MANUALLY EDITING A CRUSH MAP

Note: Manually editing the CRUSH map is considered an advanced administrator operation. All CRUSH changes that are necessary for the overwhelming majority of installations are possible via the standard ceph CLI and do not require manual CRUSH map edits. If you have identified a use case where manual edits *are* necessary, consider contacting the Ceph developers so that future versions of Ceph can make this unnecessary.

To edit an existing CRUSH map:

1. Get the CRUSH map.
2. Decompile the CRUSH map.
3. Edit at least one of Devices, Buckets and Rules.
4. Recompile the CRUSH map.
5. Set the CRUSH map.

For details on setting the CRUSH map rule for a specific pool, see Set Pool Values.

GET A CRUSH MAP

To get the CRUSH map for your cluster, execute the following:

```
ceph osd getcrushmap -o {compiled-crushmap-filename}
```

Ceph will output (-o) a compiled CRUSH map to the filename you specified. Since the CRUSH map is in a compiled form, you must decompile it first before you can edit it.

DECOMPILE A CRUSH MAP

To decompile a CRUSH map, execute the following:

```
crushtool -d {compiled-crushmap-filename} -o {decompiled-crushmap-filename}
```

SECTIONS

There are six main sections to a CRUSH Map.

1. **tunables:** The preamble at the top of the map described any *tunables* for CRUSH behavior that vary from the historical/legacy CRUSH behavior. These correct for old bugs, optimizations, or other changes in behavior that have been made over the years to improve CRUSH's behavior.
2. **devices:** Devices are individual `ceph-osd` daemons that can store data.
3. **types**: Bucket `types` define the types of buckets used in your CRUSH hierarchy. Buckets consist of a hierarchical aggregation of storage locations (e.g., rows, racks, chassis, hosts, etc.) and their assigned weights.
4. **buckets:** Once you define bucket types, you must define each node in the hierarchy, its type, and which devices or other nodes it containes.
5. **rules:** Rules define policy about how data is distributed across devices in the hierarchy.
6. **choose_args:** Choose_args are alternative weights associated with the hierarchy that have been adjusted to optimize data placement. A single choose_args map can be used for the entire cluster, or one can be created for each individual pool.

CRUSH MAP 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.

```
# devices
device {num} {osd.name} [class {class}]
```

For example:

```
# devices
device 0 osd.0 class ssd
device 1 osd.1 class hdd
device 2 osd.2
device 3 osd.3
```

In most cases, each device maps to a single `ceph-osd` daemon. This is normally a single storage device, a pair of devices (for example, one for data and one for a journal or metadata), or in some cases a small RAID device.

CRUSH MAP BUCKET TYPES

The second list in the CRUSH map defines 'bucket' types. Buckets facilitate a hierarchy of nodes and leaves. Node (or non-leaf) buckets typically represent physical locations in a hierarchy. Nodes aggregate other nodes or leaves. Leaf buckets represent `ceph-osd` daemons and their corresponding storage media.

Tip: The term "bucket" used in the context of CRUSH means a node in the hierarchy, i.e. a location or a piece of physical hardware. It is a different concept from the term "bucket" when used in the context of RADOS Gateway APIs.

To add a bucket type to the CRUSH map, create a new line under your list of bucket types. Enter `type` followed by a unique numeric ID and a bucket name. By convention, there is one leaf bucket and it is `type 0`; however, you may give it any name you like (e.g., osd, disk, drive, storage, etc.):

```
#types
type {num} {bucket-name}
```

For example:

```
# types
type 0 osd
type 1 host
type 2 chassis
type 3 rack
type 4 row
type 5 pdu
type 6 pod
type 7 room
type 8 datacenter
type 9 region
type 10 root
```

CRUSH MAP BUCKET HIERARCHY

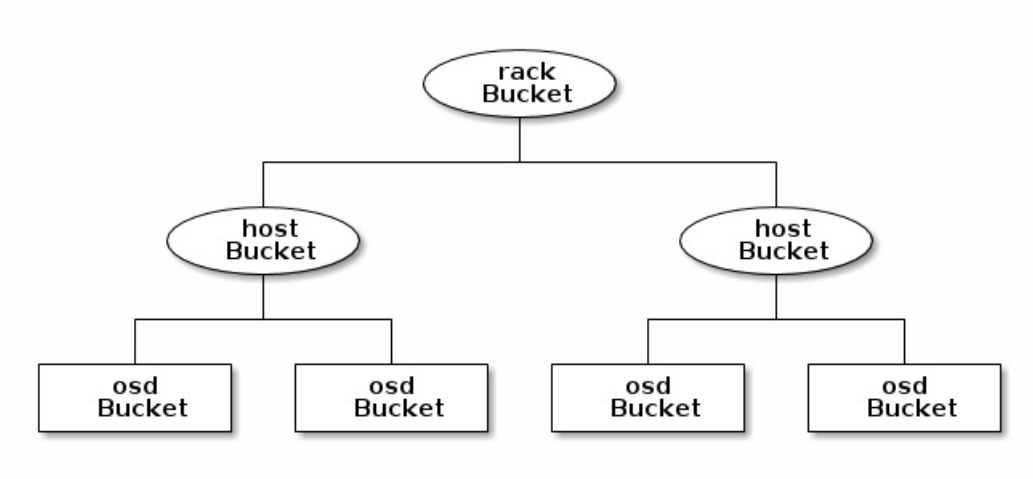
The CRUSH algorithm distributes data objects among storage devices according to a per-device weight value, approximating a uniform probability distribution. CRUSH distributes objects and their replicas according to the hierarchical cluster map you define. Your CRUSH map represents the available storage devices and the logical elements that contain them.

To map placement groups to OSDs across failure domains, a CRUSH map defines a hierarchical list of bucket types (i.e., under `#types` in the generated CRUSH map). The purpose of creating a bucket hierarchy is to segregate the leaf nodes by their failure domains, such as hosts, chassis, racks, power distribution units, pods, rows, rooms, and data centers. With the exception of the leaf nodes representing OSDs, the rest of the hierarchy is arbitrary, and you may define it according to your own needs.

We recommend adapting your CRUSH map to your firms's hardware naming conventions and using instances names that reflect the physical hardware. Your naming practice can make it easier to administer the cluster and troubleshoot problems when an OSD and/or other hardware malfunctions and the administrator need access to physical hardware.

In the following example, the bucket hierarchy has a leaf bucket named `osd`, and two node buckets named `host` and `rack`

respectively.



Note: The higher numbered `rack` bucket type aggregates the lower numbered `host` bucket type.

Since leaf nodes reflect storage devices declared under the `#devices` list at the beginning of the CRUSH map, you do not need to declare them as bucket instances. The second lowest bucket type in your hierarchy usually aggregates the devices (i.e., it's usually the computer containing the storage media, and uses whatever term you prefer to describe it, such as "node", "computer", "server," "host", "machine", etc.). In high density environments, it is increasingly common to see multiple hosts/nodes per chassis. You should account for chassis failure too–e.g., the need to pull a chassis if a node fails may result in bringing down numerous hosts/nodes and their OSDs.

When declaring a bucket instance, you must specify its type, give it a unique name (string), assign it a unique ID expressed as a negative integer (optional), specify a weight relative to the total capacity/capability of its item(s), specify the bucket algorithm (usually `straw`), and the hash (usually `0`, reflecting hash algorithm `rjenkins1`). A bucket may have one or more items. The items may consist of node buckets or leaves. Items may have a weight that reflects the relative weight of the item.

You may declare a node bucket with the following syntax:

```
[bucket-type] [bucket-name] {
        id [a unique negative numeric ID]
        weight [the relative capacity/capability of the item(s)]
        alg [the bucket type: uniform | list | tree | straw ]
        hash [the hash type: 0 by default]
        item [item-name] weight [weight]
}
```

For example, using the diagram above, we would define two host buckets and one rack bucket. The OSDs are declared as items within the host buckets:

```
host node1 {
        id -1
        alg straw
        hash 0
        item osd.0 weight 1.00
        item osd.1 weight 1.00
}

host node2 {
        id -2
        alg straw
        hash 0
        item osd.2 weight 1.00
        item osd.3 weight 1.00
}

rack rack1 {
        id -3
        alg straw
        hash 0
        item node1 weight 2.00
        item node2 weight 2.00
}
```

Note: In the foregoing example, note that the rack bucket does not contain any OSDs. Rather it contains lower level host buckets, and includes the sum total of their weight in the item entry.

Bucket Types

Ceph supports four bucket types, each representing a tradeoff between performance and reorganization efficiency. If you are unsure of which bucket type to use, we recommend using a `straw` bucket. For a detailed discussion of bucket types, refer to CRUSH - Controlled, Scalable, Decentralized Placement of Replicated Data, and more specifically to **Section 3.4**. The bucket types are:

1. **Uniform**: Uniform buckets aggregate devices with **exactly** the same weight. For example, when firms commission or decommission hardware, they typically do so with many machines that have exactly the same physical configuration (e.g., bulk purchases). When storage devices have exactly the same weight, you may use the `uniform` bucket type, which allows CRUSH to map replicas into uniform buckets in constant time. With non-uniform weights, you should use another bucket algorithm.
2. **List**: List buckets aggregate their content as linked lists. Based on the RUSH $_P$ algorithm, a list is a natural and intuitive choice for an **expanding cluster**: either an object is relocated to the newest device with some appropriate probability, or it remains on the older devices as before. The result is optimal data migration when items are added to the bucket. Items removed from the middle or tail of the list, however, can result in a significant amount of unnecessary movement, making list buckets most suitable for circumstances in which they **never (or very rarely) shrink**.
3. **Tree**: Tree buckets use a binary search tree. They are more efficient than list buckets when a bucket contains a larger set of items. Based on the RUSH $_R$ algorithm, tree buckets reduce the placement time to O(log $_n$), making them suitable for managing much larger sets of devices or nested buckets.
4. **Straw**: List and Tree buckets use a divide and conquer strategy in a way that either gives certain items precedence (e.g., those at the beginning of a list) or obviates the need to consider entire subtrees of items at all. That improves the performance of the replica placement process, but can also introduce suboptimal reorganization behavior when the contents of a bucket change due an addition, removal, or re-weighting of an item. The straw bucket type allows all items to fairly “compete” against each other for replica placement through a process analogous to a draw of straws.
5. **Straw2**: Straw2 buckets improve Straw to correctly avoid any data movement between items when neighbor weights change.

   For example the weight of item A including adding it anew or removing it completely, there will be data movement only to or from item A.

Hash

Each bucket uses a hash algorithm. Currently, Ceph supports `rjenkins1`. Enter `0` as your hash setting to select `rjenkins1`.

Weighting Bucket Items

Ceph expresses bucket weights as doubles, which allows for fine weighting. A weight is the relative difference between device capacities. We recommend using `1.00` as the relative weight for a 1TB storage device. In such a scenario, a weight of `0.5` would represent approximately 500GB, and a weight of `3.00` would represent approximately 3TB. Higher level buckets have a weight that is the sum total of the leaf items aggregated by the bucket.

A bucket item weight is one dimensional, but you may also calculate your item weights to reflect the performance of the storage drive. For example, if you have many 1TB drives where some have relatively low data transfer rate and the others have a relatively high data transfer rate, you may weight them differently, even though they have the same capacity (e.g., a weight of 0.80 for the first set of drives with lower total throughput, and 1.20 for the second set of drives with higher total throughput).

CRUSH MAP RULES

CRUSH maps support the notion of ‘CRUSH rules’, which are the rules that determine data placement for a pool. The default CRUSH map has a rule for each pool. For large clusters, you will likely create many pools where each pool may have its own non-default CRUSH rule.

Note: In most cases, you will not need to modify the default rule. When you create a new pool, by default the rule will be

set to `0`.

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**.

A rule takes the following form:

```
rule <rulename> {

        ruleset <ruleset>
        type [ replicated | erasure ]
        min_size <min-size>
        max_size <max-size>
        step take <bucket-name> [class <device-class>]
        step [choose|chooseleaf] [firstn|indep] <N> <bucket-type>
        step emit
}
```

`ruleset`

Description: A unique whole number for identifying the rule. The name `ruleset` is a carry-over from the past, when it was possible to have multiple CRUSH rules per pool.
Purpose: A component of the rule mask.
Type: Integer
Required: Yes
Default: 0

`type`

Description: Describes a rule for either a storage drive (replicated) or a RAID.
Purpose: A component of the rule mask.
Type: String
Required: Yes
Default: `replicated`
Valid Values: Currently only `replicated` and `erasure`

`min_size`

Description: If a pool makes fewer replicas than this number, CRUSH will **NOT** select this rule.
Type: Integer
Purpose: A component of the rule mask.
Required: Yes
Default: 1

`max_size`

Description: If a pool makes more replicas than this number, CRUSH will **NOT** select this rule.
Type: Integer
Purpose: A component of the rule mask.
Required: Yes
Default: 10

`step take <bucket-name> [class <device-class>]`

Description: Takes a bucket name, and begins iterating down the tree. If the `device-class` is specified, it must match a class previously used when defining a device. All devices that do not belong to the class are excluded.
Purpose: A component of the rule.
Required: Yes
Example: `step take data`

`step choose firstn {num} type {bucket-type}`

Description: Selects the number of buckets of the given type. The number is usually the number of replicas in the pool (i.e., pool size).
- If `{num} == 0`, choose `pool-num-replicas` buckets (all available).
- If `{num} > 0 && < pool-num-replicas`, choose that many buckets.
- If `{num} < 0`, it means `pool-num-replicas - {num}`.

Purpose: A component of the rule.
Prerequisite: Follows `step take` or `step choose`.
Example: `step choose firstn 1 type row`

`step chooseleaf firstn {num} type {bucket-type}`

Description: Selects a set of buckets of `{bucket-type}` and chooses a leaf node from the subtree of each bucket in the set of buckets. The number of buckets in the set is usually the number of replicas in the pool (i.e., pool size).
- If `{num} == 0`, choose `pool-num-replicas` buckets (all available).
- If `{num} > 0 && < pool-num-replicas`, choose that many buckets.
- If `{num} < 0`, it means `pool-num-replicas - {num}`.

Purpose: A component of the rule. Usage removes the need to select a device using two steps.
Prerequisite: Follows `step take` or `step choose`.
Example: `step chooseleaf firstn 0 type row`

`step emit`

Description: Outputs the current value and empties the stack. Typically used at the end of a rule, but may also be used to pick from different trees in the same rule.
Purpose: A component of the rule.
Prerequisite: Follows `step choose`.
Example: `step emit`

Important: A given CRUSH rule may be assigned to multiple pools, but it is not possible for a single pool to have multiple CRUSH rules.

PLACING DIFFERENT POOLS ON DIFFERENT OSDS:

Suppose you want to have most pools default to OSDs backed by large hard drives, but have some pools mapped to OSDs backed by fast solid-state drives (SSDs). It's possible to have multiple independent CRUSH hierarchies within the same CRUSH map. Define two hierarchies with two different root nodes–one for hard disks (e.g., "root platter") and one for SSDs (e.g., "root ssd") as shown below:

```
device 0 osd.0
device 1 osd.1
device 2 osd.2
device 3 osd.3
device 4 osd.4
device 5 osd.5
device 6 osd.6
device 7 osd.7

      host ceph-osd-ssd-server-1 {
              id -1
              alg straw
              hash 0
              item osd.0 weight 1.00
              item osd.1 weight 1.00
      }

      host ceph-osd-ssd-server-2 {
              id -2
              alg straw
              hash 0
              item osd.2 weight 1.00
              item osd.3 weight 1.00
      }

      host ceph-osd-platter-server-1 {
```

```
        id -3
        alg straw
        hash 0
        item osd.4 weight 1.00
        item osd.5 weight 1.00
}

host ceph-osd-platter-server-2 {
        id -4
        alg straw
        hash 0
        item osd.6 weight 1.00
        item osd.7 weight 1.00
}

root platter {
        id -5
        alg straw
        hash 0
        item ceph-osd-platter-server-1 weight 2.00
        item ceph-osd-platter-server-2 weight 2.00
}

root ssd {
        id -6
        alg straw
        hash 0
        item ceph-osd-ssd-server-1 weight 2.00
        item ceph-osd-ssd-server-2 weight 2.00
}

rule data {
        ruleset 0
        type replicated
        min_size 2
        max_size 2
        step take platter
        step chooseleaf firstn 0 type host
        step emit
}

rule metadata {
        ruleset 1
        type replicated
        min_size 0
        max_size 10
        step take platter
        step chooseleaf firstn 0 type host
        step emit
}

rule rbd {
        ruleset 2
        type replicated
        min_size 0
        max_size 10
        step take platter
        step chooseleaf firstn 0 type host
        step emit
}

rule platter {
        ruleset 3
        type replicated
        min_size 0
        max_size 10
        step take platter
        step chooseleaf firstn 0 type host
        step emit
}

rule ssd {
        ruleset 4
        type replicated
        min_size 0
        max_size 4
```

```
        step take ssd
        step chooseleaf firstn 0 type host
        step emit
}

rule ssd-primary {
        ruleset 5
        type replicated
        min_size 5
        max_size 10
        step take ssd
        step chooseleaf firstn 1 type host
        step emit
        step take platter
        step chooseleaf firstn -1 type host
        step emit
}
```

You can then set a pool to use the SSD rule by:

```
ceph osd pool set <poolname> crush_rule ssd
```

Similarly, using the `ssd-primary` rule will cause each placement group in the pool to be placed with an SSD as the primary and platters as the replicas.

TUNING CRUSH, THE HARD WAY

If you can ensure that all clients are running recent code, you can adjust the tunables by extracting the CRUSH map, modifying the values, and reinjecting it into the cluster.

- Extract the latest CRUSH map:

  ```
  ceph osd getcrushmap -o /tmp/crush
  ```

- Adjust tunables. These values appear to offer the best behavior for both large and small clusters we tested with. You will need to additionally specify the `--enable-unsafe-tunables` argument to `crushtool` for this to work. Please use this option with extreme care.:

  ```
  crushtool -i /tmp/crush --set-choose-local-tries 0 --set-choose-local-fallback-tries 0 --
  ```

- Reinject modified map:

  ```
  ceph osd setcrushmap -i /tmp/crush.new
  ```

LEGACY VALUES

For reference, the legacy values for the CRUSH tunables can be set with:

```
crushtool -i /tmp/crush --set-choose-local-tries 2 --set-choose-local-fallback-tries 5 --set-
```

Again, the special `--enable-unsafe-tunables` option is required. Further, as noted above, be careful running old versions of the `ceph-osd` daemon after reverting to legacy values as the feature bit is not perfectly enforced.