Red Hat Ceph Storage 6

Storage Strategies Guide

Creating storage strategies for Red Hat Ceph Storage clusters

Last Updated: 2024-03-29

Red Hat Ceph Storage 6 Storage Strategies Guide

Creating storage strategies for Red Hat Ceph Storage clusters

Legal Notice

Abstract

This document provides instructions for creating storage strategies, including creating CRUSH hierarchies, estimating the number of placement groups, determining which type of storage pool to create, and managing pools. Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

Table of Contents

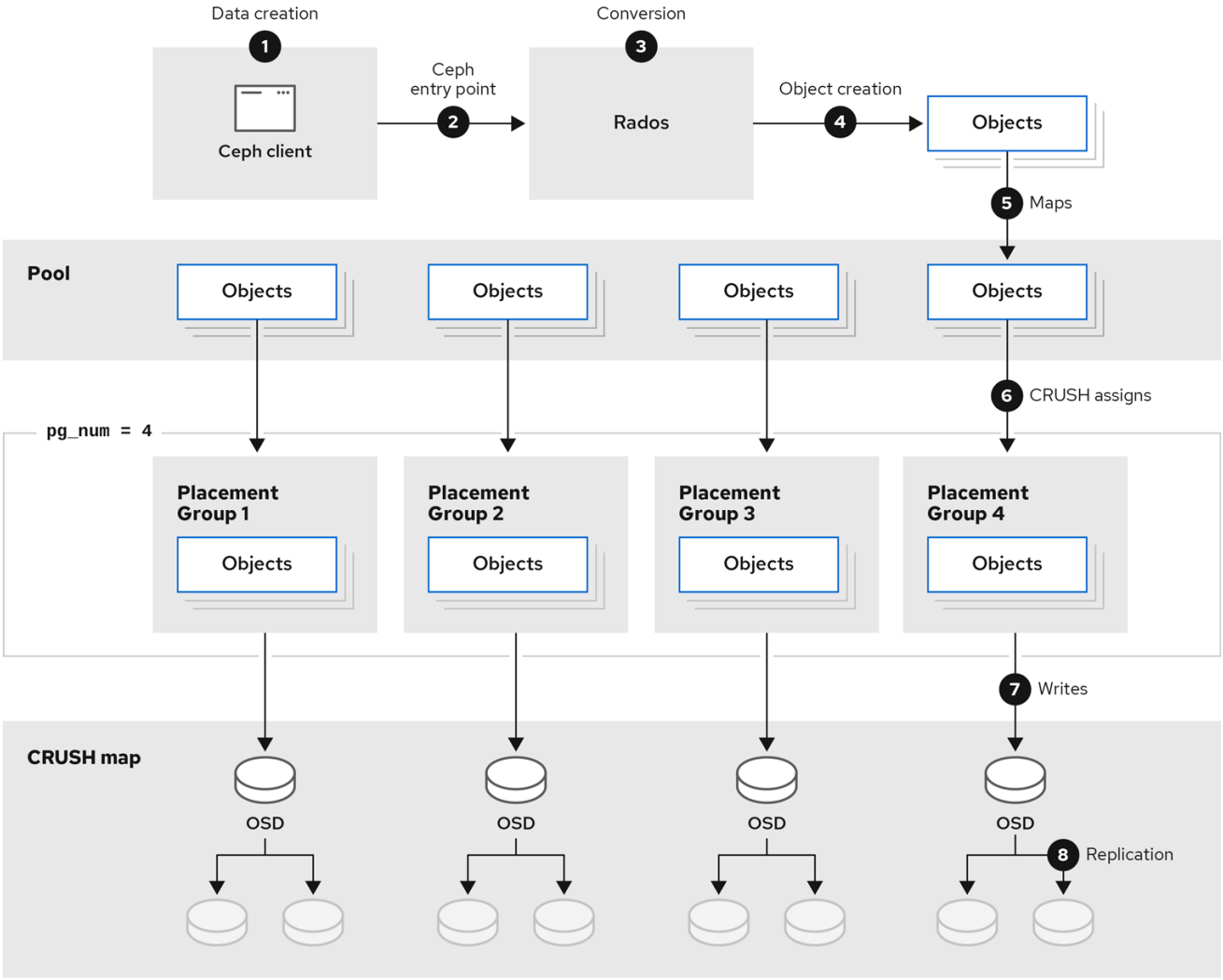
CHAPTER 1. OVERVIEW

From the perspective of a Ceph client, interacting with the Ceph storage cluster is remarkably simple:

1. Connect to the Cluster
2. Create a Pool I/O Context

This remarkably simple interface is how a Ceph client selects one of the storage strategies you define. Storage strategies are invisible to the Ceph client in all but storage capacity and performance.

The diagram below shows the logical data flow starting from the client into the Red Hat Ceph Storage cluster.



1.1. WHAT ARE STORAGE STRATEGIES?

A storage strategy is a method of storing data that serves a particular use case. For example, if you need to store volumes and images for a cloud platform like OpenStack, you might choose to store data on reasonably performant SAS drives with SSD-based journals. By contrast, if you need to store object data for an S3- or Swift-compliant gateway, you might choose to use something more economical, like SATA drives. Ceph can accommodate both scenarios in the same Ceph cluster, but you need a means of providing the SAS/SSD storage strategy to the cloud platform (for example, Glance and Cinder in OpenStack), and a means of providing SATA storage for your object store.

Storage strategies include the storage media (hard drives, SSDs, and the rest), the CRUSH maps that set up performance and failure domains for the storage media, the number of placement groups, and the pool interface. Ceph supports multiple storage strategies. Use cases, cost/benefit performance tradeoffs and data durability are the primary considerations that drive storage strategies.

1. **Use Cases:** Ceph provides massive storage capacity, and it supports numerous use cases. For example, the Ceph Block Device client is a leading storage backend for cloud platforms like OpenStack—providing limitless storage for volumes and images with high performance features like copy-on-write cloning. Likewise, Ceph can provide container-based storage for OpenShift environments. By contrast, the Ceph Object Gateway client is a leading storage backend for cloud platforms that provides RESTful S3-compliant and Swift-compliant object storage for objects like audio, bitmap, video and other data.

2. **Cost/Benefit of Performance:** Faster is better. Bigger is better. High durability is better. However, there is a price for each superlative quality, and a corresponding cost/benefit trade off. Consider the following use cases from a performance perspective: SSDs can provide very fast storage for relatively small amounts of data and journaling. Storing a database or object index might benefit from a pool of very fast SSDs, but prove too expensive for other data. SAS drives with SSD journaling provide fast performance at an economical price for volumes and images. SATA drives without SSD journaling provide cheap storage with lower overall performance. When you create a CRUSH hierarchy of OSDs, you need to consider the use case and an acceptable cost/performance trade off.

3. **Durability:** In large scale clusters, hardware failure is an expectation, not an exception. However, data loss and service interruption remain unacceptable. For this reason, data durability is very important. Ceph addresses data durability with multiple deep copies of an object or with erasure coding and multiple coding chunks. Multiple copies or multiple coding chunks present an additional cost/benefit tradeoff: it's cheaper to store fewer copies or coding chunks, but it might lead to the inability to service write requests in a degraded state. Generally, one object with two additional copies (that is, **size = 3**) or two coding chunks might allow a cluster to service writes in a degraded state while the cluster recovers. The CRUSH algorithm aids this process by ensuring that Ceph stores additional copies or coding chunks in different locations within the cluster. This ensures that the failure of a single storage device or node doesn't lead to a loss of all of the copies or coding chunks necessary to preclude data loss.

You can capture use cases, cost/benefit performance tradeoffs and data durability in a storage strategy and present it to a Ceph client as a storage pool.

IMPORTANT

Ceph's object copies or coding chunks make RAID obsolete. Do not use RAID, because Ceph already handles data durability, a degraded RAID has a negative impact on performance, and recovering data using RAID is substantially slower than using deep copies or erasure coding chunks.

1.2. CONFIGURING STORAGE STRATEGIES

Configuring storage strategies is about assigning Ceph OSDs to a CRUSH hierarchy, defining the number of placement groups for a pool, and creating a pool. The general steps are:

1. **Define a Storage Strategy:** Storage strategies require you to analyze your use case, cost/benefit performance tradeoffs and data durability. Then, you create OSDs suitable for that use case. For example, you can create SSD-backed OSDs for a high performance pool; SAS drive/SSD journal-backed OSDs for high-performance block device volumes and images; or, SATA-backed OSDs for low cost storage. Ideally, each OSD for a use case should have the same hardware configuration so that you have a consistent performance profile.

2. **Define a CRUSH Hierarchy:** Ceph rules select a node, usually the **root**, in a CRUSH hierarchy, and identify the appropriate OSDs for storing placement groups and the objects they contain. You must create a CRUSH hierarchy and a CRUSH rule for your storage strategy. CRUSH hierarchies get assigned directly to a pool by the CRUSH rule setting.

3. **Calculate Placement Groups:** Ceph shards a pool into placement groups. You do not have to manually set the number of placement groups for your pool. PG autoscaler sets an appropriate number of placement groups for your pool that remains within a healthy maximum number of placement groups in the event that you assign multiple pools to the same CRUSH rule.

4. **Create a Pool:** Finally, you must create a pool and determine whether it uses replicated or erasure-coded storage. You must set the number of placement groups for the pool, the rule for the pool and the durability, such as size or **K+M** coding chunks.

Remember, the pool is the Ceph client's interface to the storage cluster, but the storage strategy is completely transparent to the Ceph client, except for capacity and performance.

CHAPTER 2. CRUSH ADMIN OVERVIEW

The Controlled Replication Under Scalable Hashing (CRUSH) algorithm determines how to store and retrieve data by computing data storage locations.

	Any sufficiently advanced technology is indistinguishable from magic.	
	-- Arthur C. Clarke	

2.1. CRUSH INTRODUCTION

The CRUSH map for your storage cluster describes your device locations within CRUSH hierarchies and a rule for each hierarchy that determines how Ceph stores data.

The CRUSH map contains at least one hierarchy of nodes and leaves. The nodes of a hierarchy, called "buckets" in Ceph, are any aggregation of storage locations as defined by their type. For example, rows, racks, chassis, hosts, and devices. Each leaf of the hierarchy consists essentially of one of the storage devices in the list of storage devices. A leaf is always contained in one node or "bucket." A CRUSH map also has a list of rules that determine how CRUSH stores and retrieves data.

NOTE

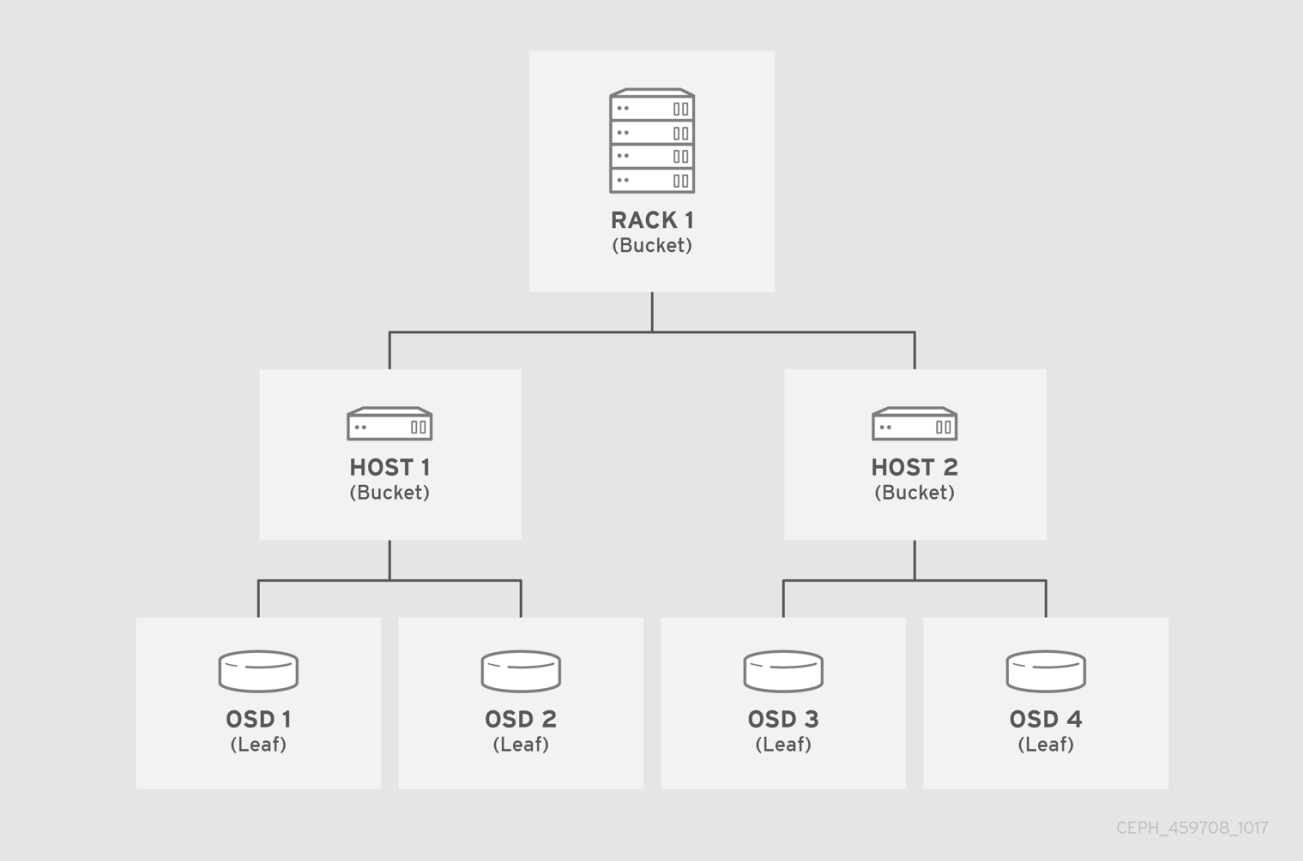
Storage devices are added to the CRUSH map when adding an OSD to the cluster.

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 or erasure-coding chunks according to the hierarchical cluster map an administrator defines. The CRUSH map represents the available storage devices and the logical buckets that contain them for the rule, and by extension each pool that uses the rule.

To map placement groups to OSDs across failure domains or performance domains, a CRUSH map defines a hierarchical list of bucket types; that is, under **types** in the generated CRUSH map. The purpose of creating a bucket hierarchy is to segregate the leaf nodes by their failure domains or performance domains or both. Failure domains include hosts, chassis, racks, power distribution units, pods, rows, rooms, and data centers. Performance domains include failure domains and OSDs of a particular configuration. For example, SSDs, SAS drives with SSD journals, SATA drives, and so on. Devices have the notion of a **class**, such as **hdd**, **ssd** and **nvme** to more rapidly build CRUSH hierarchies with a class of devices.

With the exception of the leaf nodes representing OSDs, the rest of the hierarchy is arbitrary, and you can define it according to your own needs if the default types do not suit your requirements. We recommend adapting your CRUSH map bucket types to your organization's hardware naming conventions and using instance names that reflect the physical hardware names. Your naming practice can make it easier to administer the cluster and troubleshoot problems when an OSD or other hardware malfunctions and the administrator needs remote or physical access to the host or other hardware.

In the following example, the bucket hierarchy has four leaf buckets (**osd 1-4**), two node buckets (**host 1-2**) and one rack node (**rack 1**).



Since leaf nodes reflect storage devices declared under the **devices** list at the beginning of the CRUSH map, there is no need to declare them as bucket instances. The second lowest bucket type in the hierarchy usually aggregates the devices; that is, it is usually the computer containing the storage media, and uses whatever term administrators prefer to describe it, such as "node", "computer", "server," "host", "machine", and so on. In high density environments, it is increasingly common to see multiple hosts/nodes per card and per chassis. Make sure to account for card and chassis failure too, for example, the need to pull a card or chassis if a node fails can result in bringing down numerous hosts/nodes and their OSDs.

When declaring a bucket instance, specify its type, give it a unique name as a string, assign it an optional unique ID expressed as a negative integer, specify a weight relative to the total capacity or capability of its items, specify the bucket algorithm such as **straw2**, and the hash that is usually **0** reflecting hash algorithm **rjenkins1**. A bucket can have one or more items. The items can consist of node buckets or leaves. Items can have a weight that reflects the relative weight of the item.

2.1.1. Dynamic data placement

Ceph Clients and Ceph OSDs both use the CRUSH map and the CRUSH algorithm.

- **Ceph Clients:** By distributing CRUSH maps to Ceph clients, CRUSH empowers Ceph clients to communicate with OSDs directly. This means that Ceph clients avoid a centralized object look-up table that could act as a single point of failure, a performance bottleneck, a connection limitation at a centralized look-up server and a physical limit to the storage cluster's scalability.

- **Ceph OSDs:** By distributing CRUSH maps to Ceph OSDs, Ceph empowers OSDs to handle replication, backfilling and recovery. This means that the Ceph OSDs handle storage of object replicas (or coding chunks) on behalf of the Ceph client. It also means that Ceph OSDs know enough about the cluster to re-balance the cluster (backfilling) and recover from failures dynamically.

2.1.2. CRUSH failure domain

Having multiple object replicas or **M** erasure coding chunks helps prevent data loss, but it is not sufficient to address high availability. By reflecting the underlying physical organization of the Ceph Storage Cluster, CRUSH can model—and thereby address—potential sources of correlated device failures. By encoding the cluster's topology into the cluster map, CRUSH placement policies can separate object replicas or erasure coding chunks across different failure domains while still maintaining the desired pseudo-random distribution. For example, to address the possibility of concurrent failures, it might be desirable to ensure that data replicas or erasure coding chunks are on devices using different shelves, racks, power supplies, controllers or physical locations. This helps to prevent data loss and allows the cluster to operate in a degraded state.

2.1.3. CRUSH performance domain

Ceph can support multiple hierarchies to separate one type of hardware performance profile from another type of hardware performance profile. For example, CRUSH can create one hierarchy for hard disk drives and another hierarchy for SSDs. Performance domains—hierarchies that take the performance profile of the underlying hardware into consideration—are increasingly popular due to the need to support different performance characteristics. Operationally, these are just CRUSH maps with more than one **root** type bucket. Use case examples include:

- **Object Storage:** Ceph hosts that serve as an object storage back end for S3 and Swift interfaces might take advantage of less expensive storage media such as SATA drives that might not be suitable for VMs—reducing the cost per gigabyte for object storage, while separating more economical storage hosts from more performing ones intended for storing volumes and images on cloud platforms. HTTP tends to be the bottleneck in object storage systems.

- **Cold Storage**: Systems designed for cold storage—infrequently accessed data, or data retrieval with relaxed performance requirements—might take advantage of less expensive storage media and erasure coding. However, erasure coding might require a bit of additional RAM and CPU, and thus differ in RAM and CPU requirements from a host used for object storage or VMs.

- **SSD-backed Pools:** SSDs are expensive, but they provide significant advantages over hard disk drives. SSDs have no seek time and they provide high total throughput. In addition to using SSDs for journaling, a cluster can support SSD-backed pools. Common use cases include high performance SSD pools. For example, it is possible to map the **.rgw.buckets.index** pool for the Ceph Object Gateway to SSDs instead of SATA drives.

A CRUSH map supports the notion of a device **class**. Ceph can discover aspects of a storage device and automatically assign a class such as **hdd**, **ssd** or **nvme**. However, CRUSH is not limited to these defaults. For example, CRUSH hierarchies might also be used to separate different types of workloads. For example, an SSD might be used for a journal or write-ahead log, a bucket index or for raw object storage. CRUSH can support different device classes, such as **ssd-bucket-index** or **ssd-object-storage** so Ceph does not use the same storage media for different workloads—making performance more predictable and consistent.

Behind the scenes, Ceph generates a crush root for each device-class. These roots should only be modified by setting or changing device classes on OSDs. You can view the generated roots using the following command:

Example

```
[ceph: root@host01 /]# ceph osd crush tree --show-shadow
ID   CLASS  WEIGHT    TYPE NAME
-24    ssd   4.54849  root default~ssd
```

```
-19   ssd   0.90970      host ceph01~ssd
  8   ssd   0.90970         osd.8
-20   ssd   0.90970      host ceph02~ssd
  7   ssd   0.90970         osd.7
-21   ssd   0.90970      host ceph03~ssd
  3   ssd   0.90970         osd.3
-22   ssd   0.90970      host ceph04~ssd
  5   ssd   0.90970         osd.5
-23   ssd   0.90970      host ceph05~ssd
  6   ssd   0.90970         osd.6
 -2   hdd  50.94173   root default~hdd
 -4   hdd   7.27739      host ceph01~hdd
 10   hdd   7.27739         osd.10
-12   hdd  14.55478      host ceph02~hdd
  0   hdd   7.27739         osd.0
 12   hdd   7.27739         osd.12
 -6   hdd  14.55478      host ceph03~hdd
  4   hdd   7.27739         osd.4
 11   hdd   7.27739         osd.11
-10   hdd   7.27739      host ceph04~hdd
  1   hdd   7.27739         osd.1
 -8   hdd   7.27739      host ceph05~hdd
  2   hdd   7.27739         osd.2
 -1         55.49022   root default
 -3          8.18709      host ceph01
 10   hdd   7.27739         osd.10
  8   ssd   0.90970         osd.8
-11         15.46448      host ceph02
  0   hdd   7.27739         osd.0
 12   hdd   7.27739         osd.12
  7   ssd   0.90970         osd.7
 -5         15.46448      host ceph03
  4   hdd   7.27739         osd.4
 11   hdd   7.27739         osd.11
  3   ssd   0.90970         osd.3
 -9          8.18709      host ceph04
  1   hdd   7.27739         osd.1
  5   ssd   0.90970         osd.5
 -7          8.18709      host ceph05
  2   hdd   7.27739         osd.2
  6   ssd   0.90970         osd.6
```

2.2. CRUSH HIERARCHY

The CRUSH map is a directed acyclic graph, so it can accommodate multiple hierarchies, for example, performance domains. The easiest way to create and modify a CRUSH hierarchy is with the Ceph CLI; however, you can also decompile a CRUSH map, edit it, recompile it, and activate it.

When declaring a bucket instance with the Ceph CLI, you must specify its type and give it a unique string name. Ceph automatically assigns a bucket ID, sets the algorithm to **straw2**, sets the hash to **0** reflecting **rjenkins1** and sets a weight. When modifying a decompiled CRUSH map, assign the bucket 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 **straw2**), and the hash (usually **0**, reflecting hash algorithm **rjenkins1**).

A bucket can have one or more items. The items can consist of node buckets (for example, racks, rows, hosts) or leaves (for example, an OSD disk). Items can have a weight that reflects the relative weight of the item.

When modifying a decompiled CRUSH map, you can declare a node bucket with the following syntax:

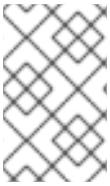
```
[bucket-type] [bucket-name] {
    id [a unique negative numeric ID]
    weight [the relative capacity/capability of the item(s)]
    alg [the bucket type: uniform | list | tree | straw2 ]
    hash [the hash type: 0 by default]
    item [item-name] weight [weight]
}
```

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 straw2
    hash 0
    item osd.0 weight 1.00
    item osd.1 weight 1.00
}

host node2 {
    id -2
    alg straw2
    hash 0
    item osd.2 weight 1.00
    item osd.3 weight 1.00
}

rack rack1 {
    id -3
    alg straw2
    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.

2.2.1. CRUSH location

A CRUSH location is the position of an OSD in terms of the CRUSH map's hierarchy. When you express a CRUSH location on the command line interface, a CRUSH location specifier takes the form of a list of name/value pairs describing the OSD's position. For example, if an OSD is in a particular row, rack, chassis and host, and is part of the **default** CRUSH tree, its crush location could be described as:

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

Note:

1. 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**. You might edit the CRUSH map to change the types to suit your needs.
3. You do not need to specify all the buckets/keys. 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**).

2.2.2. Adding a bucket

To add a bucket instance to your CRUSH hierarchy, specify the bucket name and its type. Bucket names must be unique in the CRUSH map.

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

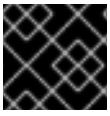
If you plan to use multiple hierarchies, for example, for different hardware performance profiles, consider naming buckets based on their type of hardware or use case.

For example, you could create a hierarchy for solid state drives (**ssd**), a hierarchy for SAS disks with SSD journals (**hdd-journal**), and another hierarchy for SATA drives (**hdd**):

```
ceph osd crush add-bucket ssd-root root
ceph osd crush add-bucket hdd-journal-root root
ceph osd crush add-bucket hdd-root root
```

The Ceph CLI outputs:

```
added bucket ssd-root type root to crush map
added bucket hdd-journal-root type root to crush map
added bucket hdd-root type root to crush map
```



IMPORTANT

Using colons (:) in bucket names is not supported.

Add an instance of each bucket type you need for your hierarchy. The following example demonstrates adding buckets for a row with a rack of SSD hosts and a rack of hosts for object storage.

```
ceph osd crush add-bucket ssd-row1 row
ceph osd crush add-bucket ssd-row1-rack1 rack
ceph osd crush add-bucket ssd-row1-rack1-host1 host
ceph osd crush add-bucket ssd-row1-rack1-host2 host
ceph osd crush add-bucket hdd-row1 row
ceph osd crush add-bucket hdd-row1-rack2 rack
ceph osd crush add-bucket hdd-row1-rack1-host1 host
```

```
ceph osd crush add-bucket hdd-row1-rack1-host2 host
ceph osd crush add-bucket hdd-row1-rack1-host3 host
ceph osd crush add-bucket hdd-row1-rack1-host4 host
```

Once you have completed these steps, view your tree.

```
ceph osd tree
```

Notice that the hierarchy remains flat. You must move your buckets into a hierarchical position after you add them to the CRUSH map.

2.2.3. Moving a bucket

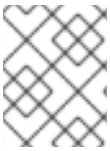
When you create your initial cluster, Ceph has a default CRUSH map with a root bucket named **default** and your initial OSD hosts appear under the **default** bucket. When you add a bucket instance to your CRUSH map, it appears in the CRUSH hierarchy, but it does not necessarily appear under a particular bucket.

To move a bucket instance to a particular location in your CRUSH hierarchy, specify the bucket name and its type. For example:

```
ceph osd crush move ssd-row1 root=ssd-root
ceph osd crush move ssd-row1-rack1 row=ssd-row1
ceph osd crush move ssd-row1-rack1-host1 rack=ssd-row1-rack1
ceph osd crush move ssd-row1-rack1-host2 rack=ssd-row1-rack1
```

Once you have completed these steps, you can view your tree.

```
ceph osd tree
```



NOTE

You can also use **ceph osd crush create-or-move** to create a location while moving an OSD.

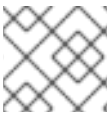
2.2.4. Removing a bucket

To remove a bucket instance from your CRUSH hierarchy, specify the bucket name. For example:

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

Or:

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



NOTE

The bucket must be empty in order to remove it.

If you are removing higher level buckets (for example, a root like **default**), check to see if a pool uses a CRUSH rule that selects that bucket. If so, you need to modify your CRUSH rules; otherwise, peering fails.

2.2.5. CRUSH Bucket algorithms

When you create buckets using the Ceph CLI, Ceph sets the algorithm to **straw2** by default. Ceph supports four bucket algorithms, each representing a tradeoff between performance and reorganization efficiency. If you are unsure of which bucket type to use, we recommend using a **straw2** bucket. The bucket algorithms 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 (for example, bulk purchases). When storage devices have exactly the same weight, you can 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 (Replication Under Scalable Hashing) $_{P}$ 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 listing buckets when a bucket contains a larger set of items. Based on the RUSH (Replication Under Scalable Hashing) $_{R}$ algorithm, tree buckets reduce the placement time to zero ($\log_{n}$), making them suitable for managing much larger sets of devices or nested buckets.

4. **Straw2 (default):** List and Tree buckets use a divide and conquer strategy in a way that either gives certain items precedence, for example, 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 **straw2** bucket type allows all items to fairly "compete" against each other for replica placement through a process analogous to a draw of straws.

2.3. CEPH OSDS IN CRUSH

Once you have a CRUSH hierarchy for the OSDs, add OSDs to the CRUSH hierarchy. You can also move or remove OSDs from an existing hierarchy. The Ceph CLI usage has the following values:

id

Description

The numeric ID of the OSD.

Type

Integer

Required

Yes

Example

0

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

root

Description

The name of the root bucket of the hierarchy or tree in which the OSD resides.

Type

Key-value pair.

Required

Yes

Example

root=default, **root=replicated_rule**, and so on

bucket-type

Description

One or more name-value pairs, where the name is the bucket type and the value is the bucket's name. You can specify a CRUSH location for an OSD in the CRUSH hierarchy.

Type

Key-value pairs.

Required

No

Example

datacenter=dc1 room=room1 row=foo rack=bar host=foo-bar-1

2.3.1. Viewing OSDs in CRUSH

The **ceph osd crush tree** command prints CRUSH buckets and items in a tree view. Use this command to determine a list of OSDs in a particular bucket. It will print output similar to **ceph osd tree**.

To return additional details, execute the following:

```
# ceph osd crush tree -f json-pretty
```

The command returns an output similar to the following:

```
[
    {
        "id": -2,
        "name": "ssd",
        "type": "root",
        "type_id": 10,
        "items": [
            {
                "id": -6,
                "name": "dell-per630-11-ssd",
                "type": "host",
                "type_id": 1,
                "items": [
                    {
                        "id": 6,
                        "name": "osd.6",
                        "type": "osd",
                        "type_id": 0,
                        "crush_weight": 0.099991,
                        "depth": 2
                    }
                ]
            },
            {
                "id": -7,
                "name": "dell-per630-12-ssd",
                "type": "host",
                "type_id": 1,
                "items": [
                    {
                        "id": 7,
                        "name": "osd.7",
                        "type": "osd",
                        "type_id": 0,
                        "crush_weight": 0.099991,
                        "depth": 2
                    }
                ]
            },
            {
                "id": -8,
                "name": "dell-per630-13-ssd",
                "type": "host",
                "type_id": 1,
                "items": [
                    {
                        "id": 8,
                        "name": "osd.8",
                        "type": "osd",
```

```
                "type_id": 0,
                "crush_weight": 0.099991,
                "depth": 2
            }
        ]
    }
  ]
},
{
    "id": -1,
    "name": "default",
    "type": "root",
    "type_id": 10,
    "items": [
        {
            "id": -3,
            "name": "dell-per630-11",
            "type": "host",
            "type_id": 1,
            "items": [
                {
                    "id": 0,
                    "name": "osd.0",
                    "type": "osd",
                    "type_id": 0,
                    "crush_weight": 0.449997,
                    "depth": 2
                },
                {
                    "id": 3,
                    "name": "osd.3",
                    "type": "osd",
                    "type_id": 0,
                    "crush_weight": 0.289993,
                    "depth": 2
                }
            ]
        },
        {
            "id": -4,
            "name": "dell-per630-12",
            "type": "host",
            "type_id": 1,
            "items": [
                {
                    "id": 1,
                    "name": "osd.1",
                    "type": "osd",
                    "type_id": 0,
                    "crush_weight": 0.449997,
                    "depth": 2
                },
                {
                    "id": 4,
                    "name": "osd.4",
                    "type": "osd",
```

```
                "type_id": 0,
                "crush_weight": 0.289993,
                "depth": 2
            }
        ]
    },
    {
        "id": -5,
        "name": "dell-per630-13",
        "type": "host",
        "type_id": 1,
        "items": [
            {
                "id": 2,
                "name": "osd.2",
                "type": "osd",
                "type_id": 0,
                "crush_weight": 0.449997,
                "depth": 2
            },
            {
                "id": 5,
                "name": "osd.5",
                "type": "osd",
                "type_id": 0,
                "crush_weight": 0.289993,
                "depth": 2
            }
        ]
    }
  ]
}
]
```

2.3.2. Adding an OSD to CRUSH

Adding a Ceph OSD to a CRUSH hierarchy is the final step before you might start an OSD (rendering it **up** and **in**) and Ceph assigns placement groups to the OSD.

You must prepare a Ceph OSD before you add it to the CRUSH hierarchy. Deployment utilities, such as the Ceph Orchestrator, can perform this step for you. For example creating a Ceph OSD on a single node:

Syntax

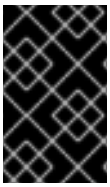
```
ceph orch daemon add osd HOST:_DEVICE_,[DEVICE]
```

The CRUSH hierarchy is notional, so the **ceph osd crush add** command allows you to add OSDs to the CRUSH hierarchy wherever you wish. The location you specify *should* reflect its actual location. If you specify at least one bucket, the command places the OSD into the most specific bucket you specify, *and* it moves that bucket underneath any other buckets you specify.

To add an OSD to a CRUSH hierarchy:

Syntax

```
ceph osd crush add ID_OR_NAME WEIGHT [BUCKET_TYPE=BUCKET_NAME ...]
```

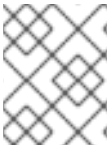


IMPORTANT

If you specify only the root bucket, the command attaches the OSD directly to the root. However, CRUSH rules expect OSDs to be inside of hosts or chassis, and host or chassis *should* be inside of other buckets reflecting your cluster topology.

The following example adds **osd.0** to the hierarchy:

```
ceph osd crush add osd.0 1.0 root=default datacenter=dc1 room=room1 row=foo rack=bar host=foo-bar-1
```



NOTE

You can also use **ceph osd crush set** or **ceph osd crush create-or-move** to add an OSD to the CRUSH hierarchy.

2.3.3. Moving an OSD within a CRUSH Hierarchy

If the storage cluster topology changes, you can move an OSD in the CRUSH hierarchy to reflect its actual location.

IMPORTANT

Moving an OSD in the CRUSH hierarchy means that Ceph will recompute which placement groups get assigned to the OSD, potentially resulting in significant redistribution of data.

To move an OSD within the CRUSH hierarchy:

Syntax

```
ceph osd crush set ID_OR_NAME WEIGHT root=POOL_NAME
[BUCKET_TYPE=BUCKET_NAME...]
```

NOTE

You can also use **ceph osd crush create-or-move** to move an OSD within the CRUSH hierarchy.

2.3.4. Removing an OSD from a CRUSH Hierarchy

Removing an OSD from a CRUSH hierarchy is the first step when you want to remove an OSD from your cluster. When you remove the OSD from the CRUSH map, CRUSH recomputes which OSDs get the placement groups and data re-balances accordingly. See Adding/Removing OSDs for additional details.

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

Syntax

```
ceph osd crush remove NAME
```

2.4. DEVICE CLASS

Ceph's CRUSH map provides extraordinary flexibility in controlling data placement. This is one of Ceph's greatest strengths. Early Ceph deployments used hard disk drives almost exclusively. Today, Ceph clusters are frequently built with multiple types of storage devices: HDD, SSD, NVMe, or even various classes of the foregoing. For example, it is common in Ceph Object Gateway deployments to have storage policies where clients can store data on slower HDDs and other storage policies for storing data on fast SSDs. Ceph Object Gateway deployments might even have a pool backed by fast SSDs for bucket indices. Additionally, OSD nodes also frequently have SSDs used exclusively for journals or write-ahead logs that do NOT appear in the CRUSH map. These complex hardware scenarios historically required manually editing the CRUSH map, which can be time-consuming and tedious. It is not required to have different CRUSH hierarchies for different classes of storage devices.

CRUSH rules work in terms of the CRUSH hierarchy. However, if different classes of storage devices reside in the same hosts, the process becomes more complicated—requiring users to create multiple CRUSH hierarchies for each class of device, and then disable the **osd crush update on start** option that automates much of the CRUSH hierarchy management. Device classes eliminate this tediousness by telling the CRUSH rule what class of device to use, dramatically simplifying CRUSH management tasks.

NOTE

The **ceph osd tree** command has a column reflecting a device class.

2.4.1. Setting a device class

To set a device class for an OSD, execute the following:

Syntax

```
ceph osd crush set-device-class  CLASS OSD_ID [OSD_ID..]
```

Example

```
[ceph: root@host01 /]# ceph osd crush set-device-class hdd osd.0 osd.1
[ceph: root@host01 /]# ceph osd crush set-device-class ssd osd.2 osd.3
[ceph: root@host01 /]# ceph osd crush set-device-class bucket-index osd.4
```

NOTE

Ceph might assign a class to a device automatically. However, class names are simply arbitrary strings. There is no requirement to adhere to **hdd**, **ssd** or **nvme**. In the foregoing example, a device class named **bucket-index** might indicate an SSD device that a Ceph Object Gateway pool uses exclusively bucket index workloads. To change a device class that was already set, use **ceph osd crush rm-device-class** first.

2.4.2. Removing a device class

To remove a device class for an OSD, execute the following:

Syntax

```
ceph osd crush rm-device-class CLASS OSD_ID [OSD_ID..]
```

Example

```
[ceph: root@host01 /]# ceph osd crush rm-device-class hdd osd.0 osd.1
[ceph: root@host01 /]# ceph osd crush rm-device-class ssd osd.2 osd.3
[ceph: root@host01 /]# ceph osd crush rm-device-class bucket-index osd.4
```

2.4.3. Renaming a device class

To rename a device class for all OSDs that use that class, execute the following:

Syntax

```
ceph osd crush class rename OLD_NAME NEW_NAME
```

Example

```
[ceph: root@host01 /]# ceph osd crush class rename hdd sas15k
```

2.4.4. Listing a device class

To list device classes in the CRUSH map, execute the following:

Syntax

```
ceph osd crush class ls
```

The output will look something like this:

Example

```
[
    "hdd",
    "ssd",
    "bucket-index"
]
```

2.4.5. Listing OSDs of a device class

To list all OSDs that belong to a particular class, execute the following:

Syntax

```
ceph osd crush class ls-osd CLASS
```

Example

```
[ceph: root@host01 /]# ceph osd crush class ls-osd hdd
```

The output is simply a list of OSD numbers. For example:

```
0
1
2
3
4
5
6
```

2.4.6. Listing CRUSH Rules by Class

To list all crush rules that reference the same class, execute the following:

Syntax

```
ceph osd crush rule ls-by-class CLASS
```

Example

```
[ceph: root@host01 /]# ceph osd crush rule ls-by-class hdd
```

2.5. CRUSH WEIGHTS

The CRUSH algorithm assigns a weight value in terabytes (by convention) per OSD device with the objective of approximating a uniform probability distribution for write requests that assign new data objects to PGs and PGs to OSDs. For this reason, as a best practice, we recommend creating CRUSH hierarchies with devices of the same type and size, and assigning the same weight. We also recommend using devices with the same I/O and throughput characteristics so that you will also have uniform performance characteristics in your CRUSH hierarchy, even though performance characteristics do not affect data distribution.

Since using uniform hardware is not always practical, you might incorporate OSD devices of different sizes and use a relative weight so that Ceph will distribute more data to larger devices and less data to smaller devices.

2.5.1. Setting CRUSH weights of OSDs

To set an OSD CRUSH weight in Terabytes within the CRUSH map, execute the following command

```
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. This should be the size of the OSD in Terabytes, where **1.0** is 1 Terabyte.

Type

Double

Required

Yes

Example

2.0

This setting is used when creating an OSD or adjusting the CRUSH weight immediately after adding the OSD. It usually does not change over the life of the OSD.

2.5.2. Setting a Bucket's OSD Weights

Using **ceph osd crush reweight** can be time-consuming. You can set (or reset) all Ceph OSD weights under a bucket (row, rack, node, and so on) by executing:

Syntax

```
osd crush reweight-subtree NAME
```

Where,

name is the name of the CRUSH bucket.

2.5.3. Set an OSD's in Weight

For the purposes of **ceph osd in** and **ceph osd out**, an OSD is either **in** the cluster or **out** of the cluster. That is how a monitor records an OSD's status. However, even though an OSD is **in** the cluster, it might be experiencing a malfunction such that you do not want to rely on it as much until you fix it (for example, replace a storage drive, change out a controller, and so on).

You can increase or decrease the **in** weight of a particular OSD (that is, without changing its weight in Terabytes) by executing:

Syntax

```
ceph osd reweight ID WEIGHT
```

Where:

- **id** is the OSD number.

- **weight** is a range from 0.0-1.0, where **0** is not **in** the cluster (that is, it does not have any PGs assigned to it) and 1.0 is **in** the cluster (that is, the OSD receives the same number of PGs as other OSDs).

2.5.4. Setting the OSDs weight by utilization

CRUSH is designed to approximate a uniform probability distribution for write requests that assign new data objects PGs and PGs to OSDs. However, a cluster might become imbalanced anyway. This can happen for a number of reasons. For example:

- **Multiple Pools:** You can assign multiple pools to a CRUSH hierarchy, but the pools might have different numbers of placement groups, size (number of replicas to store), and object size characteristics.

- **Custom Clients:** Ceph clients such as block device, object gateway and filesystem share data from their clients and stripe the data as objects across the cluster as uniform-sized smaller RADOS objects. So except for the foregoing scenario, CRUSH usually achieves its goal. However, there is another case where a cluster can become imbalanced: namely, using **librados** to store data without normalizing the size of objects. This scenario can lead to imbalanced clusters (for example, storing 100 1 MB objects and 10 4 MB objects will make a few OSDs have more data than the others).

- **Probability:** A uniform distribution will result in some OSDs with more PGs and some with less. For clusters with a large number of OSDs, the statistical outliers will be further out.

You can reweight OSDs by utilization by executing the following:

Syntax

```
ceph osd reweight-by-utilization [THRESHOLD_] [WEIGHT_CHANGE_AMOUNT] [NUMBER_OF_OSDS] [--no-increasing]
```

Example

```
[ceph: root@host01 /]# ceph osd test-reweight-by-utilization 110 .5 4 --no-increasing
```

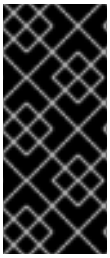
Where:

- **threshold** is a percentage of utilization such that OSDs facing higher data storage loads will receive a lower weight and thus fewer PGs assigned to them. The default value is **120**, reflecting 120%. Any value from **100+** is a valid threshold. Optional.

- **weight_change_amount** is the amount to change the weight. Valid values are greater than **0.0 - 1.0**. The default value is **0.05**. Optional.

- **number_of_OSDs** is the maximum number of OSDs to reweight. For large clusters, limiting the number of OSDs to reweight prevents significant rebalancing. Optional.

- **no-increasing** is **off** by default. Increasing the osd weight is allowed when using the **reweight-by-utilization** or **test-reweight-by-utilization** commands. If this option is used with these commands, it prevents the OSD weight from increasing, even if the OSD is underutilized. Optional.



IMPORTANT

Executing **reweight-by-utilization** is recommended and somewhat inevitable for large clusters. Utilization rates might change over time, and as your cluster size or hardware changes, the weightings might need to be updated to reflect changing utilization. If you elect to reweight by utilization, you might need to re-run this command as utilization, hardware or cluster size change.

Executing this or other weight commands that assign a weight will override the weight assigned by this command (for example, **osd reweight-by-utilization**, **osd crush weight**, **osd weight**, **in** or **out**).

2.5.5. Setting an OSD's Weight by PG distribution

In CRUSH hierarchies with a smaller number of OSDs, it's possible for some OSDs to get more PGs than other OSDs, resulting in a higher load. You can reweight OSDs by PG distribution to address this situation by executing the following:

Syntax

```
osd reweight-by-pg POOL_NAME
```

Where:

- **poolname** is the name of the pool. Ceph will examine how the pool assigns PGs to OSDs and reweight the OSDs according to this pool's PG distribution. Note that multiple pools could be assigned to the same CRUSH hierarchy. Reweighting OSDs according to one pool's distribution could have unintended effects for other pools assigned to the same CRUSH hierarchy if they do not have the same size (number of replicas) and PGs.

2.5.6. Recalculating a CRUSH Tree's weights

CRUSH tree buckets should be the sum of their leaf weights. If you manually edit the CRUSH map weights, you should execute the following to ensure that the CRUSH bucket tree accurately reflects the sum of the leaf OSDs under the bucket.

Syntax

```
osd crush reweight-all
```

2.6. 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 (for example, 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. :

Syntax

```
ceph osd primary-affinity OSD_ID WEIGHT
```

Primary affinity is **1** by default (*that is,* an OSD might act as a primary). You might set the OSD primary range from **0-1**, where **0** means that the OSD might **NOT** be used as a primary and **1** means that an OSD

might 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.

2.7. CRUSH RULES

CRUSH rules define how a Ceph client selects buckets and the primary OSD within them to store objects, and how the primary OSD selects buckets and the secondary OSDs to store replicas or coding chunks. For example, you might create a rule that selects a pair of target OSDs backed by SSDs for two object replicas, and another rule that selects three target OSDs backed by SAS drives in different data centers for three replicas.

A rule takes the following form:

```
rule <rulename> {

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

id

Description

A unique whole number for identifying the rule.

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 erasure coded.

Purpose

A component of the rule mask.

Type

String

Required

Yes

Default

replicated

Valid Values

Currently only **replicated**

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 <class-name>]

Description

Takes a bucket name, and begins iterating down the tree.

Purpose

A component of the rule.

Required

Yes

Example

step take datastep take data class ssd

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 (that is, 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

Follow **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 (that is, 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 might 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

firstn versus indep

Description

Controls the replacement strategy CRUSH uses when OSDs are marked down in the CRUSH map. If this rule is to be used with replicated pools it should be **firstn** and if it is for erasure-coded pools it should be **indep**.

Example

You have a PG stored on OSDs 1, 2, 3, 4, 5 in which 3 goes down.. In the first scenario, with the **firstn** mode, CRUSH adjusts its calculation to select 1 and 2, then selects 3 but discovers it is down, so it retries and selects 4 and 5, and then goes on to select a new OSD 6. The final CRUSH mapping change is from *1, 2, 3, 4, 5* to *1, 2, 4, 5, 6* . In the second scenario, with **indep** mode on an erasure-coded pool, CRUSH attempts to select the failed OSD 3, tries again and picks out 6, for a final transformation from *1, 2, 3, 4, 5* to *1, 2, 6, 4, 5* .

IMPORTANT

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

2.7.1. Listing CRUSH rules

To list CRUSH rules from the command line, execute the following:

Syntax

```
ceph osd crush rule list
ceph osd crush rule ls
```

2.7.2. Dumping CRUSH rules

To dump the contents of a specific CRUSH rule, execute the following:

Syntax

```
ceph osd crush rule dump NAME
```

2.7.3. Adding CRUSH rules

To add a CRUSH rule, you must specify a rule name, the root node of the hierarchy you wish to use, the type of bucket you want to replicate across (for example, 'rack', 'row', and so on and the mode for choosing the bucket.

Syntax

```
ceph osd crush rule create-simple RUENAME ROOT BUCKET_NAME FIRSTN_OR_INDEP
```

Ceph creates a rule with **chooseleaf** and one bucket of the type you specify.

Example

```
[ceph: root@host01 /]# ceph osd crush rule create-simple deleteme default host firstn
```

Create the following rule:

```
{ "id": 1,
  "rule_name": "deleteme",
  "type": 1,
  "min_size": 1,
```

```
"max_size": 10,
"steps": [
      { "op": "take",
        "item": -1,
        "item_name": "default"},
      { "op": "chooseleaf_firstn",
        "num": 0,
        "type": "host"},
      { "op": "emit"}]}
```

2.7.4. Creating CRUSH rules for replicated pools

To create a CRUSH rule for a replicated pool, execute the following:

Syntax

```
ceph osd crush rule create-replicated NAME ROOT FAILURE_DOMAIN CLASS
```

Where:

- **<name>**: The name of the rule.
- **<root>**: The root of the CRUSH hierarchy.
- **<failure-domain>**: The failure domain. For example: **host** or **rack**.
- **<class>**: The storage device class. For example: **hdd** or **ssd**.

Example

```
[ceph: root@host01 /]# ceph osd crush rule create-replicated fast default host ssd
```

2.7.5. Creating CRUSH rules for erasure coded pools

To add a CRUSH rule for use with an erasure coded pool, you might specify a rule name and an erasure code profile.

Syntax

```
ceph osd crush rule create-erasure RULE_NAME PROFILE_NAME
```

2.7.6. Removing CRUSH rules

To remove a rule, execute the following and specify the CRUSH rule name:

Syntax

```
ceph osd crush rule rm NAME
```

2.8. CRUSH TUNABLES OVERVIEW

The Ceph project has grown exponentially with many changes and many new features. Beginning with

the first commercially supported major release of Ceph, v0.48 (Argonaut), Ceph provides the ability to adjust certain parameters of the CRUSH algorithm, that is, the settings are not frozen in the source code.

A few important points to consider:

- Adjusting CRUSH values might 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 receive an updated map. However, already-connected clients are effectively grandfathered in, and will misbehave if they do not support the new feature. Make sure when you upgrade your Ceph Storage Cluster daemons that you also update your Ceph clients.
- If the CRUSH tunables are set to non-legacy values and then later changed back to the legacy 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.

2.8.1. CRUSH tuning

Before you tune CRUSH, you should ensure that all Ceph clients and all Ceph daemons use the same version. If you have recently upgraded, ensure that you have restarted daemons and reconnected clients.

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

- **legacy**: The legacy behavior from v0.47 (pre-Argonaut) and earlier.
- **argonaut**: The legacy values supported by v0.48 (Argonaut) release.
- **bobtail**: The values supported by the v0.56 (Bobtail) release.
- **firefly**: The values supported by the v0.80 (Firefly) release.
- **hammer**: The values supported by the v0.94 (Hammer) release.
- **jewel**: The values supported by the v10.0.2 (Jewel) release.
- **optimal**: The current best values.
- **default**: The current default values for a new cluster.

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

Syntax

```
# ceph osd crush tunables PROFILE
```

NOTE

This might result in some data movement.

Generally, you should set the CRUSH tunables after you upgrade, or if you receive a warning. Starting with version v0.74, Ceph issues a health warning if the CRUSH tunables are not set to their optimal values, the optimal values are the default as of v0.73.

You can remove the warning by adjusting 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 might affect performance. You can enable optimal tunables with:

```
ceph osd crush tunables optimal
```

If things go poorly (for example, 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 to an earlier profile:

Syntax

```
ceph osd crush tunables PROFILE
```

For example, to restore the pre-v0.48 (Argonaut) values, execute:

Example

```
[ceph: root@host01 /]# ceph osd crush tunables legacy
```

2.8.2. CRUSH tuning, 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 --set-choose-total-tries 50 -o /tmp/crush.new
  ```

- Reinject modified map:

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

2.8.3. CRUSH 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-choose-total-tries 19 --set-chooseleaf-descend-once 0 --set-chooseleaf-vary-r 0 -o /tmp/crush.legacy
```

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.

2.9. EDIT A CRUSH MAP

Generally, modifying your CRUSH map at runtime with the Ceph CLI is more convenient than editing the CRUSH map manually. However, there are times when you might choose to edit it, such as changing the default bucket types, or using a bucket algorithm other than **straw2**.

To edit an existing CRUSH map:

1. Getting the CRUSH map.
2. Decompiling the CRUSH map.
3. Edit at least one of the devices, and buckets and rules.
4. Compile the CRUSH map
5. Setting a CRUSH map.

To activate a CRUSH Map rule for a specific pool, identify the common rule number and specify that rule number for the pool when creating the pool.

2.9.1. Getting the CRUSH map

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

Syntax

```
ceph osd getcrushmap -o COMPILED_CRUSHMAP_FILENAME
```

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

2.9.2. Decompiling the CRUSH map

To decompile a CRUSH map, execute the following:

Syntax

```
crushtool -d COMPILED_CRUSHMAP_FILENAME -o DECOMPILED_CRUSHMAP_FILENAME
```

Ceph decompiles (-d) the compiled CRUSH map and send the output (-o) to the file name you specified.

2.9.3. Setting a CRUSH map

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

Syntax

```
ceph osd setcrushmap -i COMPILED_CRUSHMAP_FILENAME
```

Ceph inputs the compiled CRUSH map of the file name you specified as the CRUSH map for the cluster.

2.9.4. Compiling the CRUSH map

To compile a CRUSH map, execute the following:

Syntax

```
crushtool -c DECOMPILED_CRUSHMAP_FILENAME -o COMPILED_CRUSHMAP_FILENAME
```

Ceph will store a compiled CRUSH map to the file name you specified.

2.10. CRUSH STORAGE STRATEGIES EXAMPLES

If 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). CRUSH can handle these scenarios easily.

Use device classes. The process is simple to add a class to each device.

Syntax

```
ceph osd crush set-device-class CLASS OSD_ID [OSD_ID]
```

Example

```
[ceph:root@host01 /]# ceph osd crush set-device-class hdd osd.0 osd.1 osd.4 osd.5
[ceph:root@host01 /]# ceph osd crush set-device-class ssd osd.2 osd.3 osd.6 osd.7
```

Then, create rules to use the devices.

Syntax

```
ceph osd crush rule create-replicated RULENAME ROOT FAILURE_DOMAIN_TYPE
DEVICE_CLASS
```

Example

```
[ceph:root@host01 /]# ceph osd crush rule create-replicated cold default host hdd
[ceph:root@host01 /]# ceph osd crush rule create-replicated hot default host ssd
```

Finally, set pools to use the rules.

Syntax

```
ceph osd pool set POOL_NAME crush_rule RULENAME
```

Example

```
[ceph:root@host01 /]# ceph osd pool set cold crush_rule hdd
[ceph:root@host01 /]# ceph osd pool set hot crush_rule ssd
```

There is no need to manually edit the CRUSH map, because one hierarchy can serve multiple classes of devices.

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

 host ceph-osd-server-1 {
    id -1
    alg straw2
    hash 0
    item osd.0 weight 1.00
    item osd.1 weight 1.00
    item osd.2 weight 1.00
    item osd.3 weight 1.00
 }

 host ceph-osd-server-2 {
    id -2
    alg straw2
    hash 0
    item osd.4 weight 1.00
    item osd.5 weight 1.00
    item osd.6 weight 1.00
    item osd.7 weight 1.00
 }

 root default {
    id -3
    alg straw2
    hash 0
    item ceph-osd-server-1 weight 4.00
    item ceph-osd-server-2 weight 4.00
 }


 rule cold {
    ruleset 0
    type replicated
    min_size 2
    max_size 11
    step take default class hdd
    step chooseleaf firstn 0 type host
    step emit
 }


 rule hot {
    ruleset 1
    type replicated
    min_size 2
```

```
    max_size 11
    step take default class ssd
    step chooseleaf firstn 0 type host
    step emit
}
```

CHAPTER 3. PLACEMENT GROUPS

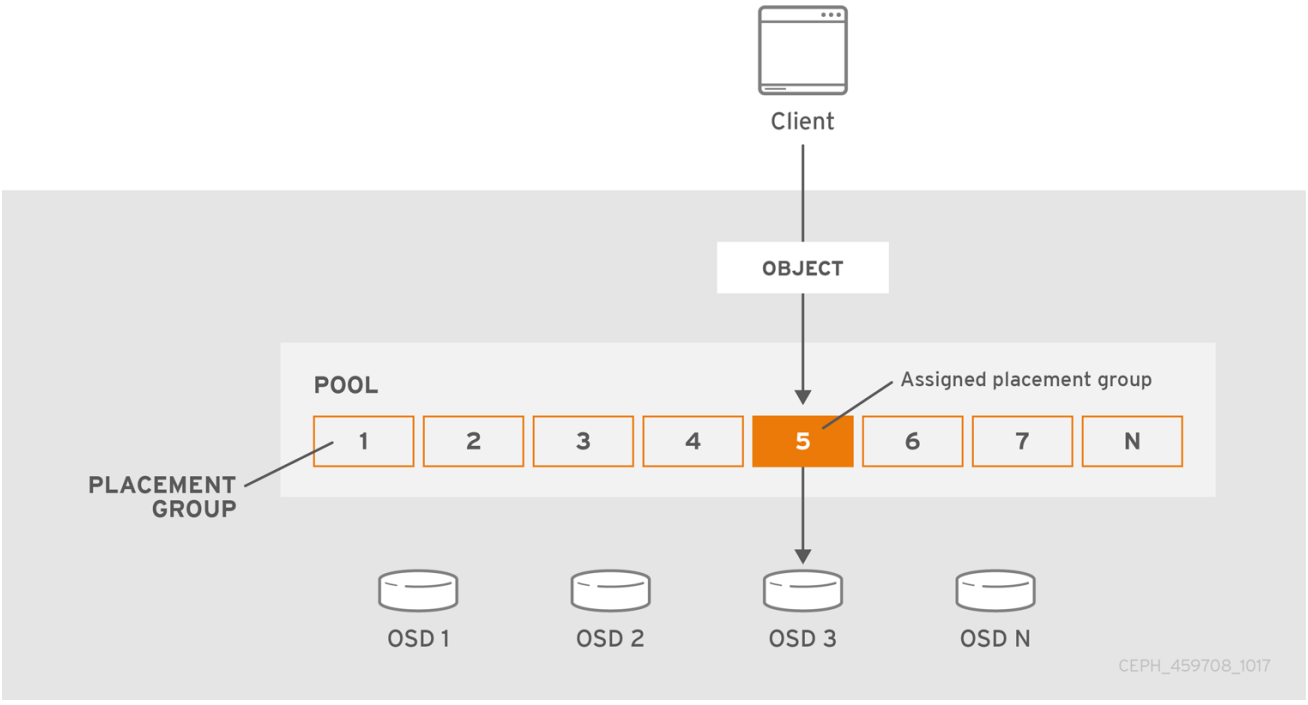
Placement Groups (PGs) are invisible to Ceph clients, but they play an important role in Ceph Storage Clusters.

A Ceph Storage Cluster might require many thousands of OSDs to reach an exabyte level of storage capacity. Ceph clients store objects in pools, which are a logical subset of the overall cluster. The number of objects stored in a pool might easily run into the millions and beyond. A system with millions of objects or more cannot realistically track placement on a per-object basis and still perform well. Ceph assigns objects to placement groups, and placement groups to OSDs to make re-balancing dynamic and efficient.

	All problems in computer science can be solved by another level of indirection, except of course for the problem of too many indirections.	
		-- David Wheeler

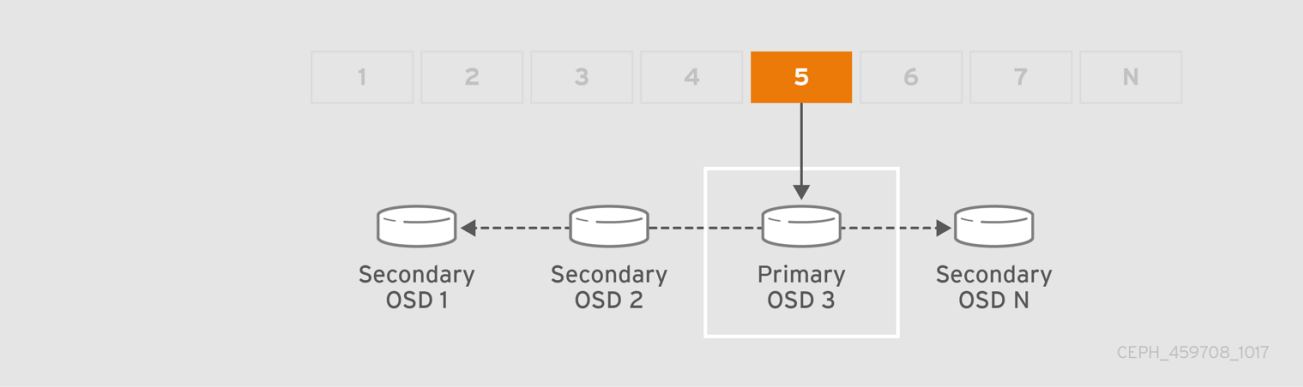
3.1. ABOUT PLACEMENT GROUPS

Tracking object placement on a per-object basis within a pool is computationally expensive at scale. To facilitate high performance at scale, Ceph subdivides a pool into placement groups, assigns each individual object to a placement group, and assigns the placement group to a **primary** OSD. If an OSD fails or the cluster re-balances, Ceph can move or replicate an entire placement group—that is, all of the objects in the placement groups—without having to address each object individually. This allows a Ceph cluster to re-balance or recover efficiently.



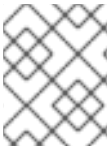
When CRUSH assigns a placement group to an OSD, it calculates a series of OSDs—the first being the primary. The **osd_pool_default_size** setting minus **1** for replicated pools, and the number of coding chunks **M** for erasure-coded pools determine the number of OSDs storing a placement group that can fail without losing data permanently. Primary OSDs use CRUSH to identify the secondary OSDs and copy the placement group's contents to the secondary OSDs. For example, if CRUSH assigns an object to a placement group, and the placement group is assigned to OSD 5 as the primary OSD, if CRUSH calculates that OSD 1 and OSD 8 are secondary OSDs for the placement group, the primary OSD 5 will

copy the data to OSDs 1 and 8. By copying data on behalf of clients, Ceph simplifies the client interface and reduces the client workload. The same process allows the Ceph cluster to recover and rebalance dynamically.



When the primary OSD fails and gets marked out of the cluster, CRUSH assigns the placement group to another OSD, which receives copies of objects in the placement group. Another OSD in the **Up Set** will assume the role of the primary OSD.

When you increase the number of object replicas or coding chunks, CRUSH will assign each placement group to additional OSDs as required.



NOTE

PGs do not own OSDs. CRUSH assigns many placement groups to each OSD pseudo-randomly to ensure that data gets distributed evenly across the cluster.

3.2. PLACEMENT GROUP STATES

When you check the storage cluster's status with the **ceph -s** or **ceph -w** commands, Ceph reports on the status of the placement groups (PGs). A PG has one or more states. The optimum state for PGs in the PG map is an **active + clean** state.

activating

The PG is peered, but not yet active.

active

Ceph processes requests to the PG.

backfill_toofull

A backfill operation is waiting because the destination OSD is over the backfillfull ratio.

backfill_unfound

Backfill stopped due to unfound objects.

backfill_wait

The PG is waiting in line to start backfill.

backfilling

Ceph is scanning and synchronizing the entire contents of a PG instead of inferring what contents need to be synchronized from the logs of recent operations. Backfill is a special case of recovery.

clean

Ceph replicated all objects in the PG accurately.

creating

Ceph is still creating the PG.

deep

Ceph is checking the PG data against stored checksums.

degraded

Ceph has not replicated some objects in the PG accurately yet.

down

A replica with necessary data is down, so the PG is offline. A PG with less than **min_size** replicas is marked as down. Use **ceph health detail** to understand the backing OSD state.

forced_backfill

High backfill priority of that PG is enforced by user.

forced_recovery

High recovery priority of that PG is enforced by user.

incomplete

Ceph detects that a PG is missing information about writes that might have occurred, or does not have any healthy copies. If you see this state, try to start any failed OSDs that might contain the needed information. In the case of an erasure coded pool, temporarily reducing **min_size** might allow recovery.

inconsistent

Ceph detects inconsistencies in one or more replicas of an object in the PG, such as objects are the wrong size, objects are missing from one replica after recovery finished.

peering

The PG is undergoing the peering process. A peering process should clear off without much delay, but if it stays and the number of PGs in a peering state does not reduce in number, the peering might be stuck.

peered

The PG has peered, but cannot serve client IO due to not having enough copies to reach the pool's configured **min_size** parameter. Recovery might occur in this state, so the PG might heal up to **min_size** eventually.

recovering

Ceph is migrating or synchronizing objects and their replicas.

recovery_toofull

A recovery operation is waiting because the destination OSD is over its full ratio.

recovery_unfound

Recovery stopped due to unfound objects.

recovery_wait

The PG is waiting in line to start recovery.

remapped

The PG is temporarily mapped to a different set of OSDs from what CRUSH specified.

repair

Ceph is checking the PG and repairing any inconsistencies it finds, if possible.

replay

The PG is waiting for clients to replay operations after an OSD crashed.

snaptrim

Trimming snaps.

snaptrim_error

Error stopped trimming snaps.

snaptrim_wait

Queued to trim snaps.

scrubbing

Ceph is checking the PG metadata for inconsistencies.

splitting

Ceph is splitting the PG into multiple PGs.

stale

The PG is in an unknown state; the monitors have not received an update for it since the PG mapping changed.

undersized

The PG has fewer copies than the configured pool replication level.

unknown

The **ceph-mgr** has not yet received any information about the PG's state from an OSD since Ceph Manager started up.

Additional resources

- See the knowledge base *What are the possible Placement Group states in an Ceph cluster* for more information.

3.3. PLACEMENT GROUP TRADEOFFS

Data durability and data distribution among all OSDs call for more placement groups but their number should be reduced to the minimum required for maximum performance to conserve CPU and memory resources.

3.3.1. Data durability

Ceph strives to prevent the permanent loss of data. However, after an OSD fails, the risk of permanent data loss increases until the data it had is fully recovered. Permanent data loss, though rare, is still possible. The following scenario describes how Ceph could permanently lose data in a single placement group with three copies of the data:

- An OSD fails and all copies of the object it contains are lost. For all objects within a placement group stored on the OSD, the number of replicas suddenly drops from three to two.
- Ceph starts recovery for each placement group stored on the failed OSD by choosing a new OSD to re-create the third copy of all objects for each placement group.
- The second OSD containing a copy of the same placement group fails before the new OSD is fully populated with the third copy. Some objects will then only have one surviving copy.
- Ceph picks yet another OSD and keeps copying objects to restore the desired number of copies.
- The third OSD containing a copy of the same placement group fails before recovery is complete. If this OSD contained the only remaining copy of an object, the object is lost permanently.

Hardware failure isn't an exception, but an expectation. To prevent the foregoing scenario, ideally the recovery process should be as fast as reasonably possible. The size of your cluster, your hardware configuration and the number of placement groups play an important role in total recovery time.

Small clusters don't recover as quickly.

In a cluster containing 10 OSDs with 512 placement groups in a three replica pool, CRUSH will give each placement group three OSDs. Each OSD will end up hosting **(512 * 3)** / **10 = ~150** placement groups. When the first OSD fails, the cluster will start recovery for all 150 placement groups simultaneously.

It is likely that Ceph stored the remaining 150 placement groups randomly across the 9 remaining OSDs. Therefore, each remaining OSD is likely to send copies of objects to all other OSDs and also receive some new objects, because the remaining OSDs become responsible for some of the 150 placement groups now assigned to them.

The total recovery time depends upon the hardware supporting the pool. For example, in a 10 OSD cluster, if a host contains one OSD with a 1 TB SSD, and a 10 GB/s switch connects each of the 10 hosts, the recovery time will take **M** minutes. By contrast, if a host contains two SATA OSDs and a 1 GB/s switch connects the five hosts, recovery will take substantially longer. Interestingly, in a cluster of this size, the number of placement groups has almost no influence on data durability. The placement group count could be 128 or 8192 and the recovery would not be slower or faster.

However, growing the same Ceph cluster to 20 OSDs instead of 10 OSDs is likely to speed up recovery and therefore improve data durability significantly. Why? Each OSD now participates in only 75 placement groups instead of 150. The 20 OSD cluster will still require all 19 remaining OSDs to perform the same amount of copy operations in order to recover. In the 10 OSD cluster, each OSDs had to copy approximately 100 GB. In the 20 OSD cluster each OSD only has to copy 50 GB each. If the network was the bottleneck, recovery will happen twice as fast. In other words, recovery time decreases as the number of OSDs increases.

In large clusters, PG count is important!

If the exemplary cluster grows to 40 OSDs, each OSD will only host 35 placement groups. If an OSD dies, recovery time will decrease unless another bottleneck precludes improvement. However, if this cluster grows to 200 OSDs, each OSD will only host approximately 7 placement groups. If an OSD dies, recovery will happen between at most of 21 **(7 * 3)** OSDs in these placement groups: **recovery will take longer than when there were 40 OSDs, meaning the number of placement groups should be increased!**

IMPORTANT

No matter how short the recovery time, there is a chance for another OSD storing the placement group to fail while recovery is in progress.

In the 10 OSD cluster described above, if any OSD fails, then approximately 8 placement groups (that is **75 pgs** / **9 osds** being recovered) will only have one surviving copy. And if any of the 8 remaining OSDs fail, the last objects of one placement group are likely to be lost (that is **8 pgs** / **8 osds** with only one remaining copy being recovered). This is why starting with a somewhat larger cluster is preferred (for example, 50 OSDs).

When the size of the cluster grows to 20 OSDs, the number of placement groups damaged by the loss of three OSDs drops. The second OSD lost will degrade approximately 2 (that is **35 pgs** / **19 osds** being recovered) instead of 8 and the third OSD lost will only lose data if it is one of the two OSDs containing the surviving copy. In other words, if the probability of losing one OSD is **0.0001%** during the recovery time frame, it goes from **8 * 0.0001%** in the cluster with 10 OSDs to **2 * 0.0001%** in the cluster with 20 OSDs. Having 512 or 4096 placement groups is roughly equivalent in a cluster with less than 50 OSDs as far as data durability is concerned.

TIP

In a nutshell, more OSDs means faster recovery and a lower risk of cascading failures leading to the permanent loss of a placement group and its objects.

When you add an OSD to the cluster, it might take a long time to populate the new OSD with placement groups and objects. However there is no degradation of any object and adding the OSD has no impact on data durability.

3.3.2. Data distribution

Ceph seeks to avoid hot spots—that is, some OSDs receive substantially more traffic than other OSDs. Ideally, CRUSH assigns objects to placement groups evenly so that when the placement groups get assigned to OSDs (also pseudo randomly), the primary OSDs store objects such that they are evenly distributed across the cluster and hot spots and network over-subscription problems cannot develop because of data distribution.

Since CRUSH computes the placement group for each object, but does not actually know how much data is stored in each OSD within this placement group, **the ratio between the number of placement groups and the number of OSDs might influence the distribution of the data significantly.**

For instance, if there was only one placement group with ten OSDs in a three replica pool, Ceph would only use three OSDs to store data because CRUSH would have no other choice. When more placement groups are available, CRUSH is more likely to evenly spread objects across OSDs. CRUSH also evenly assigns placement groups to OSDs.

As long as there are one or two orders of magnitude more placement groups than OSDs, the distribution should be even. For instance, 256 placement groups for 3 OSDs, 512 or 1024 placement groups for 10 OSDs, and so forth.

The ratio between OSDs and placement groups usually solves the problem of uneven data distribution for Ceph clients that implement advanced features like object striping. For example, a 4 TB block device might get sharded up into 4 MB objects.

The ratio between OSDs and placement groups does not address uneven data distribution in other cases, because CRUSH does not take object size into account. Using the **librados** interface to store some relatively small objects and some very large objects can lead to uneven data distribution. For example, one million 4K objects totaling 4 GB are evenly spread among 1000 placement groups on 10 OSDs. They will use **4 GB / 10 = 400 MB** on each OSD. If one 400 MB object is added to the pool, the three OSDs supporting the placement group in which the object has been placed will be filled with **400 MB + 400 MB = 800 MB** while the seven others will remain occupied with only 400 MB.

3.3.3. Resource usage

For each placement group, OSDs and Ceph monitors need memory, network and CPU at all times, and even more during recovery. Sharing this overhead by clustering objects within a placement group is one of the main reasons placement groups exist.

Minimizing the number of placement groups saves significant amounts of resources.

3.4. PLACEMENT GROUP COUNT

The number of placement groups in a pool plays a significant role in how a cluster peers, distributes data and rebalances. Small clusters don't see as many performance improvements compared to large clusters by increasing the number of placement groups. However, clusters that have many pools accessing the

same OSDs might need to carefully consider PG count so that Ceph OSDs use resources efficiently.

TIP

Red Hat recommends 100 to 200 PGs per OSD.

3.4.1. Placement group calculator

The placement group (PG) calculator calculates the number of placement groups for you and addresses specific use cases. The PG calculator is especially helpful when using Ceph clients like the Ceph Object Gateway where there are many pools typically using the same rule (CRUSH hierarchy). You might still calculate PGs manually using the guidelines in Placement group count for small clusters and Calculating placement group count. However, the PG calculator is the preferred method of calculating PGs.

See Ceph Placement Groups (PGs) per Pool Calculator on the Red Hat Customer Portal for details.

3.4.2. Configuring default placement group count

When you create a pool, you also create a number of placement groups for the pool. If you don't specify the number of placement groups, Ceph will use the default value of **8**, which is unacceptably low. You can increase the number of placement groups for a pool, but we recommend setting reasonable default values too.

```
osd pool default pg num = 100
osd pool default pgp num = 100
```

You need to set both the number of placement groups (total), and the number of placement groups used for objects (used in PG splitting). They should be equal.

3.4.3. Placement group count for small clusters

Small clusters don't benefit from large numbers of placement groups. As the number of OSDs increase, choosing the right value for **pg_num** and **pgp_num** becomes more important because it has a significant influence on the behavior of the cluster as well as the durability of the data when something goes wrong (that is the probability that a catastrophic event leads to data loss). It is important to use the PG calculator with small clusters.

3.4.4. Calculating placement group count

If you have more than 50 OSDs, we recommend approximately 50-100 placement groups per OSD to balance out resource usage, data durability and distribution. If you have less than 50 OSDs, choosing among the PG Count for Small Clusters is ideal. For a single pool of objects, you can use the following formula to get a baseline:

$$\text{Total PGs} = \frac{(\text{OSDs} * 100)}{\text{pool size}}$$

Where **pool size** is either the number of replicas for replicated pools or the **K+M** sum for erasure coded pools (as returned by **ceph osd erasure-code-profile get**).

You should then check if the result makes sense with the way you designed your Ceph cluster to maximize data durability, data distribution and minimize resource usage.

The result should be **rounded up to the nearest power of two.** Rounding up is optional, but recommended for CRUSH to evenly balance the number of objects among placement groups.

For a cluster with 200 OSDs and a pool size of 3 replicas, you would estimate your number of PGs as follows:

```
(200 * 100)
---------- = 6667. Nearest power of 2: 8192
    3
```

With 8192 placement groups distributed across 200 OSDs, that evaluates to approximately 41 placement groups per OSD. You also need to consider the number of pools you are likely to use in your cluster, since each pool will create placement groups too. Ensure that you have a reasonable maximum placement group count.

3.4.5. Maximum placement group count

When using multiple data pools for storing objects, you need to ensure that you balance the number of placement groups per pool with the number of placement groups per OSD so that you arrive at a reasonable total number of placement groups. The aim is to achieve reasonably low variance per OSD without taxing system resources or making the peering process too slow.

In an exemplary Ceph Storage Cluster consisting of 10 pools, each pool with 512 placement groups on ten OSDs, there are a total of 5,120 placement groups spread over ten OSDs, or 512 placement groups per OSD. That might not use too many resources depending on your hardware configuration. By contrast, if you create 1,000 pools with 512 placement groups each, the OSDs will handle ~50,000 placement groups each and it would require significantly more resources. Operating with too many placement groups per OSD can significantly reduce performance, especially during rebalancing or recovery.

The Ceph Storage Cluster has a default maximum value of 300 placement groups per OSD. You can set a different maximum value in your Ceph configuration file.

```
mon pg warn max per osd
```

TIP

Ceph Object Gateways deploy with 10-15 pools, so you might consider using less than 100 PGs per OSD to arrive at a reasonable maximum number.

3.5. AUTO-SCALING PLACEMENT GROUPS

The number of placement groups (PGs) in a pool plays a significant role in how a cluster peers, distributes data, and rebalances.

Auto-scaling the number of PGs can make managing the cluster easier. The **pg-autoscaling** command provides recommendations for scaling PGs, or automatically scales PGs based on how the cluster is being used.

- To learn more about how auto-scaling works, see Section 3.5.1, “Placement group auto-scaling”.
- To enable, or disable auto-scaling, see Section 3.5.3, “Setting placement group auto-scaling modes”.

- To view placement group scaling recommendations, see Section 3.5.4, “Viewing placement group scaling recommendations”.
- To set placement group auto-scaling, see Section 3.5.5, “Setting placement group auto-scaling”.
- To update the autoscaler globally, see Section 3.5.6, “Updating **noautoscale** flag”
- To set target pool size see, Section 3.6, “Specifying target pool size” .

3.5.1. Placement group auto-scaling

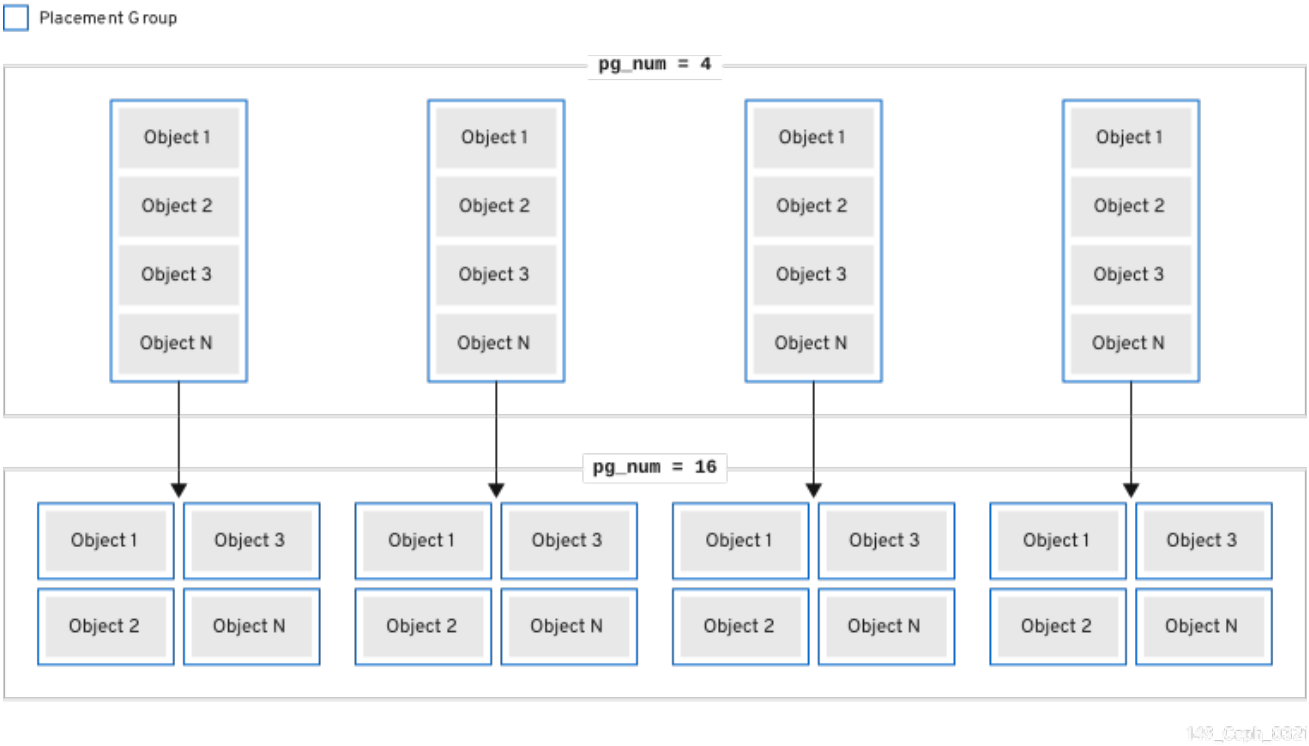
How the auto-scaler works

The auto-scaler analyzes pools and adjusts on a per-subtree basis. Because each pool can map to a different CRUSH rule, and each rule can distribute data across different devices, Ceph considers utilization of each subtree of the hierarchy independently. For example, a pool that maps to OSDs of class **ssd**, and a pool that maps to OSDs of class **hdd**, will each have optimal PG counts that depend on the number of those respective device types.

3.5.2. Placement group splitting and merging

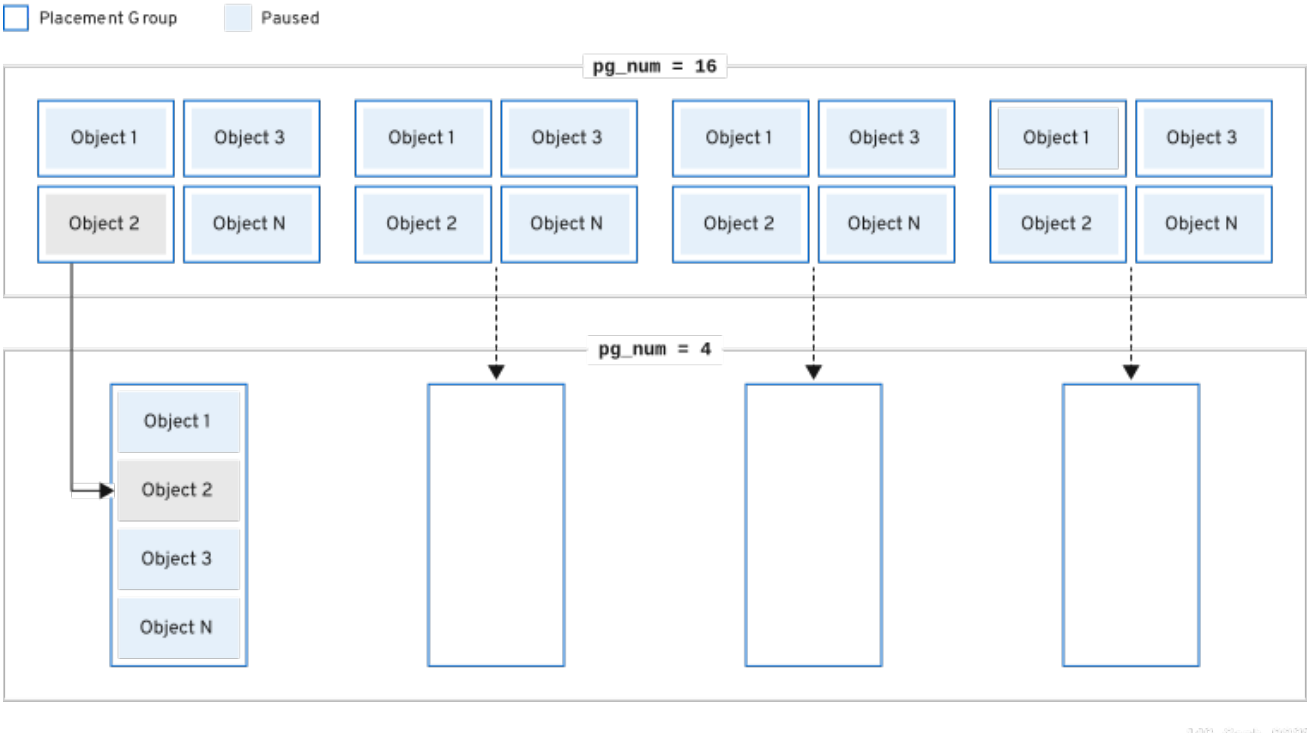
Splitting

Red Hat Ceph Storage can split existing placement groups (PGs) into smaller PGs, which increases the total number of PGs for a given pool. Splitting existing placement groups (PGs) allows a small Red Hat Ceph Storage cluster to scale over time as storage requirements increase. The PG auto-scaling feature can increase the **pg_num** value, which causes the existing PGs to split as the storage cluster expands. If the PG auto-scaling feature is disabled, then you can manually increase the **pg_num** value, which triggers the PG split process to begin. For example, increasing the **pg_num** value from **4** to **16**, will split into four pieces. Increasing the **pg_num** value will also increase the **pgp_num** value, but the **pgp_num** value increases at a gradual rate. This gradual increase is done to minimize the impact to a storage cluster’s performance and to a client’s workload, because migrating object data adds a significant load to the system. By default, Ceph queues and moves no more than 5% of the object data that is in a "misplaced" state. This default percentage can be adjusted with the **target_max_misplaced_ratio** option.



Merging

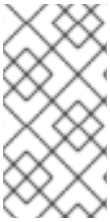
Red Hat Ceph Storage can also merge two existing PGs into a larger PG, which decreases the total number of PGs. Merging two PGs together can be useful, especially when the relative amount of objects in a pool decreases over time, or when the initial number of PGs chosen was too large. While merging PGs can be useful, it is also a complex and delicate process. When doing a merge, pausing I/O to the PG occurs, and only one PG is merged at a time to minimize the impact to a storage cluster's performance. Ceph works slowly on merging the object data until the new **pg_num** value is reached.



3.5.3. Setting placement group auto-scaling modes

Each pool in the Red Hat Ceph Storage cluster has a **pg_autoscale_mode** property for PGs that you can set to **off**, **on**, or **warn**.

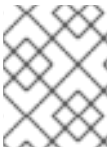
- **off**: Disables auto-scaling for the pool. It is up to the administrator to choose an appropriate PG number for each pool. Refer to the Placement group count section for more information.
- **on**: Enables automated adjustments of the PG count for the given pool.
- **warn**: Raises health alerts when the PG count needs adjustment.



NOTE

In Red Hat Ceph Storage 5 and later releases, **pg_autoscale_mode** is **on** by default. Upgraded storage clusters retain the existing **pg_autoscale_mode** setting. The **pg_auto_scale** mode is **on** for the newly created pools. PG count is automatically adjusted, and **ceph status** might display a recovering state during PG count adjustment.

The autoscaler uses the **bulk** flag to determine which pool should start with a full complement of PGs and only scales down when the usage ratio across the pool is not even. However, if the pool does not have the **bulk** flag, the pool starts with minimal PGs and only when there is more usage in the pool.



NOTE

The autoscaler identifies any overlapping roots and prevents the pools with such roots from scaling because overlapping roots can cause problems with the scaling process.

Procedure

- Enable auto-scaling on an existing pool:

  Syntax

  ```
  ceph osd pool set POOL_NAME pg_autoscale_mode on
  ```

  Example

  ```
  [ceph: root@host01 /]# ceph osd pool set testpool pg_autoscale_mode on
  ```

- Enable auto-scaling on a newly created pool:

  Syntax

  ```
  ceph config set global osd_pool_default_pg_autoscale_mode MODE
  ```

  Example

  ```
  [ceph: root@host01 /]# ceph config set global osd_pool_default_pg_autoscale_mode on
  ```

- Create a pool with the **bulk** flag:

  Syntax

```
ceph osd pool create POOL_NAME --bulk
```

Example

```
[ceph: root@host01 /]#  ceph osd pool create testpool --bulk
```

- Set or unset the **bulk** flag for an existing pool:

IMPORTANT

The values must be written as **true**, **false**, **1**, or **0**. **1** is equivalent to **true** and **0** is equivalent to **false**. If written with different capitalization, or with other content, an error is emitted.

The following is an example of the command written with the wrong syntax:

```
[ceph: root@host01 /]# ceph osd pool set ec_pool_overwrite bulk True
Error EINVAL: expecting value 'true', 'false', '0', or '1'
```

Syntax

```
ceph osd pool set POOL_NAME bulk true/false/1/0
```

Example

```
[ceph: root@host01 /]#  ceph osd pool set testpool bulk true
```

- Get the **bulk** flag of an existing pool:

Syntax

```
ceph osd pool get POOL_NAME bulk
```

Example

```
[ceph: root@host01 /]# ceph osd pool get testpool bulk
bulk: true
```

3.5.4. Viewing placement group scaling recommendations

You can view the pool, it's relative utilization and any suggested changes to the PG count in the storage cluster.

Prerequisites

- A running Red Hat Ceph Storage cluster
- Root-level access to all the nodes.

Procedure

- You can view each pool, its relative utilization, and any suggested changes to the PG count using:

```
[ceph: root@host01 /]# ceph osd pool autoscale-status
```

Output will look similar to the following:

```
POOL                     SIZE  TARGET SIZE  RATE  RAW CAPACITY   RATIO  TARGET RATIO
EFFECTIVE RATIO  BIAS  PG_NUM  NEW PG_NUM  AUTOSCALE  BULK
device_health_metrics      0               3.0        374.9G  0.0000                                  1.0       1
on         False
cephfs.cephfs.meta     24632               3.0        374.9G  0.0000                                  4.0
32            on         False
cephfs.cephfs.data         0               3.0        374.9G  0.0000                                  1.0      32
on         False
.rgw.root               1323               3.0        374.9G  0.0000                                  1.0      32
on         False
default.rgw.log         3702               3.0        374.9G  0.0000                                  1.0      32
on         False
default.rgw.control        0               3.0        374.9G  0.0000                                  1.0      32
on         False
default.rgw.meta         382               3.0        374.9G  0.0000                                  4.0       8
on         False
```

SIZE is the amount of data stored in the pool.

TARGET SIZE, if present, is the amount of data the administrator has specified they expect to eventually be stored in this pool. The system uses the larger of the two values for its calculation.

RATE is the multiplier for the pool that determines how much raw storage capacity the pool uses. For example, a **3** replica pool has a ratio of **3.0**, while a **k=4,m=2** erasure coded pool has a ratio of **1.5**.

RAW CAPACITY is the total amount of raw storage capacity on the OSDs that are responsible for storing the pool's data.

RATIO is the ratio of the total capacity that the pool is consuming, that is, ratio = size * rate / raw capacity.

TARGET RATIO, if present, is the ratio of storage the administrator has specified that they expect the pool to consume relative to other pools with target ratios set. If both target size bytes and ratio are specified, the ratio takes precedence. The default value of **TARGET RATIO** is **0** unless it was specified while creating the pool. The more the **--target_ratio** you give in a pool, the larger the PGs you are expecting the pool to have.

EFFECTIVE RATIO, is the target ratio after adjusting in two ways: 1. subtracting any capacity expected to be used by pools with target size set. 2. normalizing the target ratios among pools with target ratio set so they collectively target the rest of the space. For example, 4 pools with **target ratio** 1.0 would have an **effective ratio** of 0.25. The system uses the larger of the actual ratio and the effective ratio for its calculation.

BIAS, is used as a multiplier to manually adjust a pool's PG based on prior information about how much PGs a specific pool is expected to have. By default, the value if 1.0 unless it was specified when creating a pool. The more **--bias** you give in a pool, the larger the PGs you are expecting the pool to have.

PG_NUM is the current number of PGs for the pool, or the current number of PGs that the pool is working towards, if a **pg_num** change is in progress. **NEW PG_NUM**, if present, is the suggested number

of PGs (**pg_num**). It is always a power of 2, and is only present if the suggested value varies from the current value by more than a factor of 3.

AUTOSCALE, is the pool **pg_autoscale_mode**, and is either **on**, **off**, or **warn**.

BULK, is used to determine which pool should start out with a full complement of PGs. **BULK** only scales down when the usage ratio cross the pool is not even. If the pool does not have this flag the pool starts out with a minimal amount of PGs and only used when there is more usage in the pool.

The **BULK** values are **true**, **false**, **1**, or **0**, where **1** is equivalent to **true** and **0** is equivalent to **false**. The default value is **false**.

Set the **BULK** value either during or after pool creation.

For more information about using the bulk flag, see Creating a pool and Setting placement group auto-scaling modes.

3.5.5. Setting placement group auto-scaling

Allowing the cluster to automatically scale PGs based on cluster usage is the simplest approach to scaling PGs. Red Hat Ceph Storage takes the total available storage and the target number of PGs for the whole system, compares how much data is stored in each pool, and apportions the PGs accordingly. The command only makes changes to a pool whose current number of PGs (**pg_num**) is more than three times off from the calculated or suggested PG number.

The target number of PGs per OSD is based on the **mon_target_pg_per_osd** configurable. The default value is set to **100**.

Procedure

- To adjust **mon_target_pg_per_osd**:

  Syntax

  ```
  ceph config set global mon_target_pg_per_osd number
  ```

  For example:

  ```
  [ceph: root@host01 /]# ceph config set global mon_target_pg_per_osd 150
  ```

3.5.6. Updating noautoscale flag

If you want to enable or disable the autoscaler for all the pools at the same time, you can use the **noautoscale** global flag. This global flag is useful during upgradation of the storage cluster when some OSDs are bounced or when the cluster is under maintenance. You can set the flag before any activity and unset it once the activity is complete.

By default, the **noautoscale** flag is set to **off**. When this flag is set, then all the pools have **pg_autoscale_mode** as **off** and all the pools have the autoscaler disabled.

Prerequisites

- A running Red Hat Ceph Storage cluster

- Root-level access to all the nodes.

Procedure

1. Get the value of the **noautoscale** flag:

   Example

   ```
   [ceph: root@host01 /]# ceph osd pool get noautoscale
   ```

2. Set the **noautoscale** flag before any activity:

   Example

   ```
   [ceph: root@host01 /]# ceph osd pool set noautoscale
   ```

3. Unset the **noautoscale** flag on completion of the activity:

   Example

   ```
   [ceph: root@host01 /]# ceph osd pool unset noautoscale
   ```

3.6. SPECIFYING TARGET POOL SIZE

A newly created pool consumes a small fraction of the total cluster capacity and appears to the system that it will need a small number of PGs. However, in most cases, cluster administrators know which pools are expected to consume most of the system capacity over time. If you provide this information, known as the **target size** to Red Hat Ceph Storage, such pools can use a more appropriate number of PGs (**pg_num**) from the beginning. This approach prevents subsequent changes in **pg_num** and the overhead associated with moving data around when making those adjustments.

You can specify **target size** of a pool in these ways:

- Section 3.6.1, "Specifying target size using the absolute size of the pool"
- Section 3.6.2, "Specifying target size using the total cluster capacity"

3.6.1. Specifying target size using the absolute size of the pool

Procedure

1. Set the **target size** using the absolute size of the pool in bytes:

   ```
   ceph osd pool set pool-name target_size_bytes value
   ```

   For example, to instruct the system that **mypool** is expected to consume 100T of space:

   ```
   $ ceph osd pool set mypool target_size_bytes 100T
   ```

You can also set the target size of a pool at creation time by adding the optional **--target-size-bytes <bytes>** argument to the **ceph osd pool create** command.

3.6.2. Specifying target size using the total cluster capacity

Procedure

1. Set the **target size** using the ratio of the total cluster capacity:

   Syntax

   ```
   ceph osd pool set pool-name target_size_ratio ratio
   ```

   For Example:

   ```
   [ceph: root@host01 /]# ceph osd pool set mypool target_size_ratio 1.0
   ```

   tells the system that the pool **mypool** is expected to consume 1.0 relative to the other pools with **target_size_ratio** set. If **mypool** is the only pool in the cluster, this means an expected use of 100% of the total capacity. If there is a second pool with **target_size_ratio** as 1.0, both pools would expect to use 50% of the cluster capacity.

You can also set the target size of a pool at creation time by adding the optional **--target-size-ratio <ratio>** argument to the **ceph osd pool create** command.

NOTE

If you specify impossible target size values, for example, a capacity larger than the total cluster, or ratios that sum to more than 1.0, the cluster raises a **POOL_TARGET_SIZE_RATIO_OVERCOMMITTED** or **POOL_TARGET_SIZE_BYTES_OVERCOMMITTED** health warning.

If you specify both **target_size_ratio** and **target_size_bytes** for a pool, the cluster considers only the ratio, and raises a **POOL_HAS_TARGET_SIZE_BYTES_AND_RATIO** health warning.

3.7. PLACEMENT GROUP COMMAND LINE INTERFACE

The **ceph** CLI allows you to set and get the number of placement groups for a pool, view the PG map and retrieve PG statistics.

3.7.1. Setting number of placement groups in a pool

To set the number of placement groups in a pool, you must specify the number of placement groups at the time you create the pool. See Creating a Pool for details. Once you set placement groups for a pool, you can increase the number of placement groups (but you cannot decrease the number of placement groups). To increase the number of placement groups, execute the following:

Syntax

```
ceph osd pool set POOL_NAME pg_num PG_NUM
```

Once you increase the number of placement groups, you must also increase the number of placement groups for placement (**pgp_num**) before your cluster will rebalance. The **pgp_num** should be equal to the **pg_num**. To increase the number of placement groups for placement, execute the following:

Syntax

```
ceph osd pool set POOL_NAME pgp_num PGP_NUM
```

3.7.2. Getting number of placement groups in a pool

To get the number of placement groups in a pool, execute the following:

Syntax

```
ceph osd pool get POOL_NAME pg_num
```

3.7.3. Getting statistics for placement groups

To get the statistics for the placement groups in your storag cluster, execute the following:

Syntax

```
ceph pg dump [--format FORMAT]
```

Valid formats are **plain** (default) and **json**.

3.7.4. Getting statistics for stuck placement groups

To get the statistics for all placement groups stuck in a specified state, execute the following:

Syntax

```
ceph pg dump_stuck {inactive|unclean|stale|undersized|degraded
[inactive|unclean|stale|undersized|degraded...]} INTERVAL
```

Inactive Placement groups cannot process reads or writes because they are waiting for an OSD with the most up-to-date data to come up and in.

Unclean Placement groups contain objects that are not replicated the desired number of times. They should be recovering.

Stale Placement groups are in an unknown state - the OSDs that host them have not reported to the monitor cluster in a while (configured by **mon_osd_report_timeout**).

Valid formats are **plain** (default) and **json**. The threshold defines the minimum number of seconds the placement group is stuck before including it in the returned statistics (default 300 seconds).

3.7.5. Getting placement group maps

To get the placement group map for a particular placement group, execute the following:

Syntax

```
ceph pg map PG_ID
```

Example

```
[ceph: root@host01 /]# ceph pg map 1.6c
```

Ceph returns the placement group map, the placement group, and the OSD status:

```
osdmap e13 pg 1.6c (1.6c) -> up [1,0] acting [1,0]
```

3.7.6. Scrubbing placement groups

To scrub a placement group, execute the following:

Syntax

```
ceph pg scrub PG_ID
```

Ceph checks the primary and any replica nodes, generates a catalog of all objects in the placement group and compares them to ensure that no objects are missing or mismatched, and their contents are consistent. Assuming the replicas all match, a final semantic sweep ensures that all of the snapshot-related object metadata is consistent. Errors are reported via logs.

3.7.7. Marking unfound objects

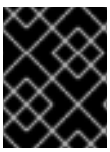
If the cluster has lost one or more objects, and you have decided to abandon the search for the lost data, you must mark the unfound objects as **lost**.

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

Currently the only supported option is "revert", which will either roll back to a previous version of the object or (if it was a new object) forget about it entirely. To mark the "unfound" objects as "lost", execute the following:

Syntax

```
ceph pg PG_ID mark_unfound_lost revert|delete
```



IMPORTANT

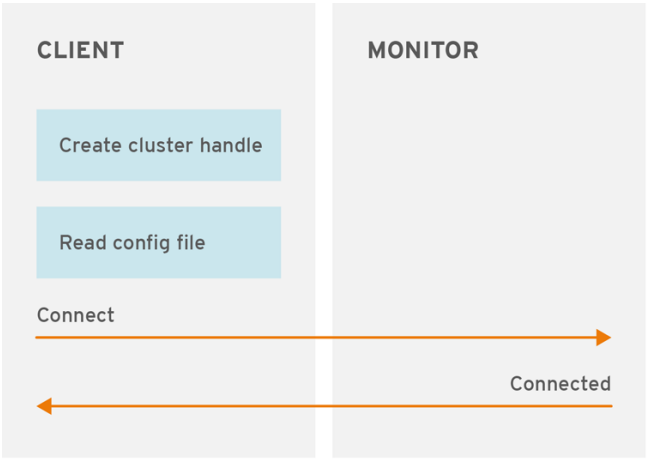
Use this feature with caution, because it might confuse applications that expect the object(s) to exist.

CHAPTER 4. POOLS OVERVIEW

Ceph clients store data in pools. When you create pools, you are creating an I/O interface for clients to store data. From the perspective of a Ceph client, that is, block device, gateway, and the rest, interacting with the Ceph storage cluster is remarkably simple: create a cluster handle and connect to the cluster; then, create an I/O context for reading and writing objects and their extended attributes.

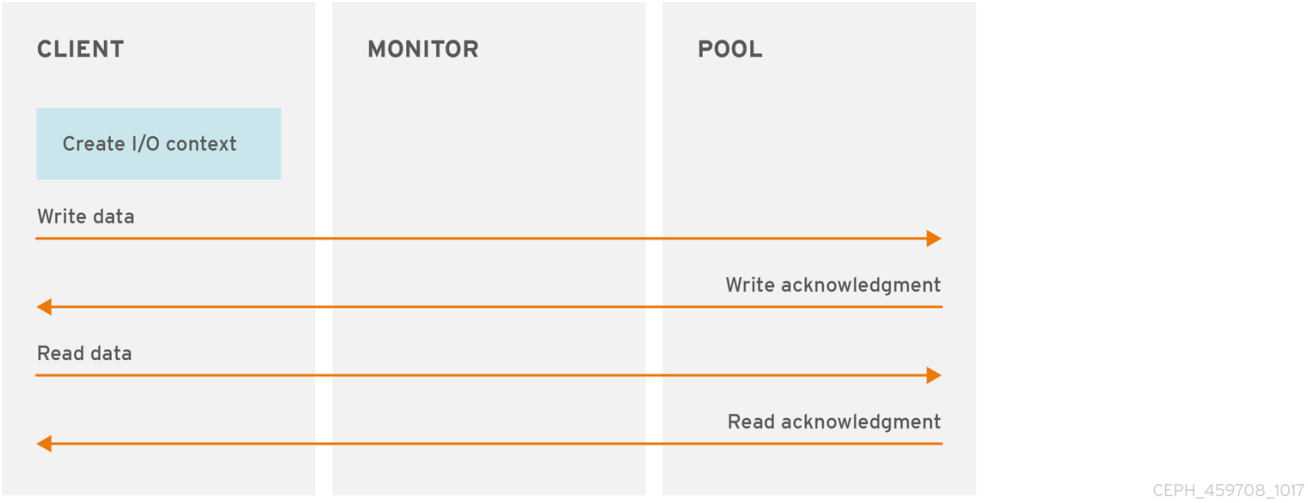
Create a Cluster Handle and Connect to the Cluster

To connect to the Ceph storage cluster, the Ceph client needs the cluster name, which is usually **ceph** by default, and an initial monitor address. Ceph clients usually retrieve these parameters using the default path for the Ceph configuration file and then read it from the file, but a user might also specify the parameters on the command line too. The Ceph client also provides a user name and secret key, authentication is **on** by default. Then, the client contacts the Ceph monitor cluster and retrieves a recent copy of the cluster map, including its monitors, OSDs and pools.



Create a Pool I/O Context

To read and write data, the Ceph client creates an I/O context to a specific pool in the Ceph storage cluster. If the specified user has permissions for the pool, the Ceph client can read from and write to the specified pool.



Ceph's architecture enables the storage cluster to provide this remarkably simple interface to Ceph clients so that clients might select one of the sophisticated storage strategies you define simply by specifying a pool name and creating an I/O context. Storage strategies are invisible to the Ceph client in

all but capacity and performance. Similarly, the complexities of Ceph clients, such as mapping objects into a block device representation or providing an S3/Swift RESTful service, are invisible to the Ceph storage cluster.

A pool provides you with:

- **Resilience**: You can set how many OSD are allowed to fail without losing data. For replicated pools, it is the desired number of copies or replicas of an object. A typical configuration stores an object and one additional copy, that is, **size = 2**, but you can determine the number of copies or replicas. For erasure coded pools, it is the number of coding chunks, that is **m=2** in the **erasure code profile**.
- **Placement Groups**: You can set the number of placement groups for the pool. A typical configuration uses approximately 50-100 placement groups per OSD to provide optimal balancing without using up too many computing resources. When setting up multiple pools, be careful to ensure you set a reasonable number of placement groups for both the pool and the cluster as a whole.
- **CRUSH Rules**: When you store data in a pool, a CRUSH rule mapped to the pool enables CRUSH to identify the rule for the placement of each object and its replicas, or chunks for erasure coded pools, in your cluster. You can create a custom CRUSH rule for your pool.
- **Quotas**: When you set quotas on a pool with **ceph osd pool set-quota** you might limit the maximum number of objects or the maximum number of bytes stored in the specified pool.

4.1. POOLS AND STORAGE STRATEGIES OVERVIEW

To manage pools, you can list, create, and remove pools. You can also view the utilization statistics for each pool.

4.2. LISTING POOL

List your cluster's pools:

Example

```
[ceph: root@host01 /]# ceph osd lspools
```

4.3. CREATING A POOL

Before creating pools, see the *Configuration Guide* for more details.

It is better to adjust the default value for the number of placement groups, as the default value does not have to suit your needs:

Example

```
[ceph: root@host01 /]# ceph config set global osd_pool_default_pg_num 250
[ceph: root@host01 /]# ceph config set global osd_pool_default_pgp_num 250
```

Create a replicated pool:

Syntax

-

```
ceph osd pool create POOL_NAME PG_NUM PGP_NUM [replicated] \
     [CRUSH_RULE_NAME] [EXPECTED_NUMBER_OBJECTS]
```

Create an erasure-coded pool:

Syntax

```
ceph osd pool create POOL_NAME PG_NUM PGP_NUM erasure \
     [ERASURE_CODE_PROFILE] [CRUSH_RULE_NAME] [EXPECTED_NUMBER_OBJECTS]
```

Create a bulk pool:

Syntax

```
ceph osd pool create POOL_NAME [--bulk]
```

Where:

POOL_NAME

Description

The name of the pool. It must be unique.

Type

String

Required

Yes. If not specified, it is set to the default value.

Default

ceph

PG_NUM

Description

The total number of placement groups for the pool. See the *Placement Groups* section and the *Ceph Placement Groups (PGs) per Pool Calculator* for details on calculating a suitable number. The default value **8** is not suitable for most systems.

Type

Integer

Required

Yes

Default

8

PGP_NUM

Description

The total number of placement groups for placement purposes. This value must be equal to the total number of placement groups, except for placement group splitting scenarios.

Type

Integer

Required

Yes. If not specified it is set to the default value.

Default

8

replicated or **erasure**

Description

The pool type can be either **replicated** to recover from lost OSDs by keeping multiple copies of the objects or **erasure** to get a kind of generalized RAID5 capability. The replicated pools require more raw storage but implement all Ceph operations. The erasure-coded pools require less raw storage but only implement a subset of the available operations.

Type

String

Required

No

Default

replicated

CRUSH_RULE_NAME

Description

The name of the crush rule for the pool. The rule **MUST** exist. For replicated pools, the name is the rule specified by the **osd_pool_default_crush_rule** configuration setting. For erasure-coded pools the name is **erasure-code** if you specify the default erasure code profile or ***POOL_NAME*** otherwise. Ceph creates this rule with the specified name implicitly if the rule does not already exist.

Type

String

Required

No

Default

Uses **erasure-code** for an erasure-coded pool. For replicated pools, it uses the value of the **osd_pool_default_crush_rule** variable from the Ceph configuration.

EXPECTED_NUMBER_OBJECTS

Description

The expected number of objects for the pool. By setting this value together with a negative **filestore_merge_threshold** variable, Ceph splits the placement groups at pool creation time to avoid the latency impact to perform runtime directory splitting.

Type

Integer

Required

No

Default

0, no splitting at the pool creation time.

ERASURE_CODE_PROFILE

Description

For erasure-coded pools only. Use the erasure code profile. It must be an existing profile as defined by the **osd erasure-code-profile set** variable in the Ceph configuration file. For further information, see the Erasure Code Profiles section.

Type

String

Required

No

When you create a pool, set the number of placement groups to a reasonable value, for example to **100**. Consider the total number of placement groups per OSD. Placement groups are computationally expensive, so performance degrades when you have many pools with many placement groups, for example, 50 pools with 100 placement groups each. The point of diminishing returns depends upon the power of the OSD host.

Additional Resources

See the Placement Groups section and Ceph Placement Groups (PGs) per Pool Calculator for details on calculating an appropriate number of placement groups for your pool.

4.4. SETTING POOL QUOTA

You can set pool quotas for the maximum number of bytes and the maximum number of objects per pool.

Syntax

```
ceph osd pool set-quota POOL_NAME [max_objects OBJECT_COUNT] [max_bytes BYTES]
```

Example

```
[ceph: root@host01 /]# ceph osd pool set-quota data max_objects 10000
```

To remove a quota, set its value to **0**.

NOTE

In-flight write operations might overrun pool quotas for a short time until Ceph propagates the pool usage across the cluster. This is normal behavior. Enforcing pool quotas on in-flight write operations would impose significant performance penalties.

4.5. DELETING A POOL

Delete a pool:

Syntax

```
ceph osd pool delete POOL_NAME [POOL_NAME --yes-i-really-really-mean-it]
```

IMPORTANT

To protect data, storage administrators cannot delete pools by default. Set the **mon_allow_pool_delete** configuration option before deleting pools.

If a pool has its own rule, consider removing it after deleting the pool. If a pool has users strictly for its own use, consider deleting those users after deleting the pool.

4.6. RENAMING A POOL

Rename a pool:

Syntax

```
ceph osd pool rename CURRENT_POOL_NAME NEW_POOL_NAME
```

If you rename a pool and you have per-pool capabilities for an authenticated user, you must update the user's capabilities, that is caps, with the new pool name.

4.7. MIGRATING A POOL

Sometimes it is necessary to migrate all objects from one pool to another. This is done in cases such as needing to change parameters that cannot be modified on a specific pool. For example, needing to reduce the number of placement groups of a pool.

IMPORTANT

When a workload is using **only** Ceph Block Device images, follow the procedures documented for moving and migrating a pool within the *Red Hat Ceph Storage Block Device Guide*:

- Moving images between pools
- Migrating pools

The migration methods described for Ceph Block Device are more recommended than those documented here. using the cppool does not preserve all snapshots and snapshot related metadata, resulting in an unfaithful copy of the data. For example, copying an RBD pool does not completely copy the image. In this case, snaps are not present and will not work properly. The cppool also does not preserve the **user_version** field that some librados users may rely on.

If migrating a pool is necessary and your user workloads contain images other than Ceph Block Devices, continue with one of the procedures documented here.

Prerequisites

- If using the **rados cppool** command:
  - Read-only access to the pool is required.
  - Only use this command if you do not have RBD images and its snaps and **user_version** consumed by librados.

- If using the local drive RADOS commands, verify that sufficient cluster space is available. Two, three, or more copies of data will be present as per pool replication factor.

Procedure

Method one - the recommended direct way

Copy all objects with the **rados cppool** command.

IMPORTANT

Read-only access to the pool is required during copy.

Syntax

```
ceph osd pool create NEW_POOL PG_NUM [ <other new pool parameters> ]
rados cppool SOURCE_POOL NEW_POOL
ceph osd pool rename SOURCE_POOL NEW_SOURCE_POOL_NAME
ceph osd pool rename NEW_POOL SOURCE_POOL
```

Example

```
[ceph: root@host01 /]# ceph osd pool create pool1 250
[ceph: root@host01 /]# rados cppool pool2 pool1
[ceph: root@host01 /]# ceph osd pool rename pool2 pool3
[ceph: root@host01 /]# ceph osd pool rename pool1 pool2
```

Method two - using a local drive

1. Use the **rados export** and **rados import** commands and a temporary local directory to save all exported data.

   ### Syntax

   ```
   ceph osd pool create NEW_POOL PG_NUM [ <other new pool parameters> ]
   rados export --create SOURCE_POOL FILE_PATH
   rados import FILE_PATH NEW_POOL
   ```

   ### Example

   ```
   [ceph: root@host01 /]# ceph osd pool create pool1 250
   [ceph: root@host01 /]# rados export --create pool2 <path of export file>
   [ceph: root@host01 /]# rados import <path of export file> pool1
   ```

2. Required. Stop all I/O to the source pool.

3. Required. Resynchronize all modified objects.

   ### Syntax

   ```
   rados export --workers 5 SOURCE_POOL FILE_PATH
   rados import --workers 5 FILE_PATH NEW_POOL
   ```

Example

```
[ceph: root@host01 /]# rados export --workers 5 pool2 <path of export file>
[ceph: root@host01 /]# rados import --workers 5 <path of export file> pool1
```

4.8. VIEWING POOL STATISTICS

Show a pool's utilization statistics:

Example

```
[ceph: root@host01 /] rados df
```

4.9. SETTING POOL VALUES

Set a value to a pool:

Syntax

```
ceph osd pool set POOL_NAME KEY VALUE
```

The Pool Values section lists all key-values pairs that you can set.

4.10. GETTING POOL VALUES

Get a value from a pool:

Syntax

```
ceph osd pool get POOL_NAME KEY
```

You can view the list of all key-values pairs that you might get in the Pool Values section.

4.11. ENABLING A CLIENT APPLICATION

Red Hat Ceph Storage provides additional protection for pools to prevent unauthorized types of clients from writing data to the pool. This means that system administrators must expressly enable pools to receive I/O operations from Ceph Block Device, Ceph Object Gateway, Ceph Filesystem or for a custom application.

Enable a client application to conduct I/O operations on a pool:

Syntax

```
ceph osd pool application enable POOL_NAME APP {--yes-i-really-mean-it}
```

Where ***APP*** is:

- **cephfs** for the Ceph Filesystem.
- **rbd** for the Ceph Block Device.

- **rgw** for the Ceph Object Gateway.

NOTE

Specify a different ***APP*** value for a custom application.

IMPORTANT

A pool that is not enabled will generate a **HEALTH_WARN** status. In that scenario, the output for **ceph health detail -f json-pretty** gives the following output:

```
{
    "checks": {
        "POOL_APP_NOT_ENABLED": {
            "severity": "HEALTH_WARN",
            "summary": {
                "message": "application not enabled on 1 pool(s)"
            },
            "detail": [
                {
                    "message": "application not enabled on pool '_POOL_NAME_'"
                },
                {
                    "message": "use 'ceph osd pool application enable _POOL_NAME_
_APP_', where _APP_ is 'cephfs', 'rbd', 'rgw', or freeform for custom applications."
                }
            ]
        }
    },
    "status": "HEALTH_WARN",
    "overall_status": "HEALTH_WARN",
    "detail": [
        "'ceph health' JSON format has changed in luminous. If you see this your
monitoring system is scraping the wrong fields. Disable this with 'mon health
preluminous compat warning = false'"
    ]
}
```

NOTE

Initialize pools for the Ceph Block Device with **rbd pool init *POOL_NAME***.

4.12. DISABLING A CLIENT APPLICATION

Disable a client application from conducting I/O operations on a pool:

Syntax

```
ceph osd pool application disable POOL_NAME APP {--yes-i-really-mean-it}
```

Where ***APP*** is:

- **cephfs** for the Ceph Filesystem.

- **rbd** for the Ceph Block Device.
- **rgw** for the Ceph Object Gateway.

NOTE

Specify a different ***APP*** value for a custom application.

4.13. SETTING APPLICATION METADATA

Provides the functionality to set key-value pairs describing attributes of the client application.

Set client application metadata on a pool:

Syntax

```
ceph osd pool application set POOL_NAME APP KEY
```

Where ***APP*** is:

- **cephfs** for the Ceph Filesystem.
- **rbd** for the Ceph Block Device
- **rgw** for the Ceph Object Gateway

NOTE

Specify a different ***APP*** value for a custom application.

4.14. REMOVING APPLICATION METADATA

Remove client application metadata on a pool:

Syntax

```
ceph osd pool application rm POOL_NAME APP KEY
```

Where ***APP*** is:

- **cephfs** for the Ceph Filesystem.
- **rbd** for the Ceph Block Device
- **rgw** for the Ceph Object Gateway

NOTE

Specify a different ***APP*** value for a custom application.

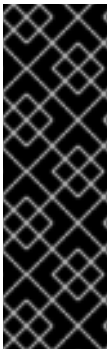
4.15. SETTING THE NUMBER OF OBJECT REPLICAS

Set the number of object replicas on a replicated pool:

Syntax

```
ceph osd pool set POOL_NAME size NUMBER_OF_REPLICAS
```

You can run this command for each pool.

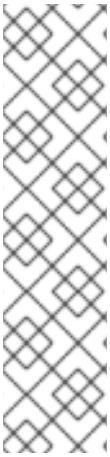


IMPORTANT

The ***NUMBER_OF_REPLICAS*** parameter includes the object itself. If you want to include the object and two copies of the object for a total of three instances of the object, specify **3**.

Example

```
[ceph: root@host01 /]# ceph osd pool set data size 3
```



NOTE

An object might accept I/O operations in degraded mode with fewer replicas than specified by the **pool size** setting. To set a minimum number of required replicas for I/O, use the **min_size** setting.

Example

```
[ceph: root@host01 /]# ceph osd pool set data min_size 2
```

This ensures that no object in the data pool receives an I/O with fewer replicas than specified by the **min_size** setting.

4.16. GETTING THE NUMBER OF OBJECT REPLICAS

Get the number of object replicas:

Example

```
[ceph: root@host01 /]# ceph osd dump | grep 'replicated size'
```

Ceph lists the pools, with the **replicated size** attribute highlighted. By default, Ceph creates two replicas of an object, that is a total of three copies, or a size of **3**.

4.17. POOL VALUES

The following list contains key-values pairs that you can set or get. For further information, see the Set Pool Values and Getting Pool Values sections.

size

Description

Specifies the number of replicas for objects in the pool. See the Setting the Number of Object Replicas section for further details. Applicable for the replicated pools only.

Type

Integer

min_size

Description

Sets the minimum number of replicas required for I/O. See the Setting the Number of Object Replicas section for further details. For erasure-coded pools, this should be set to a value greater than **k**. If I/O is allowed at the value **k**, then there is no redundancy and data is lost in the event of a permanent OSD failure. For more information, see Erasure code pools overview .

Type

Integer

crash_replay_interval

Description

Specifies the number of seconds to allow clients to replay acknowledged, but uncommitted requests.

Type

Integer

pg-num

Description

The total number of placement groups for the pool. See the *Pool, placement groups, and CRUSH Configuration Reference* section in the Red Hat Ceph Storage *Configuration Guide* for details on calculating a suitable number. The default value **8** is not suitable for most systems.

Type

Integer

Required

Yes.

Default

8

pgp-num

Description

The total number of placement groups for placement purposes. This **should be equal to the total number of placement groups**, except for placement group splitting scenarios.

Type

Integer

Required

Yes. Picks up default or Ceph configuration value if not specified.

Default

8

Valid Range

Equal to or less than what specified by the **pg_num** variable.

crush_rule

Description

The rule to use for mapping object placement in the cluster.

Type

String

hashpspool

Description

Enable or disable the **HASHPSPOOL** flag on a given pool. With this option enabled, pool hashing and placement group mapping are changed to improve the way pools and placement groups overlap.

Type

Integer

Valid Range

1 enables the flag, **0** disables the flag.

IMPORTANT

Do not enable this option on production pools of a cluster with a large amount of OSDs and data. All placement groups in the pool would have to be remapped causing too much data movement.

fast_read

Description

On a pool that uses erasure coding, if this flag is enabled, the read request issues subsequent reads to all shards, and waits until it receives enough shards to decode to serve the client. In the case of the **jerasure** and **isa erasure** plug-ins, once the first K replies return, the client's request is served immediately using the data decoded from these replies. This helps to allocate some resources for better performance. Currently this flag is only supported for erasure coding pools.

Type

Boolean

Defaults

0

allow_ec_overwrites

Description

Whether writes to an erasure coded pool can update part of an object, so the Ceph Filesystem and Ceph Block Device can use it.

Type

Boolean

compression_algorithm

Description

Sets inline compression algorithm to use with the BlueStore storage backend. This setting overrides the **bluestore_compression_algorithm** configuration setting.

Type

String

Valid Settings

lz4, **snappy**, **zlib**, **zstd**

compression_mode

Description

Sets the policy for the inline compression algorithm for the BlueStore storage backend. This setting overrides the **bluestore_compression_mode** configuration setting.

Type

String

Valid Settings

none, **passive**, **aggressive**, **force**

compression_min_blob_size

Description

BlueStore does not compress chunks smaller than this size. This setting overrides the **bluestore_compression_min_blob_size** configuration setting.

Type

Unsigned Integer

compression_max_blob_size

Description

BlueStore breaks chunks larger than this size into smaller blobs of **compression_max_blob_size** before compressing the data.

Type

Unsigned Integer

nodelete

Description

Set or unset the **NODELETE** flag on a given pool.

Type

Integer

Valid Range

1 sets flag. **0** unsets flag.

nopgchange

Description

Set or unset the **NOPGCHANGE** flag on a given pool.

Type

Integer

Valid Range

1 sets the flag. **0** unsets the flag.

nosizechange

Description

Set or unset the **NOSIZECHANGE** flag on a given pool.

Type

Integer

Valid Range

1 sets the flag. **0** unsets the flag.

write_fadvise_dontneed

Description

Set or unset the **WRITE_FADVISE_DONTNEED** flag on a given pool.

Type

Integer

Valid Range

1 sets the flag. **0** unsets the flag.

noscrub

Description

Set or unset the **NOSCRUB** flag on a given pool.

Type

Integer

Valid Range

1 sets the flag. **0** unsets the flag.

nodeep-scrub

Description

Set or unset the **NODEEP_SCRUB** flag on a given pool.

Type

Integer

Valid Range

1 sets the flag. **0** unsets the flag.

scrub_min_interval

Description

The minimum interval in seconds for pool scrubbing when load is low. If it is **0**, Ceph uses the **osd_scrub_min_interval** configuration setting.

Type

Double

Default

0

scrub_max_interval

Description

The maximum interval in seconds for pool scrubbing irrespective of cluster load. If it is **0**, Ceph uses the **osd_scrub_max_interval** configuration setting.

Type

Double

Default

0

deep_scrub_interval

Description

The interval in seconds for pool 'deep' scrubbing. If it is **0**, Ceph uses the **osd_deep_scrub_interval** configuration setting.

Type

Double

Default

0

CHAPTER 5. ERASURE CODE POOLS OVERVIEW

Ceph storage strategies involve defining data durability requirements. Data durability means the ability to sustain the loss of one or more OSDs without losing data.

Ceph stores data in pools and there are two types of the pools:

- *replicated*
- *erasure-coded*

Ceph uses the replicated pools by default, meaning the Ceph copies every object from a primary OSD node to one or more secondary OSDs.

The erasure-coded pools reduce the amount of disk space required to ensure data durability but it is computationally a bit more expensive than replication.

Erasure coding is a method of storing an object in the Ceph storage cluster durably where the erasure code algorithm breaks the object into data chunks (**k**) and coding chunks (**m**), and stores those chunks in different OSDs.

In the event of the failure of an OSD, Ceph retrieves the remaining data (**k**) and coding (**m**) chunks from the other OSDs and the erasure code algorithm restores the object from those chunks.

NOTE

Red Hat recommends **min_size** for erasure-coded pools to be **K+1** or more to prevent loss of writes and data.

Erasure coding uses storage capacity more efficiently than replication. The n-replication approach maintains **n** copies of an object (3x by default in Ceph), whereas erasure coding maintains only **k** + **m** chunks. For example, 3 data and 2 coding chunks use 1.5x the storage space of the original object.

While erasure coding uses less storage overhead than replication, the erasure code algorithm uses more RAM and CPU than replication when it accesses or recovers objects. Erasure coding is advantageous when data storage must be durable and fault tolerant, but do not require fast read performance (for example, cold storage, historical records, and so on).

For the mathematical and detailed explanation on how erasure code works in Ceph, see the *Ceph Erasure Coding* section in the *Architecture Guide* for Red Hat Ceph Storage 6.

Ceph creates a **default** erasure code profile when initializing a cluster with **k=2** and **m=2**, This mean that Ceph will spread the object data over four OSDs (**k+m == 4**) and Ceph can lose one of those OSDs without losing data. To know more about erasure code profiling see the Erasure Code Profiles section.

IMPORTANT

Configure only the **.rgw.buckets** pool as erasure-coded and all other Ceph Object Gateway pools as replicated, otherwise an attempt to create a new bucket fails with the following error:

```
set_req_state_err err_no=95 resorting to 500
```

The reason for this is that erasure-coded pools do not support the **omap** operations and certain Ceph Object Gateway metadata pools require the **omap** support.

5.1. CREATING A SAMPLE ERASURE-CODED POOL

The simplest erasure coded pool is equivalent to RAID5 and requires at least three hosts:

Example

```
$ ceph osd pool create ecpool 32 32 erasure
pool 'ecpool' created
$ echo ABCDEFGHI | rados --pool ecpool put NYAN -
$ rados --pool ecpool get NYAN -
ABCDEFGHI
```

NOTE

The 32 in **pool create** stands for the number of placement groups.

5.2. ERASURE CODE PROFILES

Ceph defines an erasure-coded pool with a **profile**. Ceph uses a profile when creating an erasure-coded pool and the associated CRUSH rule.

Ceph creates a default erasure code profile when initializing a cluster and it provides the same level of redundancy as two copies in a replicated pool. However, it uses 25% less storage capacity. The default profiles define **k=2** and **m=2**, meaning Ceph will spread the object data over four OSDs (**k+m=4**) and Ceph can lose one of those OSDs without losing data.

The default erasure code profile can sustain the loss of a single OSD. It is equivalent to a replicated pool with a size two, but requires 1.5 TB instead of 2 TB to store 1 TB of data. To display the default profile use the following command:

```
$ ceph osd erasure-code-profile get default
k=2
m=2
plugin=jerasure
technique=reed_sol_van
```

You can create a new profile to improve redundancy without increasing raw storage requirements. For instance, a profile with **k=8** and **m=4** can sustain the loss of four (**m=4**) OSDs by distributing an object on 12 (**k+m=12**) OSDs. Ceph divides the object into **8** chunks and computes **4** coding chunks for recovery. For example, if the object size is 8 MB, each data chunk is 1 MB and each coding chunk has the same size as the data chunk, that is also 1 MB. The object will not be lost even if four OSDs fail simultaneously.

The most important parameters of the profile are *k*, *m* and *crush-failure-domain*, because they define the storage overhead and the data durability.

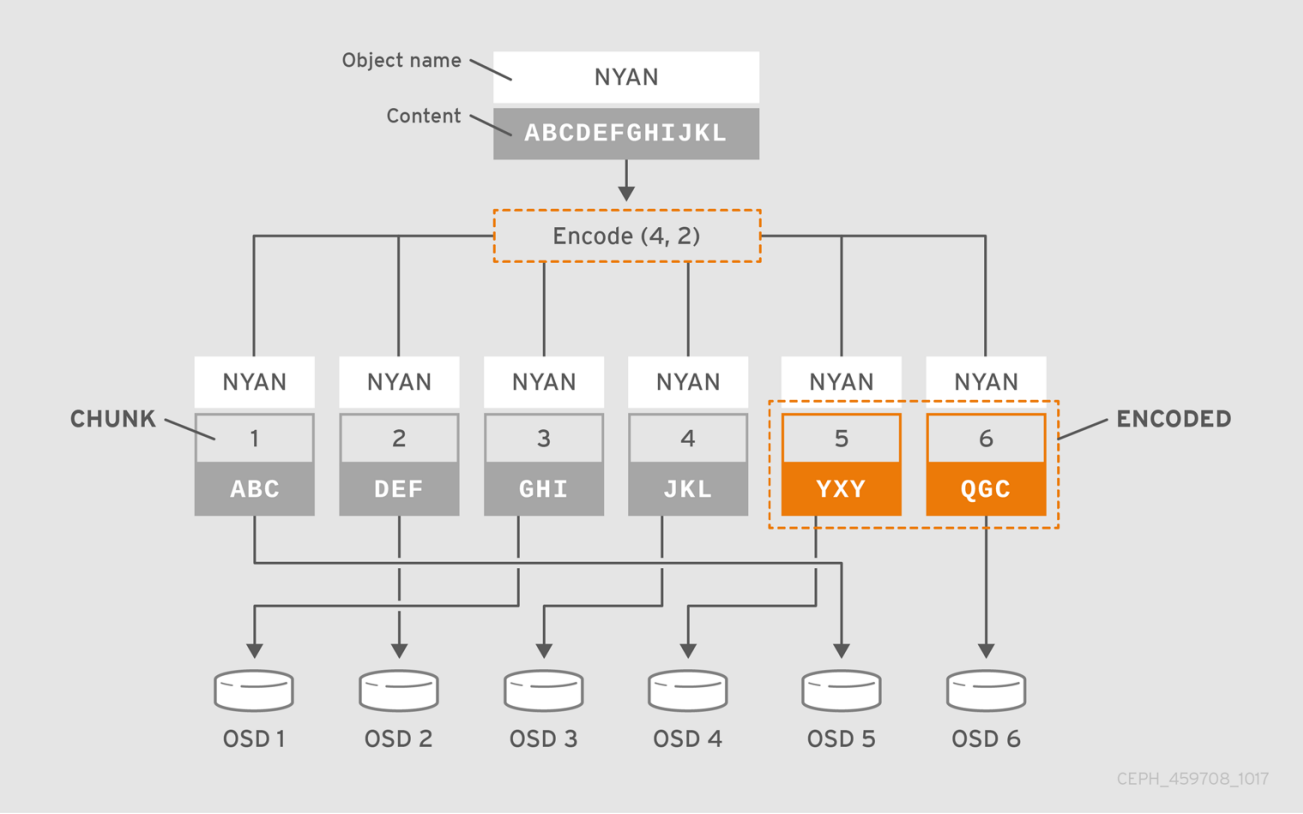
IMPORTANT

Choosing the correct profile is important because you cannot change the profile after you create the pool. To modify a profile, you must create a new pool with a different profile and migrate the objects from the old pool to the new pool.

For instance, if the desired architecture must sustain the loss of two racks with a storage overhead of 40% overhead, the following profile can be defined:

```
$ ceph osd erasure-code-profile set myprofile \
   k=4 \
   m=2 \
   crush-failure-domain=rack
$ ceph osd pool create ecpool 12 12 erasure *myprofile*
$ echo ABCDEFGHIJKL | rados --pool ecpool put NYAN -
$ rados --pool ecpool get NYAN -
ABCDEFGHIJKL
```

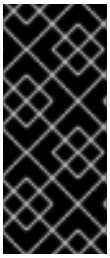
The primary OSD will divide the *NYAN* object into four (*k=4*) data chunks and create two additional chunks (*m=2*). The value of *m* defines how many OSDs can be lost simultaneously without losing any data. The *crush-failure-domain=rack* will create a CRUSH rule that ensures no two chunks are stored in the same rack.



IMPORTANT

Red Hat supports the following *jerasure* coding values for *k*, and *m*:

- k=8 m=3
- k=8 m=4
- k=4 m=2



IMPORTANT

If the number of OSDs lost equals the number of coding chunks (**m**), some placement groups in the erasure coding pool will go into incomplete state. If the number of OSDs lost is less than **m**, no placement groups will go into incomplete state. In either situation, no data loss will occur. If placement groups are in incomplete state, temporarily reducing **min_size** of an erasure coded pool will allow recovery.

5.2.1. Setting OSD erasure-code-profile

To create a new erasure code profile:

Syntax

```
ceph osd erasure-code-profile set NAME \
     [<directory=DIRECTORY>] \
     [<plugin=PLUGIN>] \
     [<stripe_unit=STRIPE_UNIT>] \
     [<_CRUSH_DEVICE_CLASS_>]\
     [<_CRUSH_FAILURE_DOMAIN_>]\
     [<key=value> ...] \
     [--force]
```

Where:

directory

Description

Set the **directory** name from which the erasure code plug-in is loaded.

Type

String

Required

No.

Default

/usr/lib/ceph/erasure-code

plugin

Description

Use the erasure code plug-in to compute coding chunks and recover missing chunks. See the Erasure Code Plug-ins section for details.

Type

String

Required

No.

Default

jerasure

stripe_unit

Description

The amount of data in a data chunk, per stripe. For example, a profile with 2 data chunks and **stripe_unit=4K** would put the range 0-4K in chunk 0, 4K-8K in chunk 1, then 8K-12K in chunk 0 again. This should be a multiple of 4K for best performance. The default value is taken from the monitor config option **osd_pool_erasure_code_stripe_unit** when a pool is created. The stripe_width of a pool using this profile will be the number of data chunks multiplied by this **stripe_unit**.

Type

String

Required

No.

Default

4K

crush-device-class

Description

The device class, such as **hdd** or **ssd**.

Type

String

Required

No

Default

none, meaning CRUSH uses all devices regardless of class.

crush-failure-domain

Description

The failure domain, such as **host** or **rack**.

Type

String

Required

No

Default

host

key

Description

The semantic of the remaining key-value pairs is defined by the erasure code plug-in.

Type

String

Required

No.

--force

Description

Override an existing profile by the same name.

Type

String

Required

No.

5.2.2. Removing OSD erasure-code-profile

To remove an erasure code profile:

Syntax

```
ceph osd erasure-code-profile rm NAME
```

If the profile is referenced by a pool, the deletion will fail.

5.2.3. Getting OSD erasure-code-profile

To display an erasure code profile:

Syntax

```
ceph osd erasure-code-profile get NAME
```

5.2.4. Listing OSD erasure-code-profile

To list the names of all erasure code profiles:

Syntax

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

5.3. ERASURE CODING WITH OVERWRITES

By default, erasure coded pools only work with the Ceph Object Gateway, which performs full object writes and appends.

Using erasure coded pools with overwrites allows Ceph Block Devices and CephFS store their data in an erasure coded pool:

Syntax

```
ceph osd pool set ERASURE_CODED_POOL_NAME allow_ec_overwrites true
```

Example

```
[ceph: root@host01 /]# ceph osd pool set ec_pool allow_ec_overwrites true
```

Enabling erasure coded pools with overwrites can only reside in a pool using BlueStore OSDs. Since BlueStore's checksumming is used to detect bit rot or other corruption during deep scrubs.

Erasure coded pools do not support omap. To use erasure coded pools with Ceph Block Devices and CephFS, store the data in an erasure coded pool, and the metadata in a replicated pool.

For Ceph Block Devices, use the **--data-pool** option during image creation:

Syntax

```
rbd create --size IMAGE_SIZE_M|G|T --data-pool _ERASURE_CODED_POOL_NAME
REPLICATED_POOL_NAME/IMAGE_NAME
```

Example

```
[ceph: root@host01 /]# rbd create --size 1G --data-pool ec_pool rep_pool/image01
```

If using erasure coded pools for CephFS, then setting the overwrites must be done in a file layout.

5.4. ERASURE CODE PLUGINS

Ceph supports erasure coding with a plug-in architecture, which means you can create erasure coded pools using different types of algorithms. Ceph supports Jerasure.

5.4.1. Creating a new erasure code profile using jerasure erasure code plugin

The *jerasure* plug-in is the most generic and flexible plug-in. It is also the default for Ceph erasure coded pools.

The *jerasure* plug-in encapsulates the JerasureH library. For detailed information about the parameters, see the *jerasure* documentation.

To create a new erasure code profile using the *jerasure* plug-in, run the following command:

Syntax

```
ceph osd erasure-code-profile set NAME \
    plugin=jerasure \
    k=DATA_CHUNKS \
    m=DATA_CHUNKS \
    technique=TECHNIQUE \
    [crush-root=ROOT] \
    [crush-failure-domain=BUCKET_TYPE] \
    [directory=DIRECTORY] \
    [--force]
```

Where:

k

Description

Each object is split in **data-chunks** parts, each stored on a different OSD.

Type

Integer

Required

Yes.

Example

4

m

Description

Compute **coding chunks** for each object and store them on different OSDs. The number of coding chunks is also the number of OSDs that can be down without losing data.

Type

Integer

Required

Yes.

Example

2

technique

Description

The more flexible technique is *reed_sol_van*; it is enough to set *k* and *m*. The *cauchy_good* technique can be faster but you need to choose the *packetsize* carefully. All of *reed_sol_r6_op*, *liberation*, *blaum_roth*, *liber8tion* are *RAID6* equivalents in the sense that they can only be configured with *m=2*.

Type

String

Required

No.

Valid Settings

reed_sol_vanreed_sol_r6_opcauchy_origcauchy_goodliberationblaum_rothliber8tion

Default

reed_sol_van

packetsize

Description

The encoding will be done on packets of *bytes* size at a time. Choosing the correct packet size is difficult. The *jerasure* documentation contains extensive information on this topic.

Type

Integer

Required

No.

Default

2048

crush-root

Description

The name of the CRUSH bucket used for the first step of the rule. For instance **step take default**.

Type

String

Required

No.

Default

default

crush-failure-domain

Description

Ensure that no two chunks are in a bucket with the same failure domain. For instance, if the failure domain is **host** no two chunks will be stored on the same host. It is used to create a rule step such as **step chooseleaf host**.

Type

String

Required

No.

Default

host

directory

Description

Set the **directory** name from which the erasure code plug-in is loaded.

Type

String

Required

No.

Default

/usr/lib/ceph/erasure-code

--force

Description

Override an existing profile by the same name.

Type

String

Required

No.

5.4.2. Controlling CRUSH Placement

The default CRUSH rule provides OSDs that are on different hosts. For instance:

```
chunk nr    01234567

step 1      _cDD_cDD
```

```
step 2      cDDD____
step 3      ____cDDD
```

needs exactly *8* OSDs, one for each chunk. If the hosts are in two adjacent racks, the first four chunks can be placed in the first rack and the last four in the second rack. Recovering from the loss of a single OSD does not require using bandwidth between the two racks.

For instance:

```
crush-steps='[ [ "choose", "rack", 2 ], [ "chooseleaf", "host", 4 ] ]'
```

creates a rule that selects two crush buckets of type *rack* and for each of them choose four OSDs, each of them located in a different bucket of type *host*.

The rule can also be created manually for finer control.