SNAPS

OVERVIEW

Rados supports two related snapshotting mechanisms:

1. *pool snaps*: snapshots are implicitely applied to all objects in a pool
2. *self managed snaps*: the user must provide the current *SnapContext* on each write.

These two are mutually exclusive, only one or the other can be used on a particular pool.

The *SnapContext* is the set of snapshots currently defined for an object as well as the most recent snapshot (the *seq*) requested from the mon for sequencing purposes (a *SnapContext* with a newer *seq* is considered to be more recent).

The difference between *pool snaps* and *self managed snaps* from the OSD's point of view lies in whether the *SnapContext* comes to the OSD via the client's MOSDOp or via the most recent OSDMap.

See OSD::make_writeable

ONDISK STRUCTURES

Each object has in the pg collection a *head* object (or *snapdir*, which we will come to shortly) and possibly a set of *clone* objects. Each hobject_t has a snap field. For the *head* (the only writeable version of an object), the snap field is set to CEPH_NOSNAP. For the *clones*, the snap field is set to the *seq* of the *SnapContext* at their creation. When the OSD services a write, it first checks whether the most recent *clone* is tagged with a snapid prior to the most recent snap represented in the *SnapContext*. If so, at least one snapshot has occurred between the time of the write and the time of the last clone. Therefore, prior to performing the mutation, the OSD creates a new clone for servicing reads on snaps between the snapid of the last clone and the most recent snapid.

The *head* object contains a *SnapSet* encoded in an attribute, which tracks

1. The full set of snaps defined for the object
2. The full set of clones which currently exist
3. Overlapping intervals between clones for tracking space usage
4. Clone size

If the *head* is deleted while there are still clones, a *snapdir* object is created instead to house the *SnapSet*.

Additionally, the *object_info_t* on each clone includes a vector of snaps for which clone is defined.

SNAP REMOVAL

To remove a snapshot, a request is made to the *Monitor* cluster to add the snapshot id to the list of purged snaps (or to remove it from the set of pool snaps in the case of *pool snaps*). In either case, the *PG* adds the snap to its *snap_trimq* for trimming.

A clone can be removed when all of its snaps have been removed. In order to determine which clones might need to be removed upon snap removal, we maintain a mapping from snap to *hobject_t* using the *SnapMapper*.

See PrimaryLogPG::SnapTrimmer, SnapMapper

This trimming is performed asynchronously by the snap_trim_wq while the pg is clean and not scrubbing.

1. The next snap in PG::snap_trimq is selected for trimming
2. We determine the next object for trimming out of PG::snap_mapper. For each object, we create a log entry and repop updating the object info and the snap set (including adjusting the overlaps). If the object is a clone which no longer belongs to any live snapshots, it is removed here. (See PrimaryLogPG::trim_object() when new_snaps is empty.)
3. We also locally update our *SnapMapper* instance with the object's new snaps.
4. The log entry containing the modification of the object also contains the new set of snaps, which the replica uses to update its own *SnapMapper* instance.
5. The primary shares the info with the replica, which persists the new set of purged_snaps along with the rest of the info.

RECOVERY

Because the trim operations are implemented using repops and log entries, normal pg peering and recovery maintain the snap trimmer operations with the caveat that push and removal operations need to update the local *SnapMapper* instance. If the purged_snaps update is lost, we merely retrim a now empty snap.

SNAPMAPPER

SnapMapper is implemented on top of map_cacher<string, bufferlist>, which provides an interface over a backing store such as the filesystem with async transactions. While transactions are incomplete, the map_cacher instance buffers unstable keys allowing consistent access without having to flush the filestore. *SnapMapper* provides two mappings:

1. hobject_t -> set<snapid_t>: stores the set of snaps for each clone object
2. snapid_t -> hobject_t: stores the set of hobjects with the snapshot as one of its snaps

Assumption: there are lots of hobjects and relatively few snaps. The first encoding has a stringification of the object as the key and an encoding of the set of snaps as a value. The second mapping, because there might be many hobjects for a single snap, is stored as a collection of keys of the form stringify(snap)_stringify(object) such that stringify(snap) is constant length. These keys have a bufferlist encoding pair<snapid, hobject_t> as a value. Thus, creating or trimming a single object does not involve reading all objects for any snap. Additionally, upon construction, the *SnapMapper* is provided with a mask for filtering the objects in the single SnapMapper keyspace belonging to that pg.

SPLIT

The snapid_t -> hobject_t key entries are arranged such that for any pg, up to 8 prefixes need to be checked to determine all hobjects in a particular snap for a particular pg. Upon split, the prefixes to check on the parent are adjusted such that only the objects remaining in the pg will be visible. The children will immediately have the correct mapping.