CRUSH MAPS

The CRUSH algorithm determines how to store and retrieve data by computing data storage locations. CRUSH empowers Ceph clients to communicate with OSDs directly rather than through a centralized server or broker. With an algorithmically determined method of storing and retrieving data, Ceph avoids a single point of failure, a performance bottleneck, and a physical limit to its scalability.

CRUSH requires a map of your cluster, and uses the CRUSH map to pseudo-randomly store and retrieve data in OSDs with a uniform distribution of data across the cluster. For a detailed discussion of CRUSH, see CRUSH - Controlled, Scalable, Decentralized Placement of Replicated Data

CRUSH maps contain a list of OSDs, a list of 'buckets' for aggregating the devices into physical locations, and a list of rules that tell CRUSH how it should replicate data in a Ceph cluster's pools. By reflecting the underlying physical organization of the installation, CRUSH can model—and thereby address—potential sources of correlated device failures. Typical sources include physical proximity, a shared power source, and a shared network. By encoding this information into the cluster map, CRUSH placement policies can separate object replicas across different failure domains while still maintaining the desired distribution. For example, to address the possibility of concurrent failures, it may be desirable to ensure that data replicas are on devices using different shelves, racks, power supplies, controllers, and/or physical locations.

When you deploy OSDs they are automatically placed within the CRUSH map under a `host` node named with the hostname for the host they are running on. This, combined with the default CRUSH failure domain, ensures that replicas or erasure code shards are separated across hosts and a single host failure will not affect availability. For larger clusters, however, administrators should carefully consider their choice of failure domain. Separating replicas across racks, for example, is common for mid- to large-sized clusters.

CRUSH LOCATION

The location of an OSD in terms of the CRUSH map's hierarchy is referred to as a `crush location`. This location specifier takes the form of a list of key and value pairs describing a position. For example, if an OSD is in a particular row, rack, chassis and host, and is part of the 'default' CRUSH tree (this is the case for the vast majority of clusters), its crush location could be described as:

```
root=default row=a rack=a2 chassis=a2a host=a2a1
```

Note:

1. Note that the order of the keys does not matter.
2. The key name (left of =) must be a valid CRUSH `type`. By default these include root, datacenter, room, row, pod, pdu, rack, chassis and host, but those types can be customized to be anything appropriate by modifying the CRUSH map.
3. Not all keys need to be specified. For example, by default, Ceph automatically sets a `ceph-osd` daemon's location to be `root=default host=HOSTNAME` (based on the output from `hostname -s`).

The crush location for an OSD is normally expressed via the `crush location` config option being set in the `ceph.conf` file. Each time the OSD starts, it verifies it is in the correct location in the CRUSH map and, if it is not, it moved itself. To disable this automatic CRUSH map management, add the following to your configuration file in the `[osd]` section:

```
osd crush update on start = false
```

CUSTOM LOCATION HOOKS

A customized location hook can be used to generate a more complete crush location on startup. The crush location is based on, in order of preference:

1. A `crush location` option in ceph.conf.
2. A default of `root=default host=HOSTNAME` where the hostname is generated with the `hostname -s` command.

This is not useful by itself, as the OSD itself has the exact same behavior. However, a script can be written to provide additional location fields (for example, the rack or datacenter), and then the hook enabled via the config option:

```
crush location hook = /path/to/customized-ceph-crush-location
```

This hook is passed several arguments (below) and should output a single line to stdout with the CRUSH location description.:

```
--cluster CLUSTER --id ID --type TYPE
```

where the cluster name is typically 'ceph', the id is the daemon identifier (e.g., the OSD number or daemon identifier), and the daemon type is `osd`, `mds`, or similar.

For example, a simple hook that additionally specified a rack location based on a hypothetical file `/etc/rack` might be:

```
#!/bin/sh
echo "host=$(hostname -s) rack=$(cat /etc/rack) root=default"
```

CRUSH STRUCTURE

The CRUSH map consists of, loosely speaking, a hierarchy describing the physical topology of the cluster, and a set of rules defining policy about how we place data on those devices. The hierarchy has devices (`ceph-osd` daemons) at the leaves, and internal nodes corresponding to other physical features or groupings: hosts, racks, rows, datacenters, and so on. The rules describe how replicas are placed in terms of that hierarchy (e.g., 'three replicas in different racks').

DEVICES

Devices are individual `ceph-osd` daemons that can store data. You will normally have one defined here for each OSD daemon in your cluster. Devices are identified by an id (a non-negative integer) and a name, normally `osd.N` where `N` is the device id.

Devices may also have a *device class* associated with them (e.g., `hdd` or `ssd`), allowing them to be conveniently targetted by a crush rule.

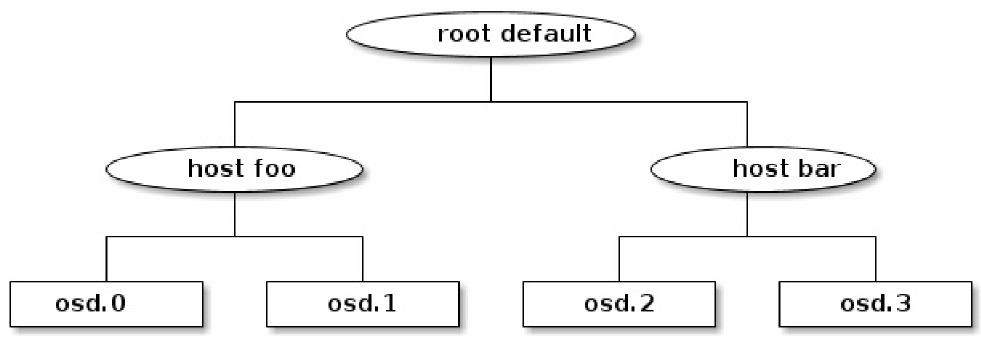
TYPES AND BUCKETS

A bucket is the CRUSH term for internal nodes in the hierarchy: hosts, racks, rows, etc. The CRUSH map defines a series of *types* that are used to describe these nodes. By default, these types include:

- osd (or device)
- host
- chassis
- rack
- row
- pdu
- pod
- room
- datacenter
- region
- root

Most clusters make use of only a handful of these types, and others can be defined as needed.

The hierarchy is built with devices (normally type `osd`) at the leaves, interior nodes with non-device types, and a root node of type `root`. For example,



Each node (device or bucket) in the hierarchy has a *weight* associated with it, indicating the relative proportion of the total data that device or hierarchy subtree should store. Weights are set at the leaves, indicating the size of the device, and automatically sum up the tree from there, such that the weight of the default node will be the total of all devices contained beneath it. Normally weights are in units of terabytes (TB).

You can get a simple view the CRUSH hierarchy for your cluster, including the weights, with:

```
ceph osd crush tree
```

RULES

Rules define policy about how data is distributed across the devices in the hierarchy.

CRUSH rules define placement and replication strategies or distribution policies that allow you to specify exactly how CRUSH places object replicas. For example, you might create a rule selecting a pair of targets for 2-way mirroring, another rule for selecting three targets in two different data centers for 3-way mirroring, and yet another rule for erasure coding over six storage devices. For a detailed discussion of CRUSH rules, refer to CRUSH - Controlled, Scalable, Decentralized Placement of Replicated Data, and more specifically to **Section 3.2**.

In almost all cases, CRUSH rules can be created via the CLI by specifying the *pool type* they will be used for (replicated or erasure coded), the *failure domain*, and optionally a *device class*. In rare cases rules must be written by hand by manually editing the CRUSH map.

You can see what rules are defined for your cluster with:

```
ceph osd crush rule ls
```

You can view the contents of the rules with:

```
ceph osd crush rule dump
```

DEVICE CLASSES

Each device can optionally have a *class* associated with it. By default, OSDs automatically set their class on startup to either *hdd*, *ssd*, or *nvme* based on the type of device they are backed by.

The device class for one or more OSDs can be explicitly set with:

```
ceph osd crush set-device-class <class> <osd-name> [...]
```

Once a device class is set, it cannot be changed to another class until the old class is unset with:

```
ceph osd crush rm-device-class <osd-name> [...]
```

This allows administrators to set device classes without the class being changed on OSD restart or by some other script.

A placement rule that targets a specific device class can be created with:

```
ceph osd crush rule create-replicated <rule-name> <root> <failure-domain> <class>
```

A pool can then be changed to use the new rule with:

```
ceph osd pool set <pool-name> crush_rule <rule-name>
```

Device classes are implemented by creating a "shadow" CRUSH hierarchy for each device class in use that contains only devices of that class. Rules can then distribute data over the shadow hierarchy. One nice thing about this approach is that it is fully backward compatible with old Ceph clients. You can view the CRUSH hierarchy with shadow items with:

```
ceph osd crush tree --show-shadow
```

WEIGHTS SETS

A *weight set* is an alternative set of weights to use when calculating data placement. The normal weights associated with each device in the CRUSH map are set based on the device size and indicate how much data we *should* be storing where. However, because CRUSH is based on a pseudorandom placement process, there is always some variation from this ideal distribution, the same way that rolling a dice sixty times will not result in rolling exactly 10 ones and 10 sixes. Weight sets allow the cluster to do a numerical optimization based on the specifics of your cluster (hierarchy, pools, etc.) to achieve a balanced distribution.

There are two types of weight sets supported:

1. A **compat** weight set is a single alternative set of weights for each device and node in the cluster. This is not well-suited for correcting for all anomalies (for example, placement groups for different pools may be different sizes and have different load levels, but will be mostly treated the same by the balancer). However, compat weight sets have the huge advantage that they are *backward compatible* with previous versions of Ceph, which means that even though weight sets were first introduced in Luminous v12.2.z, older clients (e.g., firefly) can still connect to the cluster when a compat weight set is being used to balance data.
2. A **per-pool** weight set is more flexible in that it allows placement to be optimized for each data pool. Additionally, weights can be adjusted for each position of placement, allowing the optimizer to correct for a suble skew of data toward devices with small weights relative to their peers (and effect that is usually only apparently in very large clusters but which can cause balancing problems).

When weight sets are in use, the weights associated with each node in the hierarchy is visible as a separate column (labeled either `(compat)` or the pool name) from the command:

```
ceph osd crush tree
```

When both *compat* and *per-pool* weight sets are in use, data placement for a particular pool will use its own per-pool weight set if present. If not, it will use the compat weight set if present. If neither are present, it will use the normal CRUSH weights.

Although weight sets can be set up and manipulated by hand, it is recommended that the *balancer* module be enabled to do so automatically.

MODIFYING THE CRUSH MAP

ADD/MOVE AN OSD

To add or move an OSD in the CRUSH map of a running cluster:

```
ceph osd crush set {name} {weight} root={root} [{bucket-type}={bucket-name} ...]
```

Where:

`name`

Description:	The full name of the OSD.
Type:	String
Required:	Yes
Example:	`osd.0`

`weight`

Description:	The CRUSH weight for the OSD, normally its size measure in terabytes (TB).
Type:	Double
Required:	Yes
Example:	`2.0`

`root`

Description:	The root node of the tree in which the OSD resides (normally `default`)
Type:	Key/value pair.
Required:	Yes
Example:	`root=default`

`bucket-type`

Description:	You may specify the OSD’s location in the CRUSH hierarchy.
Type:	Key/value pairs.
Required:	No
Example:	`datacenter=dc1 room=room1 row=foo rack=bar host=foo-bar-1`

The following example adds `osd.0` to the hierarchy, or moves the OSD from a previous location.

```
ceph osd crush set osd.0 1.0 root=default datacenter=dc1 room=room1 row=foo rack=bar host=foo
```

ADJUST OSD WEIGHT

To adjust an OSD’s crush weight in the CRUSH map of a running cluster, execute the following:

```
ceph osd crush reweight {name} {weight}
```

Where:

`name`

Description:	The full name of the OSD.
Type:	String
Required:	Yes
Example:	`osd.0`

`weight`

Description:	The CRUSH weight for the OSD.
Type:	Double
Required:	Yes
Example:	`2.0`

REMOVE AN OSD

To remove an OSD from the CRUSH map of a running cluster, execute the following:

```
ceph osd crush remove {name}
```

Where:

`name`

Description:	The full name of the OSD.
Type:	String
Required:	Yes
Example:	`osd.0`

ADD A BUCKET

To add a bucket in the CRUSH map of a running cluster, execute the `ceph osd crush add-bucket` command:

```
ceph osd crush add-bucket {bucket-name} {bucket-type}
```

Where:

`bucket-name`

Description:	The full name of the bucket.
Type:	String
Required:	Yes
Example:	`rack12`

`bucket-type`

Description:	The type of the bucket. The type must already exist in the hierarchy.
Type:	String
Required:	Yes
Example:	`rack`

The following example adds the `rack12` bucket to the hierarchy:

```
ceph osd crush add-bucket rack12 rack
```

MOVE A BUCKET

To move a bucket to a different location or position in the CRUSH map hierarchy, execute the following:

```
ceph osd crush move {bucket-name} {bucket-type}={bucket-name}, [...]
```

Where:

`bucket-name`

Description:	The name of the bucket to move/reposition.
Type:	String
Required:	Yes
Example:	`foo-bar-1`

`bucket-type`

Description:	You may specify the bucket's location in the CRUSH hierarchy.
Type:	Key/value pairs.
Required:	No
Example:	`datacenter=dc1 room=room1 row=foo rack=bar host=foo-bar-1`

REMOVE A BUCKET

To remove a bucket from the CRUSH map hierarchy, execute the following:

```
ceph osd crush remove {bucket-name}
```

Note: A bucket must be empty before removing it from the CRUSH hierarchy.

Where:

`bucket-name`

Description:	The name of the bucket that you'd like to remove.
Type:	String
Required:	Yes
Example:	`rack12`

The following example removes the `rack12` bucket from the hierarchy:

```
ceph osd crush remove rack12
```

CREATING A COMPAT WEIGHT SET

To create a *compat* weight set:

```
ceph osd crush weight-set create-compat
```

Weights for the compat weight set can be adjusted with:

```
ceph osd crush weight-set reweight-compat {name} {weight}
```

The compat weight set can be destroyed with:

```
ceph osd crush weight-set rm-compat
```

CREATING PER-POOL WEIGHT SETS

To create a weight set for a specific pool,:

```
ceph osd crush weight-set create {pool-name} {mode}
```

Note: Per-pool weight sets require that all servers and daemons run Luminous v12.2.z or later.

Where:

`pool-name`

Description:	The name of a RADOS pool
Type:	String
Required:	Yes
Example:	`rbd`

`mode`

Description:	Either `flat` or `positional`. A *flat* weight set has a single weight for each device or bucket. A *positional* weight set has a potentially different weight for each position in the resulting placement mapping. For example, if a pool has a replica count of 3, then a positional weight set will have three weights for each device and bucket.
Type:	String
Required:	Yes
Example:	`flat`

To adjust the weight of an item in a weight set:

```
ceph osd crush weight-set reweight {pool-name} {item-name} {weight [...]}
```

To list existing weight sets,:

```
ceph osd crush weight-set ls
```

To remove a weight set,:

```
ceph osd crush weight-set rm {pool-name}
```

CREATING A RULE FOR A REPLICATED POOL

For a replicated pool, the primary decision when creating the CRUSH rule is what the failure domain is going to be. For example, if a failure domain of `host` is selected, then CRUSH will ensure that each replica of the data is stored on a different host. If `rack` is selected, then each replica will be stored in a different rack. What failure domain you choose primarily depends on the size of your cluster and how your hierarchy is structured.

Normally, the entire cluster hierarchy is nested beneath a root node named `default`. If you have customized your hierarchy, you may want to create a rule nested at some other node in the hierarchy. It doesn't matter what type is associated with that node (it doesn't have to be a `root` node).

It is also possible to create a rule that restricts data placement to a specific *class* of device. By default, Ceph OSDs automatically classify themselves as either hdd or `ssd`, depending on the underlying type of device being used. These classes can also be customized.

To create a replicated rule,:

```
ceph osd crush rule create-replicated {name} {root} {failure-domain-type} [{class}]
```

Where:

`name`

Description:	The name of the rule
Type:	String
Required:	Yes
Example:	`rbd-rule`

`root`

Description:	The name of the node under which data should be placed.
Type:	String
Required:	Yes
Example:	`default`

`failure-domain-type`

Description:	The type of CRUSH nodes across which we should separate replicas.
Type:	String
Required:	Yes
Example:	`rack`

`class`

Description:	The device class data should be placed on.
Type:	String
Required:	No
Example:	`ssd`

CREATING A RULE FOR AN ERASURE CODED POOL

For an erasure-coded pool, the same basic decisions need to be made as with a replicated pool: what is the failure domain, what node in the hierarchy will data be placed under (usually `default`), and will placement be restricted to a specific device class. Erasure code pools are created a bit differently, however, because they need to be constructed carefully based on the erasure code being used. For this reason, you must include this information in the *erasure code profile*. A CRUSH rule will then be created from that either explicitly or automatically when the profile is used to create a pool.

The erasure code profiles can be listed with:

```
ceph osd erasure-code-profile ls
```

An existing profile can be viewed with:

```
ceph osd erasure-code-profile get {profile-name}
```

Normally profiles should never be modified; instead, a new profile should be created and used when creating a new pool or creating a new rule for an existing pool.

An erasure code profile consists of a set of key=value pairs. Most of these control the behavior of the erasure code that is encoding data in the pool. Those that begin with `crush-`, however, affect the CRUSH rule that is created.

The erasure code profile properties of interest are:

- **crush-root**: the name of the CRUSH node to place data under [default: `default`].
- **crush-failure-domain**: the CRUSH type to separate erasure-coded shards across [default: `host`].
- **crush-device-class**: the device class to place data on [default: none, meaning all devices are used].
- **k** and **m** (and, for the `lrc` plugin, **l**): these determine the number of erasure code shards, affecting the resulting CRUSH rule.

Once a profile is defined, you can create a CRUSH rule with:

```
ceph osd crush rule create-erasure {name} {profile-name}
```

DELETING RULES

Rules that are not in use by pools can be deleted with:

```
ceph osd crush rule rm {rule-name}
```

TUNABLES

Over time, we have made (and continue to make) improvements to the CRUSH algorithm used to calculate the placement of data. In order to support the change in behavior, we have introduced a series of tunable options that control whether the legacy or improved variation of the algorithm is used.

In order to use newer tunables, both clients and servers must support the new version of CRUSH. For this reason, we have created `profiles` that are named after the Ceph version in which they were introduced. For example, the `firefly` tunables are first supported in the firefly release, and will not work with older (e.g., dumpling) clients. Once a given set of tunables are changed from the legacy default behavior, the `ceph-mon` and `ceph-osd` will prevent older clients who do not support the new CRUSH features from connecting to the cluster.

ARGONAUT (LEGACY)

The legacy CRUSH behavior used by argonaut and older releases works fine for most clusters, provided there are not too many OSDs that have been marked out.

BOBTAIL (CRUSH_TUNABLES2)

The bobtail tunable profile fixes a few key misbehaviors:

- For hierarchies with a small number of devices in the leaf buckets, some PGs map to fewer than the desired number of replicas. This commonly happens for hierarchies with “host” nodes with a small number (1-3) of OSDs nested beneath each one.
- For large clusters, some small percentages of PGs map to less than the desired number of OSDs. This is more prevalent when there are several layers of the hierarchy (e.g., row, rack, host, osd).
- When some OSDs are marked out, the data tends to get redistributed to nearby OSDs instead of across the entire hierarchy.

The new tunables are:

- `choose_local_tries`: Number of local retries. Legacy value is 2, optimal value is 0.
- `choose_local_fallback_tries`: Legacy value is 5, optimal value is 0.
- `choose_total_tries`: Total number of attempts to choose an item. Legacy value was 19, subsequent testing indicates that a value of 50 is more appropriate for typical clusters. For extremely large clusters, a larger

value might be necessary.

- `chooseleaf_descend_once`: Whether a recursive chooseleaf attempt will retry, or only try once and allow the original placement to retry. Legacy default is 0, optimal value is 1.

Migration impact:

- Moving from argonaut to bobtail tunables triggers a moderate amount of data movement. Use caution on a cluster that is already populated with data.

FIREFLY (CRUSH_TUNABLES3)

The firefly tunable profile fixes a problem with the `chooseleaf` CRUSH rule behavior that tends to result in PG mappings with too few results when too many OSDs have been marked out.

The new tunable is:

- `chooseleaf_vary_r`: Whether a recursive chooseleaf attempt will start with a non-zero value of r, based on how many attempts the parent has already made. Legacy default is 0, but with this value CRUSH is sometimes unable to find a mapping. The optimal value (in terms of computational cost and correctness) is 1.

Migration impact:

- For existing clusters that have lots of existing data, changing from 0 to 1 will cause a lot of data to move; a value of 4 or 5 will allow CRUSH to find a valid mapping but will make less data move.

STRAW_CALC_VERSION TUNABLE (INTRODUCED WITH FIREFLY TOO)

There were some problems with the internal weights calculated and stored in the CRUSH map for `straw` buckets. Specifically, when there were items with a CRUSH weight of 0 or both a mix of weights and some duplicated weights CRUSH would distribute data incorrectly (i.e., not in proportion to the weights).

The new tunable is:

- `straw_calc_version`: A value of 0 preserves the old, broken internal weight calculation; a value of 1 fixes the behavior.

Migration impact:

- Moving to straw_calc_version 1 and then adjusting a straw bucket (by adding, removing, or reweighting an item, or by using the reweight-all command) can trigger a small to moderate amount of data movement *if* the cluster has hit one of the problematic conditions.

This tunable option is special because it has absolutely no impact concerning the required kernel version in the client side.

HAMMER (CRUSH_V4)

The hammer tunable profile does not affect the mapping of existing CRUSH maps simply by changing the profile. However:

- There is a new bucket type (`straw2`) supported. The new `straw2` bucket type fixes several limitations in the original `straw` bucket. Specifically, the old `straw` buckets would change some mappings that should have changed when a weight was adjusted, while `straw2` achieves the original goal of only changing mappings to or from the bucket item whose weight has changed.
- `straw2` is the default for any newly created buckets.

Migration impact:

- Changing a bucket type from `straw` to `straw2` will result in a reasonably small amount of data movement, depending on how much the bucket item weights vary from each other. When the weights are all the same no data will move, and when item weights vary significantly there will be more movement.

JEWEL (CRUSH_TUNABLES5)

The jewel tunable profile improves the overall behavior of CRUSH such that significantly fewer mappings change when an OSD is marked out of the cluster.

The new tunable is:

- chooseleaf_stable: Whether a recursive chooseleaf attempt will use a better value for an inner loop that greatly reduces the number of mapping changes when an OSD is marked out. The legacy value is 0, while the new value of 1 uses the new approach.

Migration impact:

- Changing this value on an existing cluster will result in a very large amount of data movement as almost every PG mapping is likely to change.

WHICH CLIENT VERSIONS SUPPORT CRUSH_TUNABLES

- argonaut series, v0.48.1 or later
- v0.49 or later
- Linux kernel version v3.6 or later (for the file system and RBD kernel clients)

WHICH CLIENT VERSIONS SUPPORT CRUSH_TUNABLES2

- v0.55 or later, including bobtail series (v0.56.x)
- Linux kernel version v3.9 or later (for the file system and RBD kernel clients)

WHICH CLIENT VERSIONS SUPPORT CRUSH_TUNABLES3

- v0.78 (firefly) or later
- Linux kernel version v3.15 or later (for the file system and RBD kernel clients)

WHICH CLIENT VERSIONS SUPPORT CRUSH_V4

- v0.94 (hammer) or later
- Linux kernel version v4.1 or later (for the file system and RBD kernel clients)

WHICH CLIENT VERSIONS SUPPORT CRUSH_TUNABLES5

- v10.0.2 (jewel) or later
- Linux kernel version v4.5 or later (for the file system and RBD kernel clients)

WARNING WHEN TUNABLES ARE NON-OPTIMAL

Starting with version v0.74, Ceph will issue a health warning if the current CRUSH tunables don't include all the optimal values from the `default` profile (see below for the meaning of the `default` profile). To make this warning go away, you have two options:

1. Adjust the tunables on the existing cluster. Note that this will result in some data movement (possibly as much as 10%). This is the preferred route, but should be taken with care on a production cluster where the data movement may affect performance. You can enable optimal tunables with:

   ```
   ceph osd crush tunables optimal
   ```

   If things go poorly (e.g., too much load) and not very much progress has been made, or there is a client compatibility problem (old kernel cephfs or rbd clients, or pre-bobtail librados clients), you can switch back with:

   ```
   ceph osd crush tunables legacy
   ```

2. You can make the warning go away without making any changes to CRUSH by adding the following option to your ceph.conf `[mon]` section:

   ```
   mon warn on legacy crush tunables = false
   ```

   For the change to take effect, you will need to restart the monitors, or apply the option to running monitors with:

```
ceph tell mon.\* config set mon_warn_on_legacy_crush_tunables false
```

A FEW IMPORTANT POINTS

- Adjusting these values will result in the shift of some PGs between storage nodes. If the Ceph cluster is already storing a lot of data, be prepared for some fraction of the data to move.
- The `ceph-osd` and `ceph-mon` daemons will start requiring the feature bits of new connections as soon as they get the updated map. However, already-connected clients are effectively grandfathered in, and will misbehave if they do not support the new feature.
- If the CRUSH tunables are set to non-legacy values and then later changed back to the defult values, `ceph-osd` daemons will not be required to support the feature. However, the OSD peering process requires examining and understanding old maps. Therefore, you should not run old versions of the `ceph-osd` daemon if the cluster has previously used non-legacy CRUSH values, even if the latest version of the map has been switched back to using the legacy defaults.

TUNING CRUSH

The simplest way to adjust the crush tunables is by changing to a known profile. Those are:

- `legacy`: the legacy behavior from argonaut and earlier.
- `argonaut`: the legacy values supported by the original argonaut release
- `bobtail`: the values supported by the bobtail release
- `firefly`: the values supported by the firefly release
- `hammer`: the values supported by the hammer release
- `jewel`: the values supported by the jewel release
- `optimal`: the best (ie optimal) values of the current version of Ceph
- `default`: the default values of a new cluster installed from scratch. These values, which depend on the current version of Ceph, are hard coded and are generally a mix of optimal and legacy values. These values generally match the `optimal` profile of the previous LTS release, or the most recent release for which we generally except more users to have up to date clients for.

You can select a profile on a running cluster with the command:

```
ceph osd crush tunables {PROFILE}
```

Note that this may result in some data movement.

PRIMARY AFFINITY

When a Ceph Client reads or writes data, it always contacts the primary OSD in the acting set. For set `[2, 3, 4]`, `osd.2` is the primary. Sometimes an OSD is not well suited to act as a primary compared to other OSDs (e.g., it has a slow disk or a slow controller). To prevent performance bottlenecks (especially on read operations) while maximizing utilization of your hardware, you can set a Ceph OSD's primary affinity so that CRUSH is less likely to use the OSD as a primary in an acting set.

```
ceph osd primary-affinity <osd-id> <weight>
```

Primary affinity is 1 by default (*i.e.,* an OSD may act as a primary). You may set the OSD primary range from `0-1`, where `0` means that the OSD may **NOT** be used as a primary and 1 means that an OSD may be used as a primary. When the weight is < 1, it is less likely that CRUSH will select the Ceph OSD Daemon to act as a primary.