LOG BASED PG

BACKGROUND

WHY PRIMARYLOGPG?

Currently, consistency for all ceph pool types is ensured by primary log-based replication. This goes for both erasure-coded and replicated pools.

PRIMARY LOG-BASED REPLICATION

Reads must return data written by any write which completed (where the client could possibly have received a commit message). There are lots of ways to handle this, but ceph's architecture makes it easy for everyone at any map epoch to know who the primary is. Thus, the easy answer is to route all writes for a particular pg through a single ordering primary and then out to the replicas. Though we only actually need to serialize writes on a single object (and even then, the partial ordering only really needs to provide an ordering between writes on overlapping regions), we might as well serialize writes on the whole PG since it lets us represent the current state of the PG using two numbers: the epoch of the map on the primary in which the most recent write started (this is a bit stranger than it might seem since map distribution itself is asyncronous – see Peering and the concept of interval changes) and an increasing per-pg version number – this is referred to in the code with type eversion_t and stored as pg_info_t::last_update. Furthermore, we maintain a log of "recent" operations extending back at least far enough to include any *unstable* writes (writes which have been started but not committed) and objects which aren't uptodate locally (see recovery and backfill). In practice, the log will extend much further (osd_pg_min_log_entries when clean, osd_pg_max_log_entries when not clean) because it's handy for quickly performing recovery.

Using this log, as long as we talk to a non-empty subset of the OSDs which must have accepted any completed writes from the most recent interval in which we accepted writes, we can determine a conservative log which must contain any write which has been reported to a client as committed. There is some freedom here, we can choose any log entry between the oldest head remembered by an element of that set (any newer cannot have completed without that log containing it) and the newest head remembered (clearly, all writes in the log were started, so it's fine for us to remember them) as the new head. This is the main point of divergence between replicated pools and ec pools in PG/PrimaryLogPG: replicated pools try to choose the newest valid option to avoid the client needing to replay those operations and instead recover the other copies. EC pools instead try to choose the *oldest* option available to them.

The reason for this gets to the heart of the rest of the differences in implementation: one copy will not generally be enough to reconstruct an ec object. Indeed, there are encodings where some log combinations would leave unrecoverable objects (as with a 4+2 encoding where 3 of the replicas remember a write, but the other 3 do not – we don't have 3 copies of either version). For this reason, log entries representing *unstable* writes (writes not yet committed to the client) must be rollbackable using only local information on ec pools. Log entries in general may therefore be rollbackable (and in that case, via a delayed application or via a set of instructions for rolling back an inplace update) or not. Replicated pool log entries are never able to be rolled back.

For more details, see PGLog.h/cc, osd_types.h:pg_log_t, osd_types.h:pg_log_entry_t, and peering in general.

REPLICATEDBACKEND/ECBACKEND UNIFICATION STRATEGY

PGBACKEND

So, the fundamental difference between replication and erasure coding is that replication can do destructive updates while erasure coding cannot. It would be really annoying if we needed to have two entire implementations of PrimaryLogPG, one for each of the two, if there are really only a few fundamental differences:

1. How reads work – async only, requires remote reads for ec
2. How writes work – either restricted to append, or must write aside and do a tpc
3. Whether we choose the oldest or newest possible head entry during peering
4. A bit of extra information in the log entry to enable rollback

and so many similarities

1. All of the stats and metadata for objects

2. The high level locking rules for mixing client IO with recovery and scrub
3. The high level locking rules for mixing reads and writes without exposing uncommitted state (which might be rolled back or forgotten later)
4. The process, metadata, and protocol needed to determine the set of osds which partcipated in the most recent interval in which we accepted writes
5. etc.

Instead, we choose a few abstractions (and a few kludges) to paper over the differences:

1. PGBackend
2. PGTransaction
3. PG::choose_acting chooses between calc_replicated_acting and calc_ec_acting
4. Various bits of the write pipeline disallow some operations based on pool type – like omap operations, class operation reads, and writes which are not aligned appends (officially, so far) for ec
5. Misc other kludges here and there

PGBackend and PGTransaction enable abstraction of differences 1, 2, and the addition of 4 as needed to the log entries.

The replicated implementation is in ReplicatedBackend.h/cc and doesn't require much explanation, I think. More detail on the ECBackend can be found in doc/dev/osd_internals/erasure_coding/ecbackend.rst.

PGBACKEND INTERFACE EXPLANATION

Note: this is from a design document from before the original firefly and is probably out of date w.r.t. some of the method names.

READABLE VS DEGRADED

For a replicated pool, an object is readable iff it is present on the primary (at the right version). For an ec pool, we need at least M shards present to do a read, and we need it on the primary. For this reason, PGBackend needs to include some interfaces for determing when recovery is required to serve a read vs a write. This also changes the rules for when peering has enough logs to prove that it

Core Changes:

- PGBackend needs to be able to return IsPG(Recoverable|Readable)Predicate objects to allow the user to make these determinations.

CLIENT READS

Reads with the replicated strategy can always be satisfied synchronously out of the primary OSD. With an erasure coded strategy, the primary will need to request data from some number of replicas in order to satisfy a read. PGBackend will therefore need to provide seperate objects_read_sync and objects_read_async interfaces where the former won't be implemented by the ECBackend.

PGBackend interfaces:

- objects_read_sync
- objects_read_async

SCRUB

We currently have two scrub modes with different default frequencies:

1. [shallow] scrub: compares the set of objects and metadata, but not the contents
2. deep scrub: compares the set of objects, metadata, and a crc32 of the object contents (including omap)

The primary requests a scrubmap from each replica for a particular range of objects. The replica fills out this scrubmap for the range of objects including, if the scrub is deep, a crc32 of the contents of each object. The primary gathers these scrubmaps from each replica and performs a comparison identifying inconsistent objects.

Most of this can work essentially unchanged with erasure coded PG with the caveat that the PGBackend implementation must be in charge of actually doing the scan.

PGBackend interfaces:

- be_*

RECOVERY

The logic for recovering an object depends on the backend. With the current replicated strategy, we first pull the object replica to the primary and then concurrently push it out to the replicas. With the erasure coded strategy, we probably want to read the minimum number of replica chunks required to reconstruct the object and push out the replacement chunks concurrently.

Another difference is that objects in erasure coded pg may be unrecoverable without being unfound. The “unfound” concept should probably then be renamed to unrecoverable. Also, the PGBackend implementation will have to be able to direct the search for pg replicas with unrecoverable object chunks and to be able to determine whether a particular object is recoverable.

Core changes:

- s/unfound/unrecoverable

PGBackend interfaces:

- on_local_recover_start
- on_local_recover
- on_global_recover
- on_peer_recover
- begin_peer_recover