ECBACKEND IMPLEMENTATION STRATEGY

MISC INITIAL DESIGN NOTES

The initial (and still true for ec pools without the hacky ec overwrites debug flag enabled) design for ec pools restricted EC pools to operations which can be easily rolled back:

- CEPH_OSD_OP_APPEND: We can roll back an append locally by including the previous object size as part of the PG log event.
- CEPH_OSD_OP_DELETE: The possibility of rolling back a delete requires that we retain the deleted object until all replicas have persisted the deletion event. ErasureCoded backend will therefore need to store objects with the version at which they were created included in the key provided to the filestore. Old versions of an object can be pruned when all replicas have committed up to the log event deleting the object.
- CEPH_OSD_OP_(SET|RM)ATTR: If we include the prior value of the attr to be set or removed, we can roll back these operations locally.

Log entries contain a structure explaining how to locally undo the operation represented by the operation (see osd_types.h:TransactionInfo::LocalRollBack).

PGTEMP AND CRUSH

Primaries are able to request a temp acting set mapping in order to allow an up-to-date OSD to serve requests while a new primary is backfilled (and for other reasons). An erasure coded pg needs to be able to designate a primary for these reasons without putting it in the first position of the acting set. It also needs to be able to leave holes in the requested acting set.

Core Changes:

- OSDMap::pg_to_*_osds needs to separately return a primary. For most cases, this can continue to be acting[0].
- MOSDPGTemp (and related OSD structures) needs to be able to specify a primary as well as an acting set.
- Much of the existing code base assumes that acting[0] is the primary and that all elements of acting are valid. This needs to be cleaned up since the acting set may contain holes.

DISTINGUISHED ACTING SET POSITIONS

With the replicated strategy, all replicas of a PG are interchangeable. With erasure coding, different positions in the acting set have different pieces of the erasure coding scheme and are not interchangeable. Worse, crush might cause chunk 2 to be written to an OSD which happens already to contain an (old) copy of chunk 4. This means that the OSD and PG messages need to work in terms of a type like pair<shard_t, pg_t> in order to distinguish different pg chunks on a single OSD.

Because the mapping of object name to object in the filestore must be 1-to-1, we must ensure that the objects in chunk 2 and the objects in chunk 4 have different names. To that end, the objectstore must include the chunk id in the object key.

Core changes:

- The objectstore ghobject_t needs to also include a chunk id making it more like tuple<hobject_t, gen_t, shard_t>.
- coll_t needs to include a shard_t.
- The OSD pg_map and similar pg mappings need to work in terms of a spg_t (essentially pair<pg_t, shard_t>). Similarly, pg->pg messages need to include a shard_t
- For client->PG messages, the OSD will need a way to know which PG chunk should get the message since the OSD may contain both a primary and non-primary chunk for the same pg

OBJECT CLASSES

Reads from object classes will return ENOTSUP on ec pools by invoking a special SYNC read.

SCRUB

The main catch, however, for ec pools is that sending a crc32 of the stored chunk on a replica isn’t particularly helpful since the chunks on different replicas presumably store different data. Because we don’t support overwrites except via DELETE, however, we have the option of maintaining a crc32 on each chunk through each append. Thus, each replica instead simply computes a crc32 of its own stored chunk and compares it with the locally stored checksum. The replica then reports to the primary whether the checksums match.

With overwrites, all scrubs are disabled for now until we work out what to do (see doc/dev/osd_internals/erasure_coding/proposals.rst).

CRUSH

If crush is unable to generate a replacement for a down member of an acting set, the acting set should have a hole at that position rather than shifting the other elements of the acting set out of position.

ECBACKEND

MAIN OPERATION OVERVIEW

A RADOS put operation can span multiple stripes of a single object. There must be code that tessellates the application level write into a set of per-stripe write operations – some whole-stripes and up to two partial stripes. Without loss of generality, for the remainder of this document we will focus exclusively on writing a single stripe (whole or partial). We will use the symbol “W” to represent the number of blocks within a stripe that are being written, i.e., W <= K.

There are three data flows for handling a write into an EC stripe. The choice of which of the three data flows to choose is based on the size of the write operation and the arithmetic properties of the selected parity-generation algorithm.

1. whole stripe is written/overwritten
2. a read-modify-write operation is performed.

WHOLE STRIPE WRITE

This is the simple case, and is already performed in the existing code (for appends, that is). The primary receives all of the data for the stripe in the RADOS request, computes the appropriate parity blocks and send the data and parity blocks to their destination shards which write them. This is essentially the current EC code.

READ-MODIFY-WRITE

The primary determines which of the K-W blocks are to be unmodified, and reads them from the shards. Once all of the data is received it is combined with the received new data and new parity blocks are computed. The modified blocks are sent to their respective shards and written. The RADOS operation is acknowledged.

OSD OBJECT WRITE AND CONSISTENCY

Regardless of the algorithm chosen above, writing of the data is a two phase process: commit and rollforward. The primary sends the log entries with the operation described (see osd_types.h:TransactionInfo::(LocalRollForward|LocalRollBack). In all cases, the “commit” is performed in place, possibly leaving some information required for a rollback in a write-aside object. The rollforward phase occurs once all acting set replicas have committed the commit (sorry, overloaded term) and removes the rollback information.

In the case of overwrites of exsting stripes, the rollback information has the form of a sparse object containing the old values of the overwritten extents populated using clone_range. This is essentially a place-holder implementation, in real life, bluestore will have an efficient primitive for this.

The rollforward part can be delayed since we report the operation as committed once all replicas have committed. Currently, whenever we send a write, we also indicate that all previously committed operations should be rolled forward (see ECBackend::try_reads_to_commit). If there aren’t any in the pipeline when we arrive at the waiting_rollforward queue, we start a dummy write to move things along (see the Pipeline section later on and ECBackend::try_finish_rmw).

EXTENTCACHE

It’s pretty important to be able to pipeline writes on the same object. For this reason, there is a cache of extents written by cacheable operations. Each extent remains pinned until the operations referring to it are committed. The pipeline prevents rmw operations from running until uncacheable transactions (clones, etc) are flushed from the pipeline.

See ExtentCache.h for a detailed explanation of how the cache states correspond to the higher level invariants about the conditions under which cuncurrent operations can refer to the same object.

PIPELINE

Reading src/osd/ExtentCache.h should have given a good idea of how operations might overlap. There are several states involved in processing a write operation and an important invariant which isn’t enforced by PrimaryLogPG at a higher level which need to be managed by ECBackend. The important invariant is that we can’t have uncacheable and rmw operations running at the same time on the same object. For simplicity, we simply enforce that any operation which contains an rmw operation must wait until all in-progress uncacheable operations complete.

There are improvements to be made here in the future.

For more details, see ECBackend::waiting_* and ECBackend::try_<from>_to_<to>.