TROUBLESHOOTING PGS

PLACEMENT GROUPS NEVER GET CLEAN

When you create a cluster and your cluster remains in `active`, `active+remapped` or `active+degraded` status and never achieve an `active+clean` status, you likely have a problem with your configuration.

You may need to review settings in the Pool, PG and CRUSH Config Reference and make appropriate adjustments.

As a general rule, you should run your cluster with more than one OSD and a pool size greater than 1 object replica.

ONE NODE CLUSTER

Ceph no longer provides documentation for operating on a single node, because you would never deploy a system designed for distributed computing on a single node. Additionally, mounting client kernel modules on a single node containing a Ceph daemon may cause a deadlock due to issues with the Linux kernel itself (unless you use VMs for the clients). You can experiment with Ceph in a 1-node configuration, in spite of the limitations as described herein.

If you are trying to create a cluster on a single node, you must change the default of the `osd crush chooseleaf type` setting from 1 (meaning `host` or `node`) to `0` (meaning `osd`) in your Ceph configuration file before you create your monitors and OSDs. This tells Ceph that an OSD can peer with another OSD on the same host. If you are trying to set up a 1-node cluster and `osd crush chooseleaf type` is greater than `0`, Ceph will try to peer the PGs of one OSD with the PGs of another OSD on another node, chassis, rack, row, or even datacenter depending on the setting.

Tip: DO NOT mount kernel clients directly on the same node as your Ceph Storage Cluster, because kernel conflicts can arise. However, you can mount kernel clients within virtual machines (VMs) on a single node.

If you are creating OSDs using a single disk, you must create directories for the data manually first. For example:

```
ceph-deploy osd create --data {disk} {host}
```

FEWER OSDS THAN REPLICAS

If you have brought up two OSDs to an `up` and `in` state, but you still don't see `active + clean` placement groups, you may have an `osd pool default size` set to greater than 2.

There are a few ways to address this situation. If you want to operate your cluster in an `active + degraded` state with two replicas, you can set the `osd pool default min size` to 2 so that you can write objects in an `active + degraded` state. You may also set the `osd pool default size` setting to 2 so that you only have two stored replicas (the original and one replica), in which case the cluster should achieve an `active + clean` state.

Note: You can make the changes at runtime. If you make the changes in your Ceph configuration file, you may need to restart your cluster.

POOL SIZE = 1

If you have the `osd pool default size` set to 1, you will only have one copy of the object. OSDs rely on other OSDs to tell them which objects they should have. If a first OSD has a copy of an object and there is no second copy, then no second OSD can tell the first OSD that it should have that copy. For each placement group mapped to the first OSD (see `ceph pg dump`), you can force the first OSD to notice the placement groups it needs by running:

```
ceph osd force-create-pg <pgid>
```

CRUSH MAP ERRORS

Another candidate for placement groups remaining unclean involves errors in your CRUSH map.

STUCK PLACEMENT GROUPS

It is normal for placement groups to enter states like “degraded” or “peering” following a failure. Normally these states indicate the normal progression through the failure recovery process. However, if a placement group stays in one of these states for a long time this may be an indication of a larger problem. For this reason, the monitor will warn when placement groups get “stuck” in a non-optimal state. Specifically, we check for:

- `inactive` - The placement group has not been `active` for too long (i.e., it hasn’t been able to service read/write requests).
- `unclean` - The placement group has not been `clean` for too long (i.e., it hasn’t been able to completely recover from a previous failure).
- `stale` - The placement group status has not been updated by a `ceph-osd`, indicating that all nodes storing this placement group may be down.

You can explicitly list stuck placement groups with one of:

```
ceph pg dump_stuck stale
ceph pg dump_stuck inactive
ceph pg dump_stuck unclean
```

For stuck `stale` placement groups, it is normally a matter of getting the right `ceph-osd` daemons running again. For stuck `inactive` placement groups, it is usually a peering problem (see Placement Group Down - Peering Failure). For stuck `unclean` placement groups, there is usually something preventing recovery from completing, like unfound objects (see Unfound Objects);

PLACEMENT GROUP DOWN - PEERING FAILURE

In certain cases, the `ceph-osd` *Peering* process can run into problems, preventing a PG from becoming active and usable. For example, `ceph health` might report:

```
ceph health detail
HEALTH_ERR 7 pgs degraded; 12 pgs down; 12 pgs peering; 1 pgs recovering; 6 pgs stuck unclean
...
pg 0.5 is down+peering
pg 1.4 is down+peering
...
osd.1 is down since epoch 69, last address 192.168.106.220:6801/8651
```

We can query the cluster to determine exactly why the PG is marked down with:

```
ceph pg 0.5 query
```

```
{ "state": "down+peering",
  ...
  "recovery_state": [
        { "name": "Started\/Primary\/Peering\/GetInfo",
          "enter_time": "2012-03-06 14:40:16.169679",
          "requested_info_from": []},
        { "name": "Started\/Primary\/Peering",
          "enter_time": "2012-03-06 14:40:16.169659",
          "probing_osds": [
                0,
                1],
          "blocked": "peering is blocked due to down osds",
          "down_osds_we_would_probe": [
                1],
          "peering_blocked_by": [
                { "osd": 1,
                  "current_lost_at": 0,
                  "comment": "starting or marking this osd lost may let us proceed"}]},
        { "name": "Started",
          "enter_time": "2012-03-06 14:40:16.169513"}
    ]
```

```
}
```

The `recovery_state` section tells us that peering is blocked due to down `ceph-osd` daemons, specifically `osd.1`. In this case, we can start that `ceph-osd` and things will recover.

Alternatively, if there is a catastrophic failure of `osd.1` (e.g., disk failure), we can tell the cluster that it is `lost` and to cope as best it can.

Important: This is dangerous in that the cluster cannot guarantee that the other copies of the data are consistent and up to date.

To instruct Ceph to continue anyway:

```
ceph osd lost 1
```

Recovery will proceed.

UNFOUND OBJECTS

Under certain combinations of failures Ceph may complain about `unfound` objects:

```
ceph health detail
HEALTH_WARN 1 pgs degraded; 78/3778 unfound (2.065%)
pg 2.4 is active+degraded, 78 unfound
```

This means that the storage cluster knows that some objects (or newer copies of existing objects) exist, but it hasn't found copies of them. One example of how this might come about for a PG whose data is on ceph-osds 1 and 2:

- 1 goes down
- 2 handles some writes, alone
- 1 comes up
- 1 and 2 repeer, and the objects missing on 1 are queued for recovery.
- Before the new objects are copied, 2 goes down.

Now 1 knows that these object exist, but there is no live `ceph-osd` who has a copy. In this case, IO to those objects will block, and the cluster will hope that the failed node comes back soon; this is assumed to be preferable to returning an IO error to the user.

First, you can identify which objects are unfound with:

```
ceph pg 2.4 list_missing [starting offset, in json]
```

```
{ "offset": { "oid": "",
      "key": "",
      "snapid": 0,
      "hash": 0,
      "max": 0},
  "num_missing": 0,
  "num_unfound": 0,
  "objects": [
     { "oid": "object 1",
       "key": "",
       "hash": 0,
       "max": 0 },
     ...
  ],
  "more": 0}
```

If there are too many objects to list in a single result, the `more` field will be true and you can query for more. (Eventually the command line tool will hide this from you, but not yet.)

Second, you can identify which OSDs have been probed or might contain data:

```
ceph pg 2.4 query
```

```
"recovery_state": [
     { "name": "Started\/Primary\/Active",
       "enter_time": "2012-03-06 15:15:46.713212",
       "might_have_unfound": [
             { "osd": 1,
               "status": "osd is down"}]},
```

In this case, for example, the cluster knows that `osd.1` might have data, but it is `down`. The full range of possible states include:

- already probed
- querying
- OSD is down
- not queried (yet)

Sometimes it simply takes some time for the cluster to query possible locations.

It is possible that there are other locations where the object can exist that are not listed. For example, if a ceph-osd is stopped and taken out of the cluster, the cluster fully recovers, and due to some future set of failures ends up with an unfound object, it won't consider the long-departed ceph-osd as a potential location to consider. (This scenario, however, is unlikely.)

If all possible locations have been queried and objects are still lost, you may have to give up on the lost objects. This, again, is possible given unusual combinations of failures that allow the cluster to learn about writes that were performed before the writes themselves are recovered. To mark the "unfound" objects as "lost":

```
ceph pg 2.5 mark_unfound_lost revert|delete
```

This the final argument specifies how the cluster should deal with lost objects.

The "delete" option will forget about them entirely.

The "revert" option (not available for erasure coded pools) will either roll back to a previous version of the object or (if it was a new object) forget about it entirely. Use this with caution, as it may confuse applications that expected the object to exist.

HOMELESS PLACEMENT GROUPS

It is possible for all OSDs that had copies of a given placement groups to fail. If that's the case, that subset of the object store is unavailable, and the monitor will receive no status updates for those placement groups. To detect this situation, the monitor marks any placement group whose primary OSD has failed as `stale`. For example:

```
ceph health
HEALTH_WARN 24 pgs stale; 3/300 in osds are down
```

You can identify which placement groups are `stale`, and what the last OSDs to store them were, with:

```
ceph health detail
HEALTH_WARN 24 pgs stale; 3/300 in osds are down
...
pg 2.5 is stuck stale+active+remapped, last acting [2,0]
...
osd.10 is down since epoch 23, last address 192.168.106.220:6800/11080
osd.11 is down since epoch 13, last address 192.168.106.220:6803/11539
osd.12 is down since epoch 24, last address 192.168.106.220:6806/11861
```

If we want to get placement group 2.5 back online, for example, this tells us that it was last managed by `osd.0` and `osd.2`. Restarting those `ceph-osd` daemons will allow the cluster to recover that placement group (and, presumably, many others).

ONLY A FEW OSDS RECEIVE DATA

If you have many nodes in your cluster and only a few of them receive data, check the number of placement groups in your

pool. Since placement groups get mapped to OSDs, a small number of placement groups will not distribute across your cluster. Try creating a pool with a placement group count that is a multiple of the number of OSDs. See Placement Groups for details. The default placement group count for pools is not useful, but you can change it here.

CAN'T WRITE DATA

If your cluster is up, but some OSDs are down and you cannot write data, check to ensure that you have the minimum number of OSDs running for the placement group. If you don't have the minimum number of OSDs running, Ceph will not allow you to write data because there is no guarantee that Ceph can replicate your data. See `osd pool default min size` in the Pool, PG and CRUSH Config Reference for details.

PGS INCONSISTENT

If you receive an `active + clean + inconsistent` state, this may happen due to an error during scrubbing. As always, we can identify the inconsistent placement group(s) with:

```
$ ceph health detail
HEALTH_ERR 1 pgs inconsistent; 2 scrub errors
pg 0.6 is active+clean+inconsistent, acting [0,1,2]
2 scrub errors
```

Or if you prefer inspecting the output in a programmatic way:

```
$ rados list-inconsistent-pg rbd
["0.6"]
```

There is only one consistent state, but in the worst case, we could have different inconsistencies in multiple perspectives found in more than one objects. If an object named `foo` in PG `0.6` is truncated, we will have:

```
$ rados list-inconsistent-obj 0.6 --format=json-pretty
```

```
{
    "epoch": 14,
    "inconsistents": [
        {
            "object": {
                "name": "foo",
                "nspace": "",
                "locator": "",
                "snap": "head",
                "version": 1
            },
            "errors": [
                "data_digest_mismatch",
                "size_mismatch"
            ],
            "union_shard_errors": [
                "data_digest_mismatch_oi",
                "size_mismatch_oi"
            ],
            "selected_object_info": "0:602f83fe:::foo:head(16'1 client.4110.0:1 dirty|data_di
            "shards": [
                {
                    "osd": 0,
                    "errors": [],
                    "size": 968,
                    "omap_digest": "0xffffffff",
                    "data_digest": "0xe978e67f"
                },
                {
                    "osd": 1,
                    "errors": [],
                    "size": 968,
                    "omap_digest": "0xffffffff",
```

```
                    "data_digest": "0xe978e67f"
                },
                {
                    "osd": 2,
                    "errors": [
                        "data_digest_mismatch_oi",
                        "size_mismatch_oi"
                    ],
                    "size": 0,
                    "omap_digest": "0xffffffff",
                    "data_digest": "0xffffffff"
                }
            ]
        }
    ]
}
```


In this case, we can learn from the output:

- The only inconsistent object is named `foo`, and it is its head that has inconsistencies.
- The inconsistencies fall into two categories:
  - `errors`: these errors indicate inconsistencies between shards without a determination of which shard(s) are bad. Check for the `errors` in the *shards* array, if available, to pinpoint the problem.
    - `data_digest_mismatch`: the digest of the replica read from OSD.2 is different from the ones of OSD.0 and OSD.1
    - `size_mismatch`: the size of the replica read from OSD.2 is 0, while the size reported by OSD.0 and OSD.1 is 968.
  - `union_shard_errors`: the union of all shard specific `errors` in `shards` array. The `errors` are set for the given shard that has the problem. They include errors like `read_error`. The `errors` ending in `oi` indicate a comparison with `selected_object_info`. Look at the `shards` array to determine which shard has which error(s).
    - `data_digest_mismatch_oi`: the digest stored in the object-info is not `0xffffffff`, which is calculated from the shard read from OSD.2
    - `size_mismatch_oi`: the size stored in the object-info is different from the one read from OSD.2. The latter is 0.

You can repair the inconsistent placement group by executing:

```
ceph pg repair {placement-group-ID}
```

Which overwrites the *bad* copies with the *authoritative* ones. In most cases, Ceph is able to choose authoritative copies from all available replicas using some predefined criteria. But this does not always work. For example, the stored data digest could be missing, and the calculated digest will be ignored when choosing the authoritative copies. So, please use the above command with caution.

If `read_error` is listed in the `errors` attribute of a shard, the inconsistency is likely due to disk errors. You might want to check your disk used by that OSD.

If you receive `active + clean + inconsistent` states periodically due to clock skew, you may consider configuring your NTP daemons on your monitor hosts to act as peers. See The Network Time Protocol and Ceph Clock Settings for additional details.

ERASURE CODED PGS ARE NOT ACTIVE+CLEAN

When CRUSH fails to find enough OSDs to map to a PG, it will show as a 2147483647 which is ITEM_NONE or `no OSD found`. For instance:

```
[2,1,6,0,5,8,2147483647,7,4]
```

NOT ENOUGH OSDS

If the Ceph cluster only has 8 OSDs and the erasure coded pool needs 9, that is what it will show. You can either create another erasure coded pool that requires less OSDs:

```
ceph osd erasure-code-profile set myprofile k=5 m=3
```

```
ceph osd pool create erasurepool 16 16 erasure myprofile
```

or add a new OSDs and the PG will automatically use them.

CRUSH CONSTRAINTS CANNOT BE SATISFIED

If the cluster has enough OSDs, it is possible that the CRUSH rule imposes constraints that cannot be satisfied. If there are 10 OSDs on two hosts and the CRUSH rule requires that no two OSDs from the same host are used in the same PG, the mapping may fail because only two OSDs will be found. You can check the constraint by displaying (“dumping”) the rule:

```
$ ceph osd crush rule ls
[
    "replicated_rule",
    "erasurepool"]
$ ceph osd crush rule dump erasurepool
{ "rule_id": 1,
  "rule_name": "erasurepool",
  "ruleset": 1,
  "type": 3,
  "min_size": 3,
  "max_size": 20,
  "steps": [
        { "op": "take",
          "item": -1,
          "item_name": "default"},
        { "op": "chooseleaf_indep",
          "num": 0,
          "type": "host"},
        { "op": "emit"}]}
```

You can resolve the problem by creating a new pool in which PGs are allowed to have OSDs residing on the same host with:

```
ceph osd erasure-code-profile set myprofile crush-failure-domain=osd
ceph osd pool create erasurepool 16 16 erasure myprofile
```

CRUSH GIVES UP TOO SOON

If the Ceph cluster has just enough OSDs to map the PG (for instance a cluster with a total of 9 OSDs and an erasure coded pool that requires 9 OSDs per PG), it is possible that CRUSH gives up before finding a mapping. It can be resolved by:

- lowering the erasure coded pool requirements to use less OSDs per PG (that requires the creation of another pool as erasure code profiles cannot be dynamically modified).
- adding more OSDs to the cluster (that does not require the erasure coded pool to be modified, it will become clean automatically)
- use a handmade CRUSH rule that tries more times to find a good mapping. This can be done by setting `set_choose_tries` to a value greater than the default.

You should first verify the problem with `crushtool` after extracting the crushmap from the cluster so your experiments do not modify the Ceph cluster and only work on a local files:

```
$ ceph osd crush rule dump erasurepool
{ "rule_name": "erasurepool",
  "ruleset": 1,
  "type": 3,
  "min_size": 3,
  "max_size": 20,
  "steps": [
        { "op": "take",
          "item": -1,
          "item_name": "default"},
        { "op": "chooseleaf_indep",
          "num": 0,
          "type": "host"},
        { "op": "emit"}]}
$ ceph osd getcrushmap > crush.map
got crush map from osdmap epoch 13
```

```
$ crushtool -i crush.map --test --show-bad-mappings \
   --rule 1 \
   --num-rep 9 \
   --min-x 1 --max-x $((1024 * 1024))
bad mapping rule 8 x 43 num_rep 9 result [3,2,7,1,2147483647,8,5,6,0]
bad mapping rule 8 x 79 num_rep 9 result [6,0,2,1,4,7,2147483647,5,8]
bad mapping rule 8 x 173 num_rep 9 result [0,4,6,8,2,1,3,7,2147483647]
```

Where `--num-rep` is the number of OSDs the erasure code CRUSH rule needs, `--rule` is the value of the `ruleset` field displayed by `ceph osd crush rule dump`. The test will try mapping one million values (i.e. the range defined by `[--min-x,--max-x]`) and must display at least one bad mapping. If it outputs nothing it means all mappings are successfull and you can stop right there: the problem is elsewhere.

The CRUSH rule can be edited by decompiling the crush map:

```
$ crushtool --decompile crush.map > crush.txt
```

and adding the following line to the rule:

```
step set_choose_tries 100
```

The relevant part of of the `crush.txt` file should look something like:

```
rule erasurepool {
        ruleset 1
        type erasure
        min_size 3
        max_size 20
        step set_chooseleaf_tries 5
        step set_choose_tries 100
        step take default
        step chooseleaf indep 0 type host
        step emit
}
```

It can then be compiled and tested again:

```
$ crushtool --compile crush.txt -o better-crush.map
```

When all mappings succeed, an histogram of the number of tries that were necessary to find all of them can be displayed with the `--show-choose-tries` option of `crushtool`:

```
$ crushtool -i better-crush.map --test --show-bad-mappings \
   --show-choose-tries \
   --rule 1 \
   --num-rep 9 \
   --min-x 1 --max-x $((1024 * 1024))
...
11:        42
12:        44
13:        54
14:        45
15:        35
16:        34
17:        30
18:        25
19:        19
20:        22
21:        20
22:        17
23:        13
24:        16
25:        13
26:        11
27:        11
```

```
28:         13
29:         11
30:         10
31:          6
32:          5
33:         10
34:          3
35:          7
36:          5
37:          2
38:          5
39:          5
40:          2
41:          5
42:          4
43:          1
44:          2
45:          2
46:          3
47:          1
48:          0
...
102:          0
103:          1
104:          0
...
```

It took 11 tries to map 42 PGs, 12 tries to map 44 PGs etc. The highest number of tries is the minimum value of `set_choose_tries` that prevents bad mappings (i.e. 103 in the above output because it did not take more than 103 tries for any PG to be mapped).