibly pushing to additional **`ECWorker`** (if multiple blocks are to be recovered or converted). In each operation **`ECManager`** informs **`ECWorker`** of the group of blocks and the name of the codec schema to be used (e.g., create a parity block from raw blocks {A, B, C} with Reed-Solomon encoding). For simplicity this design document focuses on the recovery functionality of **`ECWorker`**; its role in conversion will be discussed in a subsequent design dedicating to Phase 1.2 of the project.

User Interface

Several styles exist for HDFS users and admins to control the policies and preferences in storing a given path. The rest of this section analyzes and compares different options in the context of EC.

Storage Policy

Storage types and policies have been introduced into HDFS as a result of heterogenous and archival storage supports. Based on this framework, a new storage policy (e.g., **`EC`**) can be defined. Blocks under this policy will be erasure-coded; in the initial phase, both raw and parity data will be stored on the DISK storage type.

Then, existing APIs for getting and setting storage policies can be used. For example, the following command tells HDFS to erasure code all blocks belonging to files under **`/ec-dir`**.

```
hdfs dfsadmin -setStoragePolicy /ec-dir EC
```

The storage policy is stored in the inode header, as implemented in HDFS-6584. As shown in Figure 1, Phase 1.1 of the project focus on creating files in the {EC+Striping} form. Therefore, a directory's EC storage policy needs to be configured while it's empty, and will be fixed once configured -- similar to encryption zones.

In Phase 1.2, conversion between erasure-coded and replicated forms can be done by changing the storage policy. The new policy can be enforced by triggering the **`Mover`** or directly by **`ECManager`**. More details will be discussed in a follow-on Phase 1.2 design.

Under this framework, users should also be able to configure the codec schema for a directory. In particular, the interface should support specifying the codec algorithm (e.g., Reed-Solomon vs. XOR) and block group layout (e.g., 3-of-6 vs. 4-of-10).

DFS Commands

Similar to changing the replication factor of a path through **`hdfs dfs setrep`**, new **`dfs`** commands can be added to directly convert a set of files between replicated and erasure-coded forms.

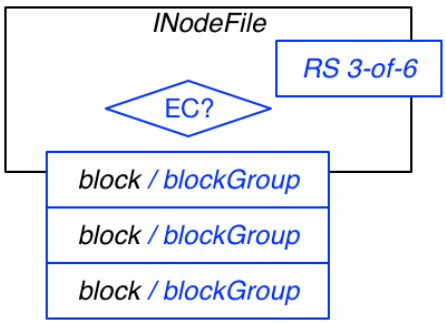
- **`hdfs dfs -convertToEC -path <path> <EC schema>`**: Converts all blocks under this path to EC form (if not already in EC form, and if can be coded).
- **`hdfs dfs -convertToRep -path <path>`**: Converts all blocks under this path to replicated form.

Simple and intuitive, this method is a good fit for "one-off" operations issued by HDFS users. To keep a path's storage policy consistent with its actual storage layout, it can be implemented as a wrapper -- first running **`setStoragePolicy`** and then triggering **`Mover`**. This requires designating 2 default storage policies -- for EC and replicated forms -- to be used in **`setStoragePolicy`**. A more sophisticated scheme is to predefine a mapping between EC and replicated storage policies (e.g., **`HOT`** ↔ **`EC`**), and adding a step of **`getStoragePolicy`** in the beginning to calculate the storage policy to change to.

Protocol and Metadata Extensions

BlockGroup

All erasure codec operations center around the concept of *block group*; they are formed in initial encoding and looked up in recoveries and conversions. A lightweight class **`BlockGroup`** is created to record the original and parity blocks in a coding group. The EC storage policy, as well as the codec schema, can be found in the file inode (file header or extended attributes). With the striping layout, the HDFS client needs to operate on all blocks in a **`BlockGroup`** concurrently. Therefore we propose to extend a file's inode to *switch between contiguous and striping modes*. A strip file has a list of **`BlockGroups`**.



BlockGroup neither has an ID nor a generation stamp. It is possible to use consecutive IDs and use the same generation stamp for all the blocks in a BlockGroup, as detailed in section NameNode Memory Usage Reduction. This allows a simplification to represent the entire **`BlockGroup`** with its first **`Block`**, and reuse existing **`blockmanagement`** code to handle block groups. Only the ID of the first block and the shared generation stamp are stored in a BlockGroup. To provide a clear conceptual discussion, the remainder of the document uses **`BlockGroup`** to denote a logical EC block group, abstracted from the actual implementation.

ErasureCodec

Codec algorithms should be pluggable, supported by an **`ErasureCodec`** interface. A set of implementations of this interface should be provided to connect to popular libraries such as

Intel's ISAL (Intelligent Storage Acceleration Library). The following 2 key EC parameters should also be configurable. Full details of erasure codec design can be found under HDFS-7337. Let

- *M* be number of original blocks in a striping group, and
- *K* be number of parity blocks in a striping group.

Then, a (M,K)-codec can tolerate K concurrent block failures. M is also minimum number of required blocks for accessing original data and recovering the unavailable blocks. The ratio of K/M is the relative storage overhead. An example of a (M,K)-codec is (6,3)-Reed-Solomon, which is chosen as the default codec.

Additional **`Block`** states might need be added. For example, when EC with *contiguous* layout is enabled in Phase 2, each **`Block`** should have a binary flag denoting whether it is a parity block (**`isParity`**). Contiguous parity blocks are created, stored, and reported the same way as raw ones. They have regular block IDs which are unrelated to those of the raw blocks in the same group; their replicas (normally only 1) are stored in the RBW and finalized directories on the *DataNode* depending on the stage; they are also included in block reports. The only distinction of a contiguous parity block is the lack of file affiliation.

Client Extensions

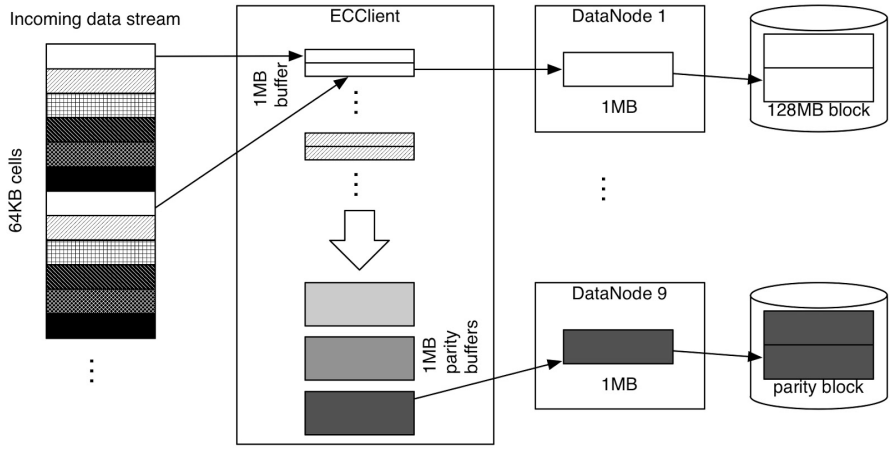


Figure 3: Client extension (revised version of Fig. 2 in [Ovsiannikov13])

ECClient

HDFS client will be extended to write to and read from multiple *DataNodes* in parallel, as illustrated in Figure 3. Note that the current high-level design is based on QFS client, and will be tailored for HDFS in the implementation phase. In particular, when seeking to a position in a file, **`ECClient`** obtains the **`BlockGroup`** rather than block information from the *NameNode*. Let

- *C* be the size of each stripe cell, usually 64kB; and
- *B* be the size of I/O transfer buffer, usually 1MB,

A smaller cell size C improve I/O parallelism but increase overhead. A bigger buffer size B improve I/O efficiency but delay data protection.

When operating on a file, the HDFS client checks its storage policy to determine whether to use **`ECClient`** for striping I/O.

For EC files, hflush/hsync/append can be supported by a similar approach as normal files. For the design of hflush/hsync/append on normal files, see [HDFS-265].

EC Writer

When writing to an EC file, a client writes data strips (e.g. 64kB) to multiple datanodes in parallel. For (6,3)-Reed-Solomon, the client writes data to the first 6 datanodes and parities to the remaining 3 datanodes. Note that there are buffers between datanodes and the client. So when a client writes some bytes to a datanode, the bytes may be first copied to a buffer (e.g. 1MB) and then flushed later on. The write buffer in datanodes is relatively small, usually 4kB. We ignore buffering in rest of the section.

In the beginning, the client write first strip data to the first datanode, second strip data to the second datanode and so on. Once it has the sixth strip data, it computes three parity strips and writes them to the last three datanodes. Similar to the write pipeline (i.e. write pipeline for multiple replications), once a datanode has received a packet (data or parity) from the client, it sends an ack to client. The client writes continuously. It does not wait for the ack from the datanodes except for the last packet.

Clients always send full strips to datanode except the last strip group. At the end of the file, the data length may not be a multiple of the size of a data strip group (which is 6 x 64kB in our example). The client computes parity once it has received a close file request. When the last strip group is partial, the client in addition writes the length of the last strip group at the end. The strip group length is required for decoding as described below.

Handling Datanode Failure during Write

When there are one or more datanode failures (e.g. datanode dies, network problem) during write, the client ignores the failed datanodes, bumps the generation stamps for all the remaining blocks in the block group and then continues writing with the remaining datanodes, provided that the number of remaining datanodes is larger than or equal to the minimum required datanodes. Note that the blocks in a block group share the same generation stamp. The missing blocks are not recovered until the client finishes writing to the block group. The recovery is scheduled by the usual EC reconstruction scheduling in namenode. If the remaining datanodes is less than the minimum required datanodes, the client throws an

exception to indicate write failure. For (6,3)-Reed-Solomon, the minimum required datanodes is 6 and the client can tolerate 3 datanode failures.

Another approach is to close the current BlockGroup and then continues writing to a new BlockGroup using the variable block length feature [HDFS-3689].

Slow Writers and Replace Datanode on Failure

For the HDFS write pipeline (for multiple replications), there is a replace-datanode-on-failure feature so that when a datanode in the write pipeline fails, it may be replaced by a new datanode. The feature is designed for supporting slow writers. Since slow writers are not a use case for EC flies (slow writers should use write pipeline), the replace-datanode-on-failure feature is not supported for EC files.

Reading a Closed File

Since the closed file is closed, the length of it is fixed. Client can read from any 6 out of 9 datanodes. The datanodes with data blocks are more preferable than the datanodes with parity blocks because less EC block reconstruction work is needed.

For the block groups with full data, client reads data/parity blocks from the chosen 6 datanodes and, if necessary, reconstructs the data blocks.

For partial block groups (only possible for the last block group in a file), the client reads data/parity blocks from the chosen 6 datanodes. If there is no parity blocks involved, no EC reconstruction is needed. Otherwise, the client also reads the length of the last strip from the parity datanodes. Client should verify if all the last strip lengths from the parity blocks, the actual data length and the file length obtained from the namenode are matched before reconstructing the EC data blocks.

Reading a Being Written File

When a files is being written, the size of each block in a block group may be different. Client first gets data lengths from the datanodes with data blocks, and parity lengths and last strip lengths from the datanodes with parity blocks. It uses the length information to determine the maximum available length and choose 6 datanodes to read from. Then, the client can read and, if necessary, reconstruct the data as before. It guarantees a newer client can read as much data as the previous client readers.

Hflush

When there is a hflush call, the client flushes all the existing data and parity to all the datanodes. Then the client is blocked and it waits for the latest acks from all datanodes. It returns success once it has received all the acks. At this point, a new reader is able to read

all the data up to the hflush position. When the write is resumed, the client requires to overwrite the last parity strip when the hflush position is not at a full strip group boundary.

Hsync

hsync is similar to hflush except that datanodes in addition issue a local fsync call before sending the ack.

Append

Appending data to an existing BlockGroup requires the number of available datanode larger than or equal to the minimum required datanodes. It starts with bumping the generation stamp of all the blocks in a BlockGroup. Then, the client writes data to all the available datanodes. Similar to hflush, the client requires to overwrite the last parity strip when the last strip group is partial.

Append can also be supported using the variable block length feature [HDFS-3689], i.e. append data to a new BlockGroup but not reopening the last existing BlockGroup.

Truncate

Truncating a EC file is similar to truncating a normal file except that the parities of the last strip may need to be recalculated. Similar to append, it requires the number of available datanode larger than or equal to the minimum required datanodes. It starts with bumping the generation stamp of all the blocks in a BlockGroup. For truncating on strip group boundary, the client truncates all the data blocks and parity blocks, and then it is done. Otherwise, the client reads all the data strips containing the truncate position. If some data blocks are unavailable, the client reads also the required parity strips containing the truncate position in order to reconstruct the missing data strips. Then, compute the new strip group length and the new parity for the truncated data. Finally the client truncates all the data blocks in the datanode, and truncates the parity block and rewrites the last parity strips with the strip group length.

If an upgrade is in progress or truncating a file in a snapshot, the existing data of the last block is first copied in order to support rollback or reading from snapshot.

BlockGroup States

When a BlockGroup is created, it is in the UNDER_CONSTRUCTION state. Once ECClient finishes writing to a BlockGroup, it sends all the remaining R blocks written to the namenode (some blocks may be excluded due to failures). If the R blocks namenode received is greater than or equal to M, where M is the minimum required blocks (e.g. M is 6 for (6,3)-Reed-Solomon), the BlockGroup is transited to COMMITTED state. When the namenode received M block receipts from M datanodes, the BlockGroup is transited to COMPLETE state. When a block group is being recovered, it is UNDER_RECOVERY state.

Generation Stamp

All the data blocks and parity blocks in a BlockGroup share the same generation stamp. BlockGroup itself does not have generation stamp. For memory efficiency, the shared generation stamp may be stored in the BlockGroup object in the implementation.

BlockGroup Recovery

BlockGroup recovery is a procedure to bump the generation stamp of all the currently available blocks such that the available blocks have a newer generation stamp and the generation stamp of the unavailable blocks remains unchanged. In this way, the unavailable blocks can be excluded and removed if they show up in the future. It is used in failure handling, append and truncate.

Client-Datanode connections

For writing a EC file, a client writes to 9 datanodes in parallel. It is 9x compared to existing 3-replica writing pipeline. Reading a EC file reads from 6 datanodes in parallel. It needs 6x connections. It may be a problem for a client to read/write many files at the same time. This problem can be fixed by changing DataTransferProtocol implementation to reuse connection so that only one connection is used from a client to a datanode for multiple concurrent block reads/writes.

NameNode Extensions

Currently, HDFS *NameNode* runs a **`ReplicationMonitor`** daemon to periodically execute block replication and invalidation tasks. Those tasks are inserted into and maintained by **`UnderReplicatedBlocks`** and **`InvalidateBlocks`**, respectively. This mechanism is a natural fit to schedule background block codec tasks, including proactive block recovery and conversion between replication and EC forms.

Figure 4 below illustrates the proposed extension on *NameNode* (mainly in **`BlockManagement`**).

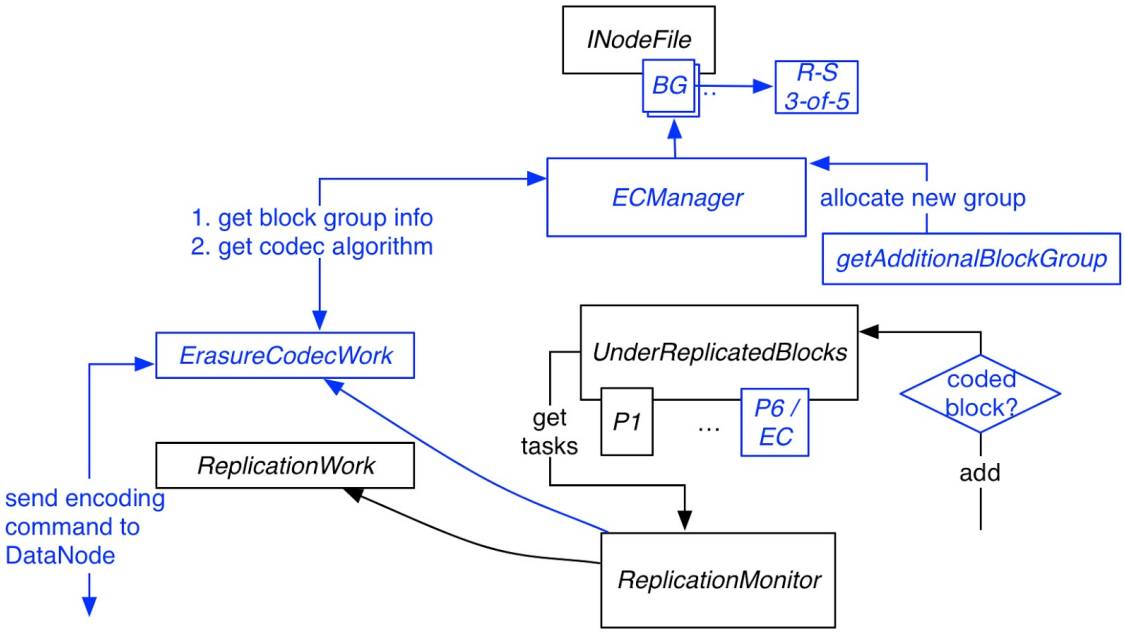


Figure 4: *NameNode* extensions

`ECManager`: This module manages **`BlockGroups`** and associated codec schemas. As a simple example, when a *{Striping+EC}* file is created and written to, it will serve requests from the client to allocate new **`BlockGroups`** and store them under the **`INodeFile`**. In the current phase, **`BlockGroups`** are allocated both in initial online encoding and in the conversion from replication to EC. **`ECManager`** also facilitates the lookup of **`BlockGroup`** information for block recovery work.

EC Block Reconstruction

When one or more EC blocks in a block group is missing, namenode may schedule the block group for EC block reconstruction. For (6,3)-Reed-Solomon, since it can tolerate 3 missing blocks, the 1- or 2- missing block cases are in low priority and the 3 missing block case is in high priority. We use the existing replication scheduling framework in the namenode for EC block reconstruction as described below.

`UnderReplicatedBlocks`: New priority queues (e.g., **`QUEUE_CONVERSION`**) should be added to the 5 existing ones to maintain the relative order of codec and replication tasks.

- When a parity block or a raw block in **`ENCODED`** state is found missing, it is added to **`UnderReplicatedBlocks`**, and the priority is determined by the condition of its group. E.g., if all parity blocks in a group are lost, they should be added to **`QUEUE_HIGHEST_PRIORITY`**. New priorities might be added for fine grained differentiation (e.g., loss of a raw block versus a parity one).
- A new priority, **`QUEUE_CONVERSION`**, is used to convert a replicated block to EC and vice versa. Since such conversions are not related to block availability, the new priority should be lower than existing ones in **`UnderReplicatedBlocks`**.

- Compared to a regular block replication task, encoding or decoding a block consumes much more I/O bandwidth and CPU cycles. Therefore, a system-wide limit should be applied on the total number of block codec tasks.

`ErasureCodecWork`: The **`ReplicationMonitor`** will be extended to distinguish replication and encoding tasks. If a parity block is to be created or an original block in a **`BlockGroup`** is to be repaired, an **`ErasureCodecWork`** task is spawned instead of **`ReplicationWork`**. It does the following:

- Obtains all codec-related information from **`ECManager`**, including the **`BlockGroup`** and associated codec schema. A new block placement algorithm is needed to select the destination *DataNode* to conduct the coding work and host the recovered or converted block. The finalized **`BlockGroup`** -- after excess replicas are deleted -- should be distributed on unique *DataNodes*, and ideally on unique racks, so the failure of any node/rack compromises at most one block in the group. Several interesting optimizations can be applied. The first is to distribute the coding work to multiple *DataNodes* to reduce the CPU load on the destination node. Second, it is possible to place the parity block near (as many as possible) replicas of raw blocks in the group so that encoding can be done partly on local data. These replicas should be deleted as excess replicas to maintain the striped distribution pattern of the group.
- Sends the codec command to the chosen *DataNode*.
- After receiving acknowledgement from the *DataNode* confirming a successful encoding, deletes excess replicas by adding tasks to **`InvalidateBlocks`**.
- When recovering a missing raw block, the chosen *DataNode* should acknowledge **`ErasureCodecWork`** as soon as the decoded block is initiated. Then the block's storage location in **`blocksMap`** should be updated, allowing clients to read partial block data as it is reconstructed.

DataNode Extensions

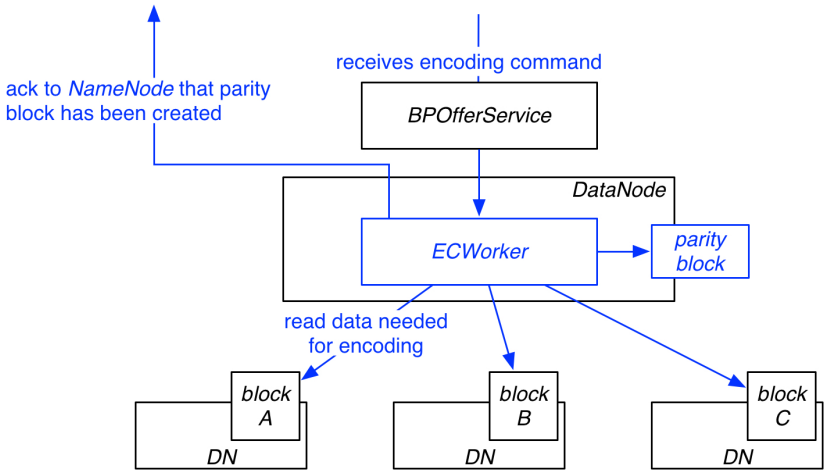


Figure 5: *DataNode* extensions

`ECWorker`: This is the main erasure coding module on *DataNode*. As illustrated above in Figure 5, after receiving a codec command from the *NameNode*, the **`ECWorker`** parses the command and obtains the list of *DataNodes* hosting original blocks in the group -- in the case of repairing an original block, this list should include one or more *DataNodes* hosting parity blocks in the group. It then tries to connect with the **`DataXceiver`** of those nodes and receive data packets to gradually build the parity block (or recover the original block). After the parity block is successfully built (or original block repaired), the **`ECWorker`** sends an acknowledgement to the *NameNode* so excess block replicas in the group can be scheduled for removal.

Optimizations

NameNode Memory Usage Reduction

Striping increases *NameNode* memory usage by introducing more blocks. For instance, HDFS currently stores a 128MB file as 1 block with 3 replicas, resulting in 1 entry in **`blocksMap`**. With striping and Reed-Solomon (6,3) schema, the file uses 9 blocks, each with 1 replica. This can be mitigated by numbering the blocks using consecutive IDs (or other predictable way.) A numbering schemes is provided below, illustrated in Figure 6.

- The 64-bit block ID space is divided into two parts, normal block ID space and EC block ID space. The highest bit indicates whether the block is a EC block (and therefore belongs to a **`BlockGroup`**).
- For EC blocks:
    - The highest bit is 1.
    - Use the lowest 4 bits to mark the order of the block inside the **`BlockGroup`**.
    - The first block in a block group has ID with lowest 4 bits zeros.
    - **`BlockGroup`** does not an ID.
- For normal blocks, the highest bit is 0. IDs are generated using the remaining 63 bits.
- Keep a **`blockGroupsMap`** besides **`blocksMap`** keyed by the first ID of the blocks in a **`BlockGroup`**. When a block is reported by a *DataNode*, it is easy to determine whether it is an encoded block, and if so, its **`BlockGroup`**.

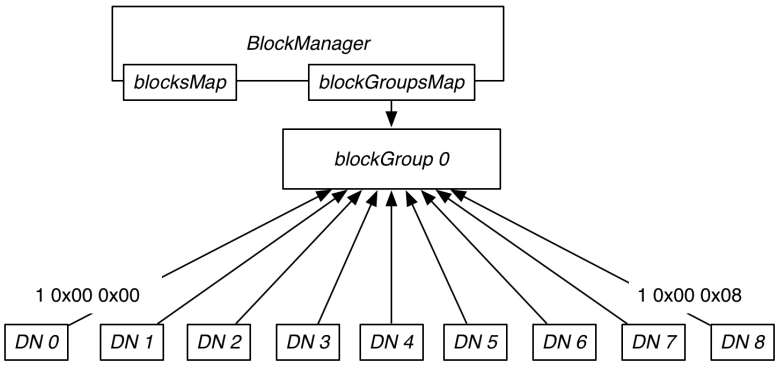


Figure 6: Continuously named blocks

This optimization, together with a few other ongoing efforts, should be able to help avoid memory insufficiency in most production clusters:

1. HDFS-6658 (optimizing block replicas list)
    a. 11% saving in memory usage (without compressed oops)
2. HDFS-7244 (Flyweight pattern)
    a. “In the case of off-heap, …, memory used for BlockInfo would reduce to a very small constant value”
3. Shrinking **`blockId`** from *long* to an array of 6 bytes
    a. This saves 8 - 6 = 2 bytes for every block

Collision with Random Block ID

Since random block IDs were supported by some early version of HDFS, the block ID reserved for EC blocks could be already used by some existing blocks in a cluster. During NameNode startup, it detects if there are reserved EC block IDs used by non-EC blocks. If it is the case, NameNode will do an additional **`blocksMap`** lookup when there is a miss in a **`blockGroupsMap`** lookup.

Datanode Decommission

EC block reconstruction is more expensive than replication because a replica can be copied from any available source datanode for blocks with replication. However, if a EC block is unavailable, it requires to read from 6 datanodes in order to reconstruct the block. Therefore, it is a good idea to move out all the EC blocks before datanode decommission when a datanode is removed from the cluster. For restarting a datanode, a new hibernation state is introduced in order to indicate that the datanode is going to rejoin the cluster soon. The namenode should not schedule any EC block reconstruction work for it.

MapReduce Data Locality

MapReduce works on the granularity of *records*. Therefore we need to align record and stripe boundaries. This alignment can be done with a separate MapReduce job. The scheduling algorithm for MapReduce tasks should also be aware of the striping pattern.

Block Movement

Both **`Balancer`** and **`Mover`** move block replicas across *DataNodes* (possibly across racks). This might place blocks in the same coding group on the same rack or *DataNode* and thereby reduce the level of fault tolerance. This compatibility issue can be addressed with 2 options:

- A simple approach is to disable balancing for erasure-coded blocks. Meanwhile, since data storage policies (i.e., HOT/WARM/COLD) are explicitly defined by users, we should honor them and skip encoding a block if it is to be moved.
- A more efficient approach is to make **`Balancer`** and **`Mover`** aware of EC groups, and avoid the aforementioned placement decisions.

Appendix: 3-replication vs (6,3)-Reed-Solomon

We compare 3-replication with (6,3)-Reed-Solomon in this section

Toleration

	3-replication	**(6,3)-Reed-Solomon**
Maximum Toleration	2	3

Minimum blocks required for data access

	3-replication	**(6,3)-Reed-Solomon**
1 block	1	6
2 blocks	2	6
3 blocks	3	6
4 blocks	4	6
5 blocks	5	6
6 blocks	6	6

Disk space usage:

	3-replication	**(6,3)-Reed-Solomon**
n bytes of data	3n	1.5n

Name space usage:

	3-replication	**(6,3)-Reed-Solomon**
1 block	1 block + 3 locations	1 block group + 9 locations
2 blocks	2 blocks + 6 locations	1 block group + 9 locations
3 blocks	3 blocks + 9 locations	1 block group + 9 locations
4 blocks	4 blocks + 12 locations	1 block group + 9 locations
5 blocks	5 blocks + 15 locations	1 block group + 9 locations
6 blocks	6 blocks + 18 locations	1 block group + 9 locations

Number of Client-Datanode Connections

	3-replication	**(6,3)-Reed-Solomon**
Write	1	9
Read	1	6

Network traffic

Assume maximum two blocks per rack.
Keys: LN = local node, LR = local rack, RR = remote rack

Writing data.

	3-replication	**(6,3)-Reed-Solomon**
Write	1 LN + 1 LR + 1 RR	1/6 LN + 1/6 LR + 7/6 RR

Reading data assuming the client can be moved to any datanode and the cluster is large so that two different blocks always store in different racks.

	3-replication	**(6,3)-Reed-Solomon**
Read 1 block	1 LN	1/6 LN + 5/6 RR
Read 2 blocks	1 LN + 1 RR	2/6 LN + 10/6 RR
Read 3 blocks	1 LN + 2 RR	3/6 LN + 15/6 RR
Read 4 blocks	1 LN + 3 RR	4/6 LN + 20/6 RR
Read 5 blocks	1 LN + 4 RR	5/6 LN + 25/6 RR
Read 6 blocks	1 LN + 5 RR	1 LN + 5 RR

Block reconstruction assuming rack local is possible for 1-missing.

	3-replication	**(6,3)-Reed-Solomon**
1-missing	1 LR	1 LR + 5 RR
2-missing	1 LR + 1 RR	1 LR + 6 RR
3-missing	Unrecoverable	1 LR + 7 RR

References:

[Fan09] "*DiskReduce: RAID for Data-Intensive Scalable Computing*", PDSW 2009

[Ford10] "*Availability in Globally Distributed Storage Systems*", OSDI 2010

[Harter14] "*Analysis of HDFS Under HBase: A Facebook Messages Case Study*", FAST 2014

[Huang12] "*Erasure Coding in Windows Azure Storage*", USENIX ATC 2012

[Khan12] "*Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads*", FAST 2012

[HDFS-265] Append/Hflush/Read Design

[HDFS-3689] Add support for variable length block

[Intel-ISA] https://01.org/intel@-storage-acceleration-library-open-source-version

[Muralidhar14] "*f4: Facebook's Warm BLOB Storage System*", OSDI 2014

[Nightingale12] "*Flat Datacenter Storage*", OSDI 2012

[Ovsiannikov13] "*The Quantcast File System*", VLDB 2013

[Sathiamoorthy13] "*XORing Elephants: Novel Erasure Codes for Big Data*", VLDB 2013