DigitalOcean

pgremapper: CRUSHing cluster operational complexity

Oct. 2023

Joshua Baergen, Fleet Storage

Contents

- Who Are We?
- An Augment Story
- The Problem
- A Solution!
- An Even Better Solution, in Theory

A brief history

What is **DigitalOcean**?

- Founded in 2012
- Core concept: Simplicity
- SSD backed VM was very attractive in 2012
- Four years later we introduce Volumes (block)
- Many more products since, including Spaces (object) and several SaaS offerings such as DBaaS, DOKS (k8s), App Platform, and Serverless Functions
- Datacenters in 9 regions, 14 total choices
- IPO in March 2021

2012 Droplet: Introduction of the SSD-backed VM

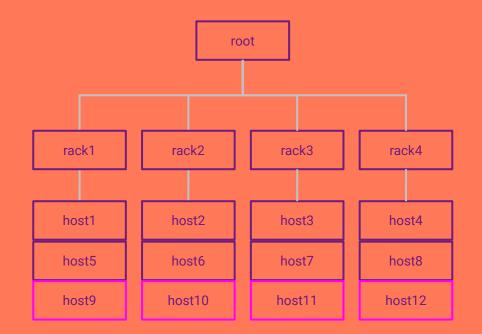
2016 Volumes: Ceph backed attachable Droplet storage

2017 Spaces: Ceph backed object storage

When something simple goes wrong.

Let's add four new hosts (24 disks each) to this system:

- Bootstrap the hosts
- ceph osd set nobackfill
- ceph osd crush move host9 rack=rack1, etc.
- ceph osd unset nobackfill



Hmm, we should be able to run at least 96 backfills in parallel (one per target disk), but only 25 are running.

 Let's increase backfill concurrency: ceph config set osd osd_max_backfills 3 \$ ceph -s ... 2730 active+clean 1341 active+remapped+backfill_wait 25 active+remapped+backfilling

110 backfilling - that's more like it!

It's a bit odd that 110 != 25*3, but the cluster seems fine so we leave for the day.

\$ ceph -s ... 2730 active+clean 1256 active+remapped+backfill_wait 110 active+remapped+backfilling

We wake up to this the next morning.

- ceph config set osd osd_max_backfills 1
- No sleep for the next week :(

\$ ceph -s ... 2840 active+clean

- 423 active+under...+degraded+remapped+backfill_wait
- 12 active+under...+degraded+remapped+backfilling
- 787 active+remapped+backfill_wait
- 34 active+remapped+backfilling

...

How does this happen?

ONE DOES NOT SIMPLY

CHANGE THE CRUSH MAP

Brief Note on Ceph Versions

To the best of our knowledge, the behaviour described in the following slides applies to all recent versions of Ceph.

The described behaviour was studied most in Luminous and Nautilus, and Pacific behaves similarly.

We have less experience with Quincy+ but we believe they behave similarly (though see next point).

The mClock scheduler (available Pacific, default in Quincy) may help a bit.

Lack of Backfill Concurrency

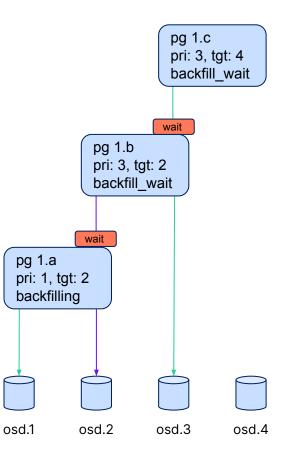
Why did we we have to increase osd_max_backfills to get the expected backfill concurrency?

In order to start a backfill, at least two reservations must be made - one on the primary OSD, then one on each backfill target OSD.



See the backfills on the right - A is backfilling, B has the primary reservation but is waiting for target, and C is stuck behind B.

Increasing osd_max_backfills increases the number of reservations (primary/target) available for each OSD, but more concurrency means more load.



Backfill Source Overload

Once we increased osd_max_backfills, why did some backfills go degraded?

As noted on the previous slide, there are at least two backfill reservations: one on the primary OSD, then one on each backfill target OSD.



There is no reservation made for the backfill source. For EC PGs, the backfill data source is likely not the primary OSD.

At any backfill concurrency, but especially at higher concurrencies, a single source OSD can get overloaded (leading to flaps and thus degradation, for example).

\$ ceph pg dump pgs_brief PG STATE	UP PRI	ACTING
<pre>"" 1.d active++backfilling 1.e active++backfilling 1.f active++backfilling 1.g active++backfilling ""</pre>	[6, <mark>2</mark> ,1,7] 6 [8,1, <mark>7</mark> ,9] 8	[1,2,3,4] [6,4,1,7] [8,1,4,9] [5,4,1,8]

Waiting Recovery

Wouldn't it be nice if the relatively quick recovery could complete ahead of backfill that blocks it?



Recovery uses the same reservation mechanism as backfill.



Thus, long-running backfills can hold PGs degraded for hours (or days!) when they would otherwise be able to recover in seconds to minutes.



In turn, these recoveries can hold partial reservations just like backfill, further reducing backfill recovery.

\$ ceph pg dump pgs_brief PG STATE	UP	PRI	ACTING
" 1.d active++backfilling 1.e+recovery_wait 1.f+recovery_wait 1.g+recovery_wait 	[1,2,3,5] [1,2,6,7] [1,8,7,9] [1,3,5,8]	1 1	[1,2,3,4] [1,2,6,7] [1,8,7,9] [1,3,5,8]

Degraded Backfill

Recovery for a PG can hide behind an ongoing backfill.



PG 1.d on the right is moving data from OSD 3 to 4.

\$ ceph pg dump pgs_brief
PG STATE UP PRI ACTING
...
1.d ...+degraded+backfilling [1,2,4] 1 [1,2,3]
...

OSD 2 has restarted and needs recovery, which would be much faster than a backfill.

However, its recovery won't happen until the backfill completes, resulting in an extended reduction in durability.

Summary

Backfill incurred due to a CRUSH change can put the system into an extended at-risk state.

Backfill concurrency is hard to control

The reservation system leads to less concurrency than desired, and the osd_max_backfills knob is too big of a hammer; it's possible to end up with too much concurrency for some OSDs.

Recovery can get held up by backfill

Because recovery uses the same reservation mechanism as backfill and because backfill for a given PG won't pause for any needed recovery on that PG, systems can be in a degraded state much longer than they need to be.

No way to stop once started

Once a CRUSH change occurs, there's no way to get the system back into an active+clean state without waiting for backfill to complete (either rolling forward or back).

pgremapper

Control backfill via the upmap exception table.



pgremapper

A CLI tool that manipulates the upmap exception table (Luminous+) in order to change what backfill is scheduled.

\$ pgremapper --help Use the upmap to manipulate PG mappings (and thus scheduled backfill)



Open source, written in Go: github.com/digitalocean/pgremapper



Inspired by CERN IT's presentation in Nov. 2019 on using the upmap exception table for backfill control.



We've extended CERN's ideas in order to support a broad range of usecases.

pgremapper Design - Mapping State

pgremapper has an internal model of the upmap exception table and an intuitive API for querying and manipulation of it.



Ceph's upmap exception table API is simple - set table entry to new value or remove table entry. However, this can be tedious to manipulate manually.



On startup, pgremapper builds an internal representation of this table and presents a simplified API (remap PG from OSD X to Y), computing the final changes to the exception table.



The current state of the table can be queried (find all PGs remapped to OSD Y), and changes can be displayed in a diff format for dry-run purposes. \$ ceph osd dump | grep upmap | head -n 2
pg_upmap_items 1.7 [74,148]
pg_upmap_items 1.d [62,164]

ceph osd pg-upmap-items <pgid> <osdname
(id|osd.id)> [<osdname (id|osd.id)>...]

ceph osd rm-pg-upmap-items <pgid>

func (m *mappingState) remap(pgid string, from,
to int) - calculates required changes to
exception table

pgremapper Design - Backfill State

pgremapper also has an internal model of the current backfill reservation state on a per-OSD basis.



Prior to Octopus, admin socket access is required to query reservation state. Thus, pgremapper infers reservation state from a PG dump.



As changes are made to the mapping state (previous slide), the reservation model is also updated.

Thus, one can ensure that backfill is not scheduled if it would exceed supplied limits on backfills per OSD - no more backfill_wait or overloaded source OSDs!

func (bs *backfillState) hasRoomForRemap(pgid
string, from, to int) bool - return whether the
proposed remapping will cause backfill that
exceeds supplied per-OSD limits

Putting It Together

The drain command uses the mapping and backfill state in order to safely move PGs off of the given source OSD.



Many of the command options take *osdspecs* - a mixture of OSDs IDs and CRUSH buckets.



In the example on the right, pgremapper will change the upmap exception table to move PGs from OSD 14 to any number of OSDs on host5.



No more than 2 backfill reservations will be consumed on primary/target OSDs (including any pre-existing backfill), but up to 5 backfills will be allowed on OSD 14 (as primary and/or source). \$ pgremapper drain <source OSD> --target-osds <osdspec>[,<osdspec>] [--allow-movement-across <bucket type>] [--max-backfill-reservations default_max[,osdspec:max]] [--max-source-backfills <n>]

\$ pgremapper drain 14 --target-osds bucket:host5 --max-backfill-reservations 2,14:5 --max-source-backfills 5

Let's Try That Augment Again!

We can use cancel-backfill and undo-upmaps to schedule augment backfill in a more orderly fashion.



cancel-backfill will create upmap exception table entries to eliminate backfill (i.e. maps data back to acting OSDs). It even works for some degraded PGs, changing them into recoveries!



undo-upmaps removes the exception table entries we created above, following the rules we give it for scheduling backfill.



Running undo-upmaps in a loop will maintain maximum backfill concurrency; killing this loop and running cancel-backfill returns the system to an active+clean state.

```
$ ceph osd set nobackfill
$ ceph osd crush move host9 rack=rack1, etc.
$ pgremapper cancel-backfill → everything
active+clean!
$ ceph osd unset nobackfill
$ pgremapper undo-upmaps bucket:host9
bucket:host10 bucket:host11 bucket:host12
--max-backfill-reservations 3
--max-source-backfills 3 --target
$ ceph -s
""
3808 active+clean
288 active+remapped+backfilling
```

Other Supported Operations

The augment pattern - nobackfill, change CRUSH map, cancel-backfill, unset nobackfill, then undo-upmaps loop - can be used for many other CRUSH changes: Weight changes, draining OSDs/hosts for removal, etc. Additional Commands - see GitHub README for usecase examples:

> balance-bucket: Rebalance PGs in a provided CRUSH bucket

import/export-mappings: Save/restore a copy of current upmap state in ison

remap: Convenience API to apply a single remapping (source→target) to a PG

Caveats

pgremapper is not able to handle all situations.

Limited by what the upmap exception table is capable of - not all desired changes are expressible as upmaps. *(see right)*



Minor issue - if the system is still reacting to an osdmap change (peering or about to peer), pgremapper will not make the right decisions.



It probably struggles with exotic CRUSH maps (e.g. linked OSDs), though we haven't tried.

\$ ceph pg dump pgs_brief PG STATE	UP	PRI	ACTING
… 1.d active++backfilling …	[1, <mark>2</mark> ,3, <mark>4</mark>] 1	[1, <mark>4</mark> ,3, <mark>2</mark>]
To cancel this backfill, o upmap entry, which is not \$ ceph osd pg-upmap-items	supported	by (

An Even Better Solution, In Theory

What would it take to eliminate the need for pgremapper?



Lack of Backfill Concurrency

Backfills/recoveries hold partial reservations, blocking unrelated work from making progress.

A common technique for acquiring multiple locks is to back off if one of the locks is already held.

By doing this, no backfill_wait PGs would hold reservations, allowing others to make progress.

Since pgremapper has a global view, it can still make better decisions, but the above solution might be Good Enough[™].

Backfill Source Overload

No reservation for EC backfill source means that some OSDs can become overloaded.

The straightforward solution would be to add reservations to backfill sources.

This would exacerbate the multiple reservation problem, and so a solution for the problem on the previous slide would become more necessary.

Or should this be a different reservation category?

Waiting Recovery

Recovery, which would be fast-running, gets stuck behind backfill.

Option 1: Allow backfill to be paused when recovery is waiting.

Option 2: Allow some reservation slots to be set aside for recovery.

The second option is simpler and probably Good Enough™.

Degraded Backfill

OSDs that need recovery in a PG cannot do so until that PG's backfill completes.

pgremapper is able to turn these backfills back into recoveries.

Thus, it seems possible to change the PG state machine to process the recovery ahead of the backfill.

Probably the most complicated of the problems to solve. Worth it?

Thank you

