PG (PLACEMENT GROUP) NOTES

Miscellaneous copy-pastes from emails, when this gets cleaned up it should move out of /dev.

OVERVIEW

PG = “placement group”. When placing data in the cluster, objects are mapped into PGs, and those PGs are mapped onto OSDs. We use the indirection so that we can group objects, which reduces the amount of per-object metadata we need to keep track of and processes we need to run (it would be prohibitively expensive to track eg the placement history on a per-object basis). Increasing the number of PGs can reduce the variance in per-OSD load across your cluster, but each PG requires a bit more CPU and memory on the OSDs that are storing it. We try and ballpark it at 100 PGs/OSD, although it can vary widely without ill effects depending on your cluster. You hit a bug in how we calculate the initial PG number from a cluster description.

There are a couple of different categories of PGs; the 6 that exist (in the original emailer’s ceph -s output) are “local” PGs which are tied to a specific OSD. However, those aren’t actually used in a standard Ceph configuration.

MAPPING ALGORITHM (SIMPLIFIED)

> How does the Object->PG mapping look like, do you map more than one object on
> one PG, or do you sometimes map an object to more than one PG? How about the
> mapping of PGs to OSDs, does one PG belong to exactly one OSD?
>
> Does one PG represent a fixed amount of storage space?

Many objects map to one PG.

Each object maps to exactly one PG.

One PG maps to a single list of OSDs, where the first one in the list is the primary and the rest are replicas.

Many PGs can map to one OSD.

A PG represents nothing but a grouping of objects; you configure the number of PGs you want (see http://ceph.com/wiki/Changing_the_number_of_PGs), number of OSDs * 100 is a good starting point, and all of your stored objects are pseudo-randomly evenly distributed to the PGs. So a PG explicitly does NOT represent a fixed amount of storage; it represents 1/pg_num ‘th of the storage you happen to have on your OSDs.

Ignoring the finer points of CRUSH and custom placement, it goes something like this in pseudocode:

```
locator = object_name
obj_hash = hash(locator)
pg = obj_hash % num_pg
osds_for_pg = crush(pg)  # returns a list of osds
primary = osds_for_pg[0]
replicas = osds_for_pg[1:]
```

If you want to understand the crush() part in the above, imagine a perfectly spherical datacenter in a vacuum ;) that is, if all osds have weight 1.0, and there is no topology to the data center (all OSDs are on the top level), and you use defaults, etc, it simplifies to consistent hashing; you can think of it as:

```
def crush(pg):
   all_osds = ['osd.0', 'osd.1', 'osd.2', ...]
   result = []
   # size is the number of copies; primary+replicas
   while len(result) < size:
       r = hash(pg)
       chosen = all_osds[ r % len(all_osds) ]
       if chosen in result:
           # osd can be picked only once
           continue
       result.append(chosen)
   return result
```

USER-VISIBLE PG STATES

Todo: diagram of states and how they can overlap

creating
: the PG is still being created

active
: requests to the PG will be processed

clean
: all objects in the PG are replicated the correct number of times

down
: a replica with necessary data is down, so the pg is offline

replay
: the PG is waiting for clients to replay operations after an OSD crashed

splitting
: the PG is being split into multiple PGs (not functional as of 2012-02)

scrubbing
: the PG is being checked for inconsistencies

degraded
: some objects in the PG are not replicated enough times yet

inconsistent
: replicas of the PG are not consistent (e.g. objects are the wrong size, objects are missing from one replica *after* recovery finished, etc.)

peering
: the PG is undergoing the Peering process

repair
: the PG is being checked and any inconsistencies found will be repaired (if possible)

recovering
: objects are being migrated/synchronized with replicas

recovery_wait
: the PG is waiting for the local/remote recovery reservations

backfill
: a special case of recovery, in which the entire contents of the PG are scanned and synchronized, instead of inferring what needs to be transferred from the PG logs of recent operations

backfill-wait
: the PG is waiting in line to start backfill

backfill_toofull
: backfill reservation rejected, OSD too full

incomplete
: a pg is missing a necessary period of history from its log. If you see this state, report a bug, and try to start any failed OSDs that may contain the needed information.

stale
: the PG is in an unknown state - the monitors have not received an update for it since the PG mapping changed.

remapped
: the PG is temporarily mapped to a different set of OSDs from what CRUSH specified