CRUSHTOOL – CRUSH MAP MANIPULATION TOOL

SYNOPSIS

crushtool (-d *map* | -c *map.txt* | –build –num_osds *numosds layer1 ...* | –test) [-o *outfile*]

DESCRIPTION

crushtool is a utility that lets you create, compile, decompile and test CRUSH map files.

CRUSH is a pseudo-random data distribution algorithm that efficiently maps input values (which, in the context of Ceph, correspond to Placement Groups) across a heterogeneous, hierarchically structured device map. The algorithm was originally described in detail in the following paper (although it has evolved some since then):

```
http://www.ssrc.ucsc.edu/Papers/weil-sc06.pdf
```

The tool has four modes of operation.

`--compile|-c` `map.txt`
: will compile a plaintext map.txt into a binary map file.

`--decompile|-d` `map`
: will take the compiled map and decompile it into a plaintext source file, suitable for editing.

`--build` `--num_osds {num-osds} layer1 ...`
: will create map with the given layer structure. See below for a detailed explanation.

`--test`
: will perform a dry run of a CRUSH mapping for a range of input values `[--min-x,--max-x]` (default `[0,1023]`) which can be thought of as simulated Placement Groups. See below for a more detailed explanation.

Unlike other Ceph tools, **crushtool** does not accept generic options such as **–debug-crush** from the command line. They can, however, be provided via the CEPH_ARGS environment variable. For instance, to silence all output from the CRUSH subsystem:

```
CEPH_ARGS="--debug-crush 0" crushtool ...
```

RUNNING TESTS WITH –TEST

The test mode will use the input crush map (as specified with **-i map**) and perform a dry run of CRUSH mapping or random placement (if **–simulate** is set). On completion, two kinds of reports can be created. 1) The **–show-...** option outputs human readable information on stderr. 2) The **–output-csv** option creates CSV files that are documented by the **–help-output** option.

Note: Each Placement Group (PG) has an integer ID which can be obtained from `ceph pg dump` (for example PG 2.2f means pool id 2, PG id 32). The pool and PG IDs are combined by a function to get a value which is given to CRUSH to map it to OSDs. crushtool does not know about PGs or pools; it only runs simulations by mapping values in the range `[--min-x,--max-x]`.

`--show-statistics`
: Displays a summary of the distribution. For instance:

```
rule 1 (metadata) num_rep 5 result size == 5:    1024/1024
```

shows that rule **1** which is named **metadata** successfully mapped **1024** values to **result size == 5** devices when trying to map them to **num_rep 5** replicas. When it fails to provide the required mapping, presumably because the number of **tries** must be increased, a breakdown of the failures is displayed. For instance:

```
rule 1 (metadata) num_rep 10 result size == 8:   4/1024
rule 1 (metadata) num_rep 10 result size == 9:   93/1024
rule 1 (metadata) num_rep 10 result size == 10:  927/1024
```

shows that although **num_rep 10** replicas were required, **4** out of **1024** values (**4/1024**) were mapped to **result size == 8** devices only.

`--show-mappings`

Displays the mapping of each value in the range `[--min-x,--max-x]`. For instance:

```
CRUSH rule 1 x 24 [11,6]
```

shows that value **24** is mapped to devices **[11,6]** by rule **1**.

`--show-bad-mappings`

Displays which value failed to be mapped to the required number of devices. For instance:

```
bad mapping rule 1 x 781 num_rep 7 result [8,10,2,11,6,9]
```

shows that when rule **1** was required to map **7** devices, it could map only six : **[8,10,2,11,6,9]**.

`--show-utilization`

Displays the expected and actual utilisation for each device, for each number of replicas. For instance:

```
device 0: stored : 951      expected : 853.333
device 1: stored : 963      expected : 853.333
...
```

shows that device **0** stored **951** values and was expected to store **853**. Implies **–show-statistics**.

`--show-utilization-all`

Displays the same as **–show-utilization** but does not suppress output when the weight of a device is zero. Implies **–show-statistics**.

`--show-choose-tries`

Displays how many attempts were needed to find a device mapping. For instance:

```
0:     95224
1:      3745
2:      2225
..
```

shows that **95224** mappings succeeded without retries, **3745** mappings succeeded with one attempts, etc. There are as many rows as the value of the **–set-choose-total-tries** option.

`--output-csv`

Creates CSV files (in the current directory) containing information documented by **–help-output**. The files are named after the rule used when collecting the statistics. For instance, if the rule : ‘metadata’ is used, the CSV files will be:

```
metadata-absolute_weights.csv
metadata-device_utilization.csv
...
```

The first line of the file shortly explains the column layout. For instance:

```
metadata-absolute_weights.csv
Device ID, Absolute Weight
0,1
...
```

`--output-name` NAME

Prepend **NAME** to the file names generated when **–output-csv** is specified. For instance **–output-name FOO** will create files:

```
FOO-metadata-absolute_weights.csv
FOO-metadata-device_utilization.csv
...
```

The **–set-…** options can be used to modify the tunables of the input crush map. The input crush map is modified in memory. For example:

```
$ crushtool -i mymap --test --show-bad-mappings
bad mapping rule 1 x 781 num_rep 7 result [8,10,2,11,6,9]
```

could be fixed by increasing the **choose-total-tries** as follows:

> $ crushtool -i mymap –test
>
> –show-bad-mappings –set-choose-total-tries 500

BUILDING A MAP WITH –BUILD

The build mode will generate hierarchical maps. The first argument specifies the number of devices (leaves) in the CRUSH hierarchy. Each layer describes how the layer (or devices) preceding it should be grouped.

Each layer consists of:

```
bucket ( uniform | list | tree | straw ) size
```

The **bucket** is the type of the buckets in the layer (e.g. "rack"). Each bucket name will be built by appending a unique number to the **bucket** string (e.g. "rack0", "rack1"…).

The second component is the type of bucket: **straw** should be used most of the time.

The third component is the maximum size of the bucket. A size of zero means a bucket of infinite capacity.

EXAMPLE

Suppose we have two rows with two racks each and 20 nodes per rack. Suppose each node contains 4 storage devices for Ceph OSD Daemons. This configuration allows us to deploy 320 Ceph OSD Daemons. Lets assume a 42U rack with 2U nodes, leaving an extra 2U for a rack switch.

To reflect our hierarchy of devices, nodes, racks and rows, we would execute the following:

```
$ crushtool -o crushmap --build --num_osds 320 \
       node straw 4 \
       rack straw 20 \
       row straw 2 \
       root straw 0
# id        weight  type name       reweight
-87 320     root root
-85 160             row row0
-81 80                      rack rack0
-1  4                               node node0
0   1                                       osd.0   1
1   1                                       osd.1   1
2   1                                       osd.2   1
3   1                                       osd.3   1
-2  4                               node node1
4   1                                       osd.4   1
5   1                                       osd.5   1
...
```

CRUSH rules are created so the generated crushmap can be tested. They are the same rules as the ones created by default when creating a new Ceph cluster. They can be further edited with:

```
# decompile
crushtool -d crushmap -o map.txt

# edit
emacs map.txt

# recompile
crushtool -c map.txt -o crushmap
```

EXAMPLE OUTPUT FROM –TEST

See https://github.com/ceph/ceph/blob/master/src/test/cli/crushtool/set-choose.t for sample `crushtool --test` commands and output produced thereby.

AVAILABILITY

crushtool is part of Ceph, a massively scalable, open-source, distributed storage system. Please refer to the Ceph documentation at http://ceph.com/docs for more information.

SEE ALSO

ceph(8), osdmaptool(8),

AUTHORS

John Wilkins, Sage Weil, Loic Dachary