Key Message:

We identified one performance degradation issue (up to 17%) for Ceph object Storage (10M objects read). Turned out the root cause is unbalanced pg distribution, and the current replica placement strategy fails to lay emphasis on it. Various tunes were tried to try to solve this issue but without luck. So we intend to do some tunings on CRUSH to solve this problem, by adding new hash algorithms, changing placement choosing logic of crush choose, or adding soft link for pgs on overloaded osds, etc. as mentioned in the **Optimization proposals** part. Hopefully we can add some POC to make more balanced pg distribution strategy.

Background:

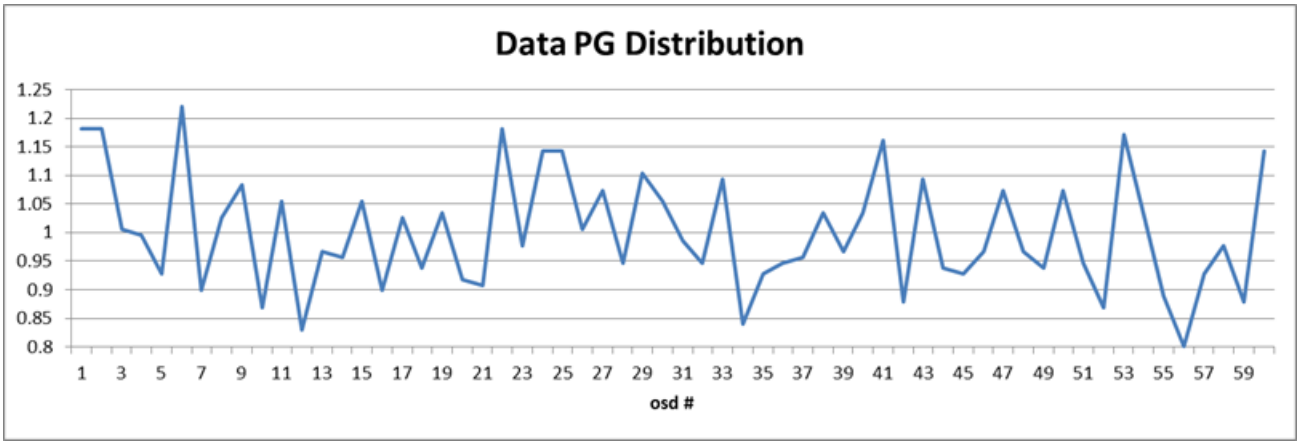
We have been doing object storage performance characterization on ceph, and found performance drop of large scaled 10M object reading tests compared to our baseline.

For base case, 100 containers x 100 10M objects were prepared for reading in our cluster, consisting of 5 hosts, each host with 12 disks.

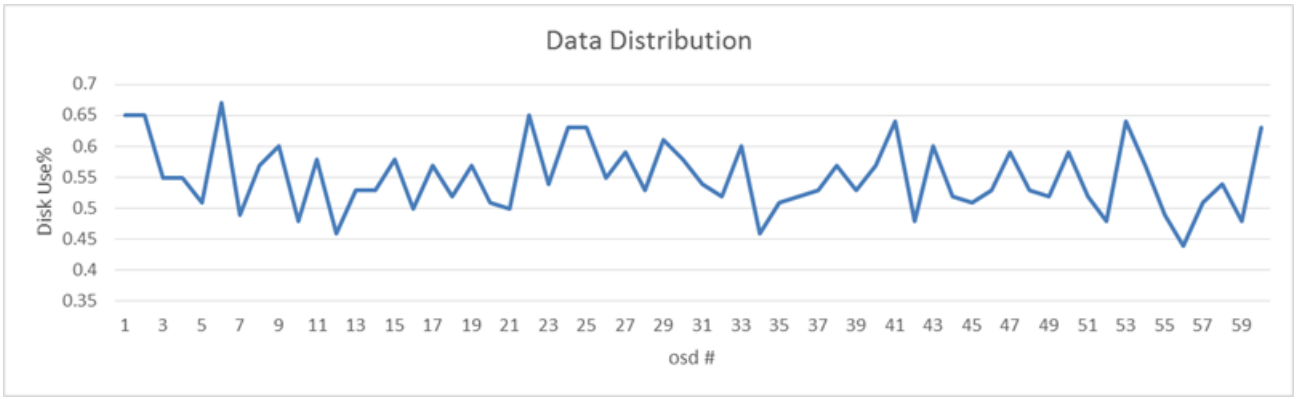
For advanced case, there were 10K containers x 100 10M objects. Table below shows the performance drop.

	data pg# (.rgw.buckets)	other pg#	data pg# per osd	pg dist	Disk-Use%	Throughput	Bandwidth	Success Ratio
	3 replica					op/s	MB/s	
10M base read	1280	128	-		-	117.2	1,117.72	100.00%
10M advance read	1280	128	64	47~87	35%~66%	96.7	922.18	100.00%
	2048	128	102	82~125	44%~65%	110.83	1,056.92	100.00%

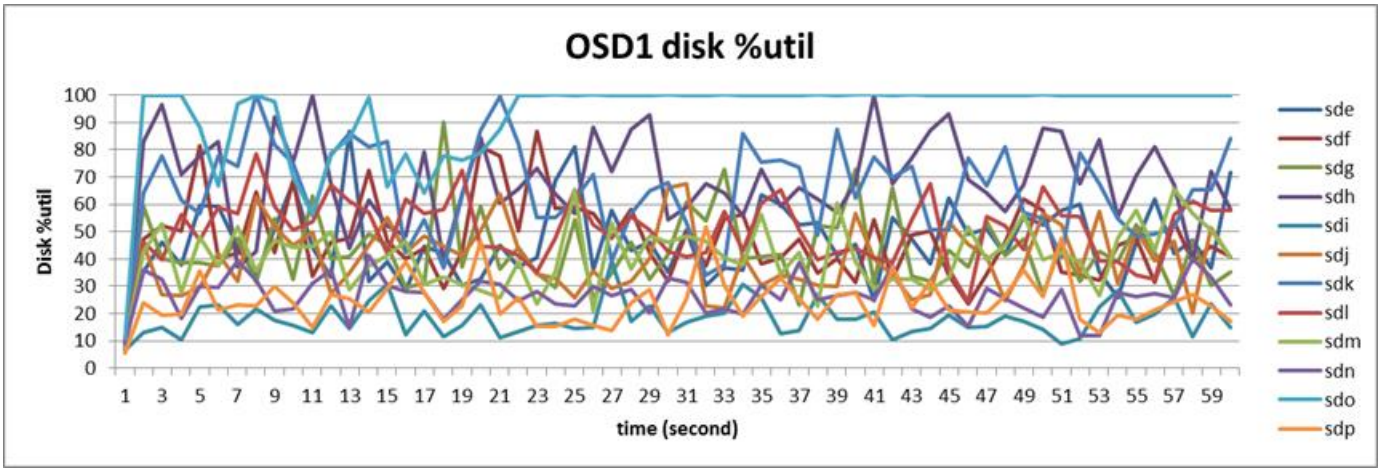
After investigating into this issue, we found unbalanced pg distribution among osds, resulting in up to 20% disk use% gap. Data PG distribution is normalized by average pg# of each osd. (data pg# = 2048x3 replica)



Data distribution stands for disk use% of each disk after we prepare 10Kx100 10M objects in cluster.



Disk use% gap leads to skewed disk %util in R/W workload, thus the overloaded disk became bottleneck and sacrificed overall performance. Figure below shows disk %util of ceph-osd1(host) in large scaled 10M reading workload. We can infer that disk sdo is always overloaded.



We found that unbalanced pg distribution is the root cause of performance drop. Various tunings were tried to solve this problem, including turning on hashpspool, using 'optimal' crush tunables, and changing bucket type, but without luck. We also increased pg_num of the pool, it seems that pg distribution is more balanced, however, it brought more memory consumption for each osd daemon at the same time.
For more detail data, please refer to the **Testing results** part.

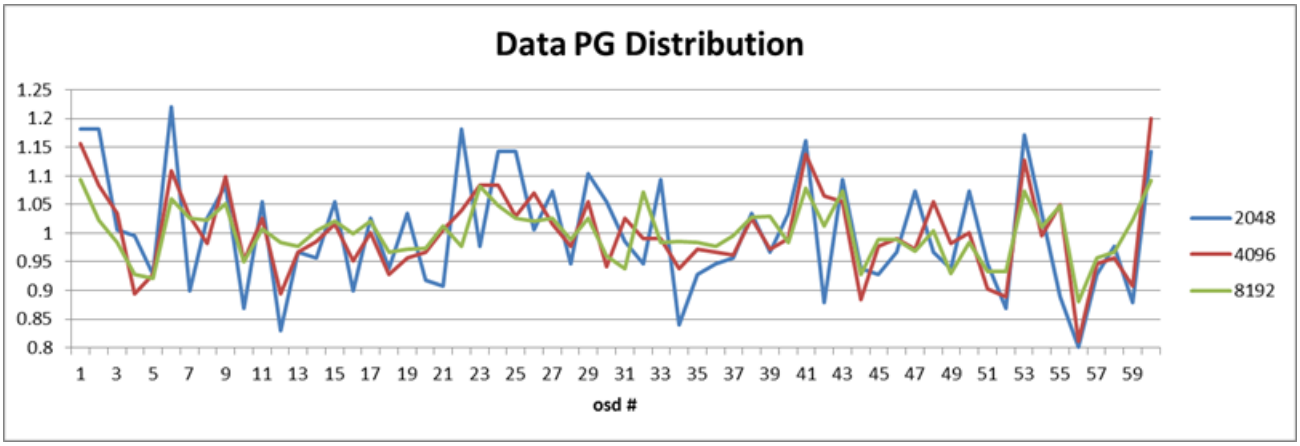
Optimization proposals:
After we dived into the source code of CRUSH and related papers, we proposed two possible optimizations:

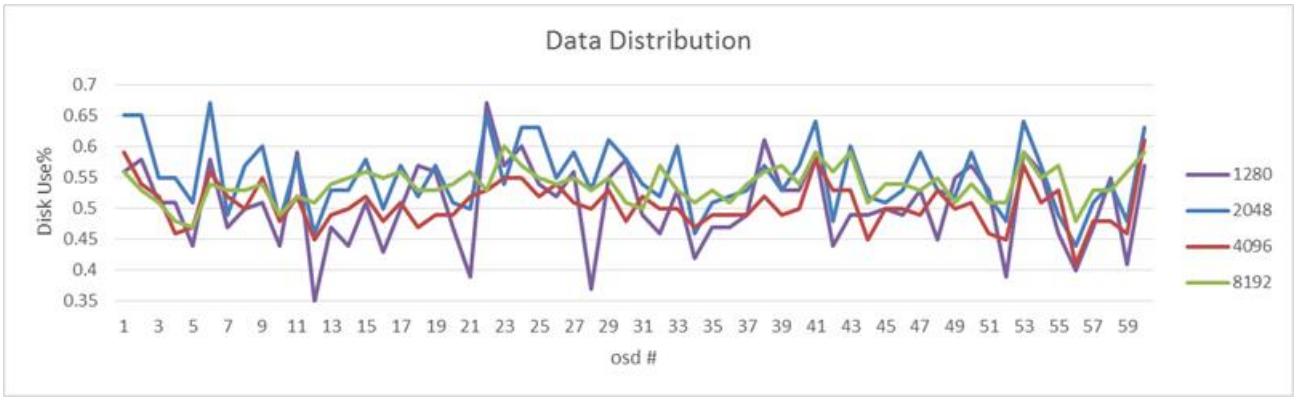
1. Add different hash algorithms, as an alternative for the Jenkin's hash, e.g. algorithm that will produce even values when range of input value (pg#) is relatively small. Or add new bucket type at the same time if necessary.
2. Find a better replica placement strategy instead of current retry logic of crush_choose_firstn, which may cause CRUSH to behave badly.
   We find there are several threshold of retry times by referring to code, choose_total_tries, choose_local_tries and choose_local_fallback_tries. They are used to decide whether to do a retry_bucket, retry_descent or use permutation to do an exhaustive bucket search. We are wondering if there is another retry strategy:
   a) Backtracking retry. Now the logic of crush_choose_firstn can only issue an retry either from the initial bucket(retry_descent) or from the current bucket (retry_bucket), how about retrying the intervening buckets?
   b) Adjust threshold of retry times by other values. We do noticed that the 'optimal' crush tunable could be used to make it, but we still encounter unbalanced [g distribution by using the optimal strategy. Please refer to 4 of the Testing results part.
   c) Add an mechanism that can adjust above mentioned thresholds adaptively. Maybe we can record the retry times of the previous call for CRUSH, and adjust retry thresholds automatically according to the record.
3. Add soft link for pg directories. During pg creation, we can create soft links for the pgs if pg# on the selected osd is more than some threshold, say 10% more than desired average number, to move objects that will be stored in this pg to another osd. Balanced disk utilization may be gained in this way.
4. Change placement strategy only for step of selecting devices from hosts. We found in our testing results that pg distribution was balanced among hosts, which is reasonable since pg# of each host is above 1K (according to the current BKM that pg# per osd should be about 100). So how about we apply CRUSH only on the interval buckets and find another simple but more balanced method to choose osd from host?

Testing results:
Here are the results of our previous testing on pg distribution.
We did tests on a cluster of 5 hosts, each host has 12 osds, all with the same weight, thus 60 osds in total. And all hosts in the same rack. The cluster map (crushmap) is shown in attached text file.
Results are show as following:

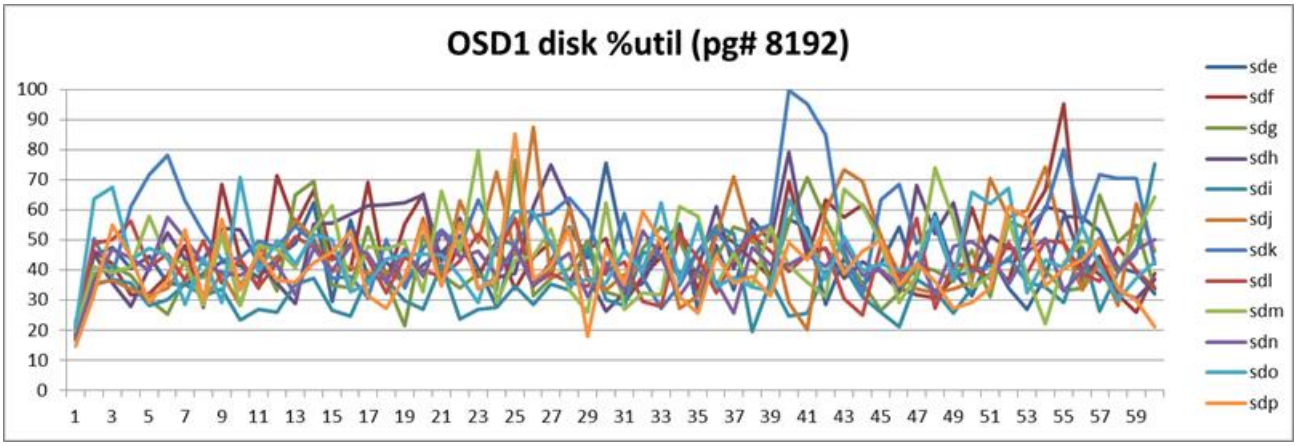
Increasing pg# actually brings balanced pg and data distribution(disk use%), resulting in better performance. But hosts may fail due to out of memory, since each osd needs more of that.

	data pg# (.rgw.buckets)	other pg#	data pg# per osd	pg dist	Disk-Use%	Throughput	Bandwidth	Success Ratio
						op/s	MB/s	
10M base read	1280	128	-		-	117.2	1,117.72	100.00%
10M advance read	1280	128	64	47~87	35%~66%	96.7	922.18	100.00%
	2048	128	102	82~125	44%~65%	110.83	1,056.92	100.00%
	4096	128	204	166~246	41%~61%	105.69	1,007.97	100.00%
	8192	128	408	361~448	47%~60%	116.83	1,114.18	99.60%





Compared with disk %util when pg# is 1280, disk is not bottleneck now.

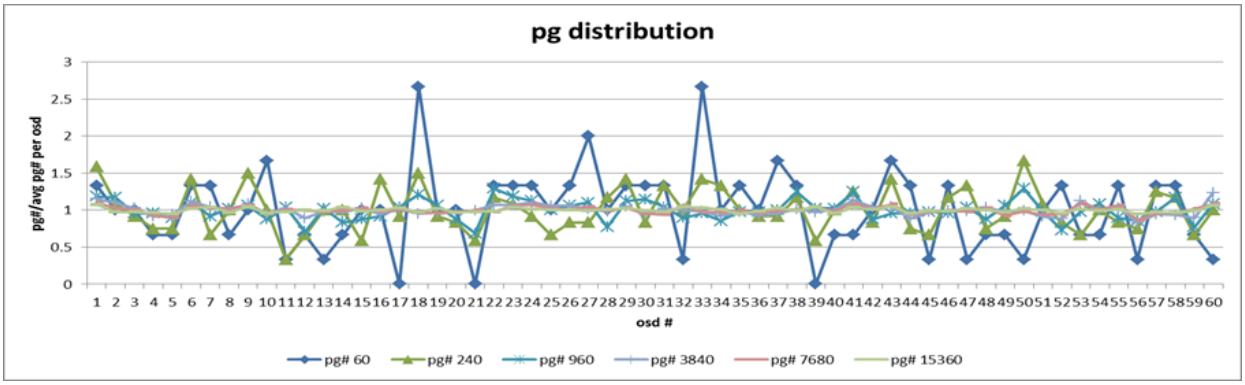


So we conclude that root reason of performance drop is unbalanced pg distribution, and then turn to try some tunings on it.

1. Default case. Turn off hashpspool, use default crush tunable, and define all buckets type as straw
   We found that pg distribution is more balanced as pg# increases. But there is still more than 30% gap even when the average pg# per osd reaches about 200 (192).

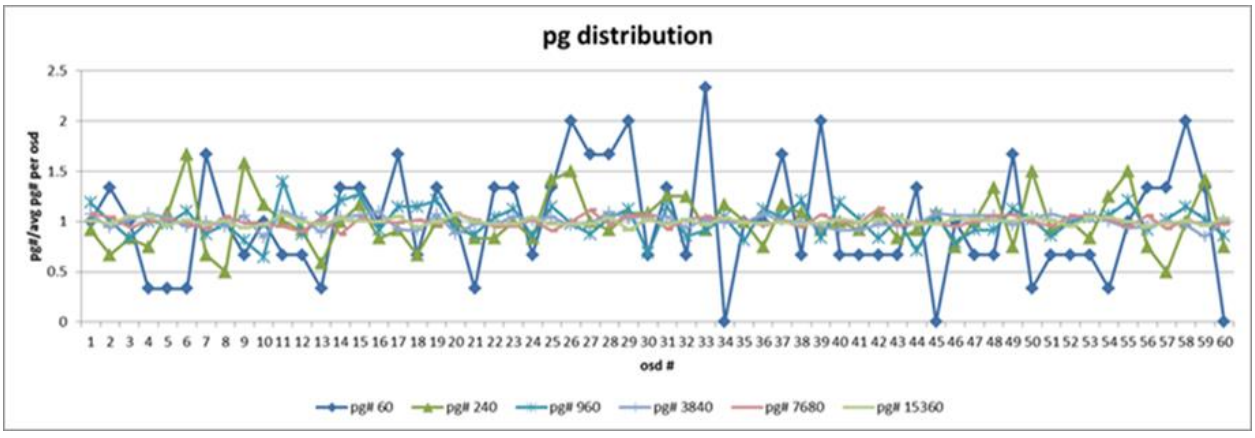
stdev in the last column stands for stdev of (pg# per osd / avg pg# per osd)

pg#	replica#	total pg#	avg pg# per osd	pg dist	pg# per osd/avg pg# per osd	stdev
60	3	180	3	0~8	0.00~2.67	0.544
240	3	720	12	4~20	0.33~1.67	0.287
960	3	2880	48	33~62	0.69~1.29	0.138
3840	3	11520	192	159~237	0.83~1.23	0.073
7680	3	23040	384	332~424	0.86~1.10	0.050
15360	3	46080	768	727~827	0.95~1.08	0.030



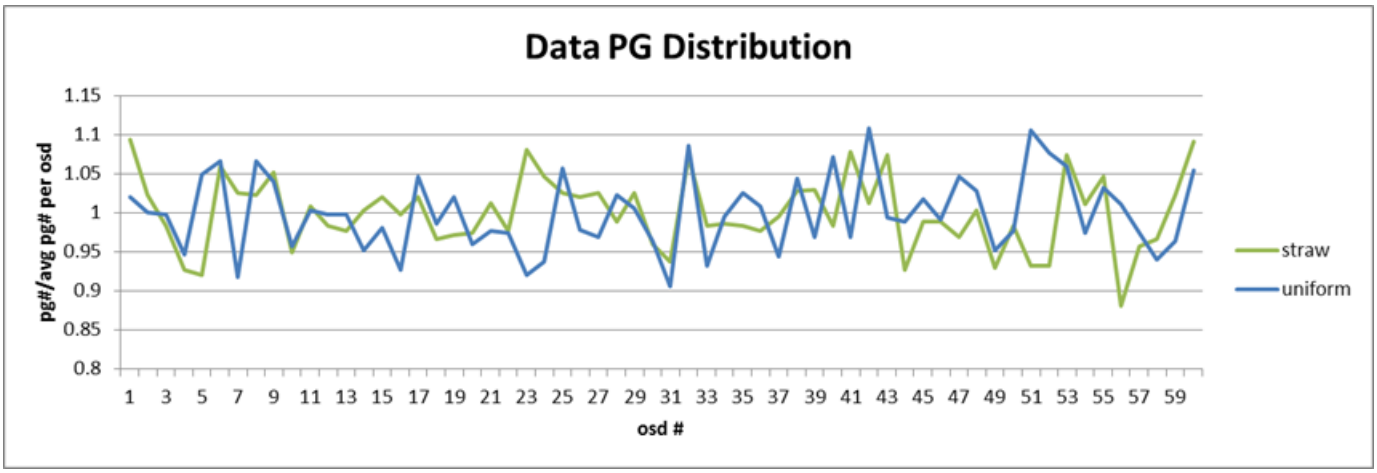
2. Turn on hashpspool, use default crush tunable, and define all buckets type as straw
   There is no significant difference of pg distribution compared to default case.

pg#	replica#	total pg#	avg pg# per osd	pg dist	pg# per osd/avg pg# per osd	stdev
60	3	180	3	0~7	0.00~2.33	0.544
240	3	720	12	6~20	0.50~1.67	0.257
960	3	2880	48	31~67	0.65~1.40	0.151
3840	3	11520	192	161~212	0.84~1.10	0.064
7680	3	23040	384	336~434	0.88~1.13	0.052
15360	3	46080	768	706~822	0.92~1.07	0.033



3. Change bucket type from straw to uniform, turn off hashpspool, and use default crush tunable
   We learned from the paper(CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data) that the placement strategy vary with bucket type. Since buckets in our cluster are all identical, we changed bucket type to uniform. However, there is still 20% gap.

bucket type	pg#	replica#	total pg#	avg pg# per osd	pg dist	pg# per osd/avg pg# per osd
straw	8192	3	24576	409.6	361~448	0.88~1.09
uniform	8192	3	24576	409.6	371~454	0.91~1.11



4. Use ‘optimal' crush tunables, turn off hashpspool, and define all buckets type as straw
There is little difference between two results.