MapX与CRUSH算法

MapX: Controlled Data Migration in the Expansion of Decentralized Object-Based Storage Systems

1. 核心

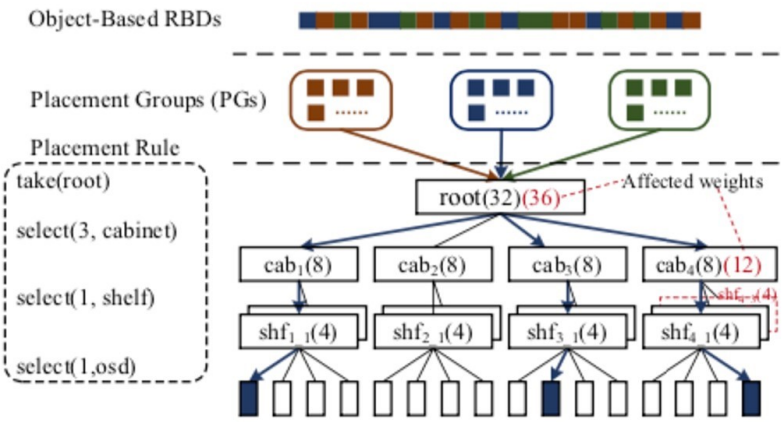
- 主要针对集群扩容/缩容场景的数据迁移过程进行优化
- 基于Ceph CRUSH进行了修改，减少了增扩容过程中的数据迁移。

2. 摘要

- 数据布局策略对于去中心化的对象存储系统的可扩展性的保证是至关重要的。CRUSH 作为最先进的数据布局算法，不依赖中心化的目录来将对象副本放置在对应的存储设备中。在受益于去中心化带来的扩展性和健壮性的好处时，CRUSH 算法在集群扩容时容易产生不受控制的数据迁移，特别是集群扩容较大规模的时候，系统性能降级现象将尤为严重。
- 该方案提出了 MAPX，基于CRUSH算法的一个扩展实现，使用了一个额外的时间维度的映射策略（对象创建时间到集群扩容时间）来保证集群扩容过程中的数据迁移可控。每一次扩容被看成 CRUSH MAP 中新的一层，被表示成为 CRUSH 根节点下的一个虚拟节点，MAPX 通过操纵 PGs 的时间戳来控制对象到层级 layer 的映射。MAPX 适用于各种各样的基于对象的存储场景，可以将对象的时间戳维护成更高级别的元数据。例如，我们将 MAPX 用于 Ceph-RBD，通过扩展 RBD 元数据来维护和检索在扩展层粒度上的对象创建时间。实验结果表明基于 MAPX 的无迁移系统比基于 CRUSH 的系统（扩展之后忙于数据迁移操作）的尾延迟降低了 4.25 倍。

3. CRUSH算法

- CRUSH 逻辑上使用了一个 ClusterMap 的结构来抽象表示集群的分层信息。集群信息如图所示有三种结构，root 节点代表一整个集群，集群由多个 cabinets 组成，也就是所谓的机架 rack，而 cabinets 又由 shelves 组成，即所谓的物理机。每一个物理机又安装了许多 OSD 节点，对应磁盘。该层次中的内部节点 root/cabinets/shelves 统称为 bucket。
- 每个OSD都有一个由管理员分配的权重，用于控制OSD的相对存储数据量，所以每一个 OSD 的负载是按照权重来进行分配的，而 bucket 的权重则是该节点对应的子节点的权重之和。如图所示，节点中数字表示节点对应的权重。



1. CRUSH Steps

- Step 1. 通过计算对象名称哈希的模，将对象分为若干个 PGs（类似于简单的 HASH 操作）
  pgid = HASH(name) % PG NUM
- Step 2. 单个 PG 的对象根据 CRUSH 算法映射到一组 OSDs（根据副本数量）

PG -> OSDs

- CRUSH 算法支持灵活的配置，可以把故障域的信息编码到 ClusterMap 中，也可以定制 Placement Rule 来明确如何通过递归选择bucket项来放置副本。通常的 Placement Rule 又大致分为四步：

1. take(root)
- 选择存储层级结构的根节点，由于是唯一入口，可以保证操作顺序执行。

2. select(3, cabinet)
- 找到三个满足故障域的 rack，3 为副本数量。
- 重复计算如下等式，根据图中的例子，则是从 四个机架中选出三个。i 指定对应四个节点中的哪一个节点，其中 pgid 为 PG 的编号，r 为argmax 计算的参数，HASH 是有三个输入参数的 HASH 函数，ID(i) 和 W(i) 则是节点 i 对应的 ID 和权重。
- 为了选择 x 个节点出来，通常该等式的执行次数大于 x 次，因为可能存在计算出来的节点已经被选择过了或者计算出的节点已经失效了。

$$C(pgid,\vec{i},r)=\underset{i\in\vec{i}}{\operatorname{argmax}}\,HASH(pgid,r,ID(i))\times W(i)$$

3. select(1, shelf)

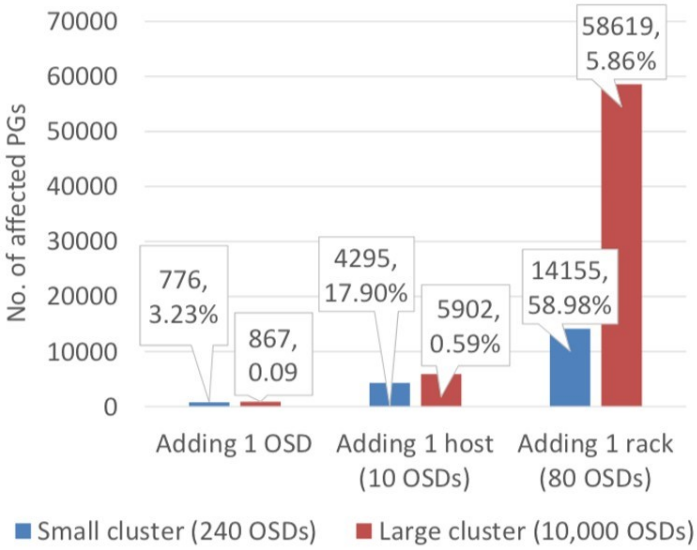
- 从 cabinet 中找到一个 shelf。机架中选择一个 物理机
- 同样执行步骤二中的等式，只是 x = 1，后续步骤一致。

4. select(1, osd)

- 从 shelf 中找到一个 osd。从物理机中选择一个 osd。
- 如上图所示，一个物理机对应了四个 OSD，每个 OSD 代入自己的 id 和 pgid，crush hash(osd id, pgid)计算出一个随机值 r（其中crush hash可以简单地看成是一个伪随机的hash函数：返回一个固定范围内的随机值。同样的输入下其返回值是确定的，但是任何一个参数的改变都会导致其返回值发生变化），每一个 osd 对应一个 r 值。用公式ri * wi计算得到 osd 的 straw 值，然后选取 straw 值最大的 osd 放置 PG。

2）CRUSH 缺点

- CRUSH 实现了不依赖中心化目录的统计意义上的负载均衡，并且可以在存储集群发生变化的时候自动调整对应的负载。但是扩容过程中可能导致不受控制的的数据迁移。如图所示，在机架4上新加一个物理机 4-3，同样带四个 OSD 节点，此时将影响对应的父节点的权重，直到根节点为止，所以将可能导致该机架上的其他物理机的数据迁移到 4-3 上，同时也可能导致其他机架的数据迁移到机架 4 上，h * ▲w/W。h 为改分层结构的层数，▲w 为扩容造成的权重的变化，W 为所有OSD节点的权重和。
- 实验测试，两个 Ceph 集群，均为三层结构，三副本，一个机架作为故障域，一个机架由八个 host 组成，每一个 host 十个 OSD 节点。集群 1 对应 3 个机架，24 个物理机，240 个 OSDs，24000 PGs，集群 2 有 125 个机架，1000 个物理机，10000 个 OSDs，100w PGs。两个集群都执行扩容操作，粒度分别为 一个 OSD，一个物理机，一个机架。结果如图所。如果扩容规模较大时，可能会有接近 60% 的 PG 将会被影响，在整个迁移期间不可避免地会导致性能降级。



4. MAPX

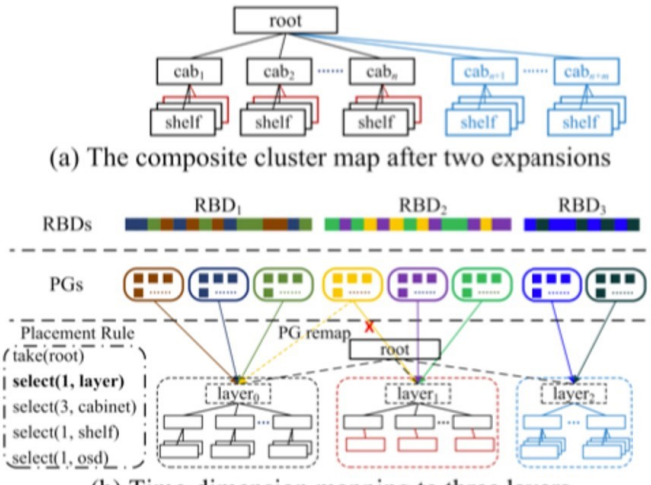
- 之所以 CRUSH 算法扩容过程中会导致大量的数据迁移，是因为该算法破坏了 新旧对象/OSD 之间的差异。为了解决这个问题，MAPX 延伸了通用的 CRUSH 实现，引入了额外的时间映射机制。

1）可控数据迁移的扩容

- 以 Haystack and HDFS 为代表的中心化目录管理的存储系统为了避免数据的迁移，往往不惜以短暂的负载不均衡为代价。随着新对象存储到新的 OSD 中，可用的容量对应随着时间下降，因此整个系统将渐渐实现负载均衡。数据迁移可以根据需求来决定何时执行。
- 受中心化数据布局策略的启发，致力于实现扩容过程中数据迁移的可控，所以基于 CRUSH 设计实现了 MAPX，核心思想就是引入时间维度的映射机制来区分 新老 对象/OSD，同时保留 CRUSH 算法随机和均匀的优点。

集群变化

- 如图 a 描述了两种扩容情况，第一种红色框表示的给每一个机架添加一个物理机，第二种蓝色框表示的给集群添加 m 个机架。此时不再像通用的 CRUSH 那样单独更新 ClusterMap，MAPX 将每一次扩容和原本的集群当作一个单独的层，其中不仅包含新的叶子osd，而且包含从叶子osd到根的所有内部 bucket。



实现思路

- 为了尽可能少地修改原有的 CRUSH 算法，在原本的 CRUSH 根节点下插入一个虚拟层，如图 b 所示，每一个虚拟节点代表一次扩容。虚拟层对应的通过使用 MAPX 实现可控数据迁移，通过在执行原本的 CRUSH 算法之前将新的对象映射到新的 layer，因为新的 layer 不会影响原有 layer 的权重，原有对象的放置还是和以前一样，不会发生改变。

算法流程

分为Mapping objects to PGs和Mapping PGs to OSDs

Mapping objects to PGs

- 每一次扩容，新的 layer 将被分配指定数量的新创建的 PGs，且每一个 PGs 会带上一个时间戳信息 t_pg，该时间戳信息等于该新 layer 的产生时间，也就是扩容时间 t layer。
- 当读/写一个对象 O 的时候，该对象携带创建时间信息 t0，先计算该对象对应的 pgid。其中
- name 为 object name
- INIT PG NUM[i] 是第 i layer的初始 PG 数量
- 第 j layer 有最近的时间戳 tl < t0，即最新的 layer
- 尽管 PG 可能会因为负载均衡被映射到其他 layer，但 INIT PG NUM 是常量，因此从对象到 PGs 的映射是不会变化的。所以每个对象在创建时 都会被映射到一个确定的 PG，该 PG 会有所有 PGs 中距离 t0 最近的时间戳 t pg < t0。
- 算法简单描述：新的对对象的 IO 请求，会被简单 HASH 到新建的 layer 对应的 PG 上。（该 PG 的计算则是通过 HASH 值对该层 PG 数量的简单取模 + 前面所有 layer 对应的 PGs 总数偏移）

$$pgid = Hash(name) \bmod \mathrm{INIT\_PG\_NUM}[j] + \sum_{i=0}^{j-1} \mathrm{INIT\_PG\_NUM}[\mathrm{i}],$$

- 如图b所示，假设有三个 RBD，分别在layer0，layer1，layer2的扩容后创建，RBD1，RBD2，RBD3 将使用三层的 INIT PG NUM 分别计算他们在 layer0，layer1，layer2 的 PGs。

Mapping PGs to OSDs

- MAPX 类似于 CRUSH，会把 PG 映射到一组 OSDs，该过程按照用户定义的 Placement Rule 中定义的顺序去执行，如图所示，MAPX 暗中增加了一个操作 select(1, layer)，是为了实现时间维度的从 PGs 到 layers 的映射，而不需要管理节点的参与。
- MAPX 延伸了 CRUSH 算法中 select 操作的实现，以支持 layer 类型的 select 操作。算法如下所示，如果不是 layer 类型，仍按照 CRUSH 处理，否则将初始化一个 layer 数组用于存储所有位于正在处理的 bucket （通常为 root）下的 layers，并按照 layer 的时间戳排序。同时也会初始化 layer num 来表示 layers 的数量以及 pg 和 output list （select 出来的结果）。循环则是将 layers 中指定个数的 layers 添加到输出列表中。大多数场景下，number 均为 1，即只需要选出一个 layer 即可，但也不排除需要更多的 layers 的场景，例如需要选两个 layers，其中一个作为另一个的镜像。

Algorithm 1 Extended `select` Procedure of MAPX

```
1:  procedure SELECT(number, type)
2:      if type ≠ "layer" then
3:          return CRUSH_SELECT(number, type)
4:      end if
5:      layers ← layers beneath currently-processing bucket
        ▷ each layer represents an expansion
6:      num_layers ← number of layers in layers
7:      pg ← current Placement Group
8:      o ← Φ                                          ▷ output list
9:      for (i = num_layers − 1; i ≥ 0; i − −) do
10:         layer ← layers[i]
11:         if layer.timestamp ≤ pg.timestamp then
12:             if layer was chosen by previous select then
13:                 continue
14:             end if
15:             o ← o + {layer}
16:             number ← number − 1
17:             if number == 0 then
18:                 break
19:             end if
20:         end if
21:     end for
22:     return o
23: end procedure
```

- 需要注意的是，对象的副本没必要放在最新的 layer 上。例如，假设新建的 layer2 新增了两个机架，但是后一步的操作需要选出三个机架，根据 CRUSH 回溯的机制，将造成 (select(1, layer)) 被调用两次才能满足规则：当一个 select 操作在 layer 之下不

能选出足够的节点，MAPX 将保留已经选中的节点，然后回溯到根节点，再去前一个 layer 中选择缺少的节点，从而避免回溯造成的一个 layer 被多次选中。双重保证使得在这类情况也能得到处理，前面提到的例子中则将先返回 layer2 再返回 layer1。

2）数据迁移控制管理

- MAPX 能保证每一层的负载均衡，主要是因为通用 CRUSH 算法的随机性和均匀性，随着时间的迁移，新的 layer 中的数据增多到和前一个 layer 时则实现了 layers 层面的负载均衡。但是一个 layer 的负载可能会因为 对象的删除、OSD 的宕机发生一些不可预测的负载变化。如图所示，当 layer1 中的负载跟原始集群 layer0 的负载一样高时，则可能会执行一次扩容产生 layer2，假设 layer1 中执行了大量的对象删除操作，则会造成不同 layers 之间的负载不均衡。
- 为了解决这个问题，MAPX 设计了三种灵活的策略来动态管理 MAPX 中的负载：
  - PG 重映射：可以控制 PGs 到 Layer 的映射，来保证 layers 负载均衡
  - 集群缩容：缩容时需要进行 PG 重映射调整负载，但也要保留部分元数据来保证映射关系不变
  - layers 合并：使用时间戳来保证物理层的变化在逻辑层 layer 上保持相同以实现负载均衡

PG remapping

- 每一个 PG 有两个时间戳，一个静态的时间戳 t_pgs，等于创建该 PG时对应 layer 的初始化的时间戳；一个动态的时间戳 t_pgd，可以被设置为任意 layer 的扩容时间。
- 对象到 PG 的映射使用了静态的时间戳，PGs 到 layers 的映射通过比较 PGs 的动态时间戳和 layers 的时间戳来执行。
- 所以通过修改动态时间戳对应的数值则可以实现将 PG 映射到指定 layer，同时也会通过内部 map 信息增量更新的方式通知所有的 OSD 节点。而对于时间戳的存储开销其实是较小的，每一个 PG 的时间戳可以使用 one byte 的索引来指向对应 layer 的初始化时间，故可以支持 2^8 = 256 个 layers，假设一个机器有 20 个 OSDs 对应 200 个 PGs，那么1000台机器组成的集群的时间戳开销则为 1000 x 20 x 200 x 2 x 1B = 8MB。

Cluster Shrinking

- 当一个 layer 负载低于设定的阈值之后，MAPX 将移除该 layer 中多余的机器或者 OSD 从集群中，相当于扩容的反向操作。
- 假定 layer a 将要从集群中被移除，首先会将 layer a 对应的所有 PG 分配给余下的其他 layer，并按照其他 layer 的权重进行分配（此处为了简化重分配的过程没有考虑 layer 实际的负载情况），然后使用上一节提到的重新映射来进行数据的迁移。
- 在移除 layer a 之后，即执行了缩容操作之后，layer a 的逻辑意义不会发生改变，特别是 INIT PG NUM（仍然保留），但是不会保留任何物理设备，逻辑意义不改变是为了保证对象到 PG 的映射的关系保持不变。

Layer merging

- 层级之间的合并可以巧妙利用前文提到的 PGs 到 OSDs 的映射机制，假设要合并 layer a 和 b，可以直接将 a 的扩容时间 t_layer 直接设置为 b 的扩容时间，此时在逻辑意义上 a 和 b 就是同一 layer 了。

5. 在 CEPH 中实现 MAPX

- 从上图中我们可以发现内部的 bucket 在 MAPX 中可能会同时属于多个 layers（譬如只进行了 OSD 层面的扩容），因此我们将内部设备分配到一个特定的 layer，例如在一个特定虚拟节点之下，使用虚拟设备 ID 将物理设备 ID 和 layer 时间戳连接在一起，使用虚拟节点的权重对应的属性记录 layers 的时间戳，将在选择 layer 的过程中和 PG 的动态时间戳进行对比。
- MAPX 不能为每一个对象都维护一个时间戳信息，因为这样的开销和中心化目录的开销就大致相同了，所以 MAPX 不适用于那种最底层最用的对象存储系统，而是适用于那些可以使用上层一点的元数据管理的对象存储系统。

1）Ceph RBD

- MAPX 使用了 rbd header （RBD本身的元数据管理数据结构）来管理时间戳信息，当一个客户端使用 rbd open 来挂载 RBD 时将检索 rbd header。因为RBD的对象可以在任何扩展之后创建，所以我们继承当前 layer 的时间戳（创建该 layer 的时间）作为对象的时间戳，因此我们在 rbd_header 中为每一个对象添加了 object_timestamp 属性，该属性对应指向 layer 的创建时间戳。假设每一个对象对应的索引属性的大小为 one byte，每个对象的大小是 4MB，那么对于一个 4TB 大小的 RBD object_timestamp 数组的开销为 1MB。

2）CephFS

- CephFS 的元数据对应存储在 inode 节点中，客户端打开一个文件会去获取对应的创建时间，此时对应地会访问 inode 数据，现在我们让一个文件对应的所有对象都继承该文件创建的时间戳，这样就能控制一个文件对应对象的时间维度的映射。假设一个文件的大小达到了阈值 T，假定 T = 100MB，我们会将其分为多个小于 100 MB 的子文件，文件的元数据中会维护源文件到子文件的映射信息以及每一个子文件的创建时间，所以同样可以控制子文件对应的时间维度的映射。

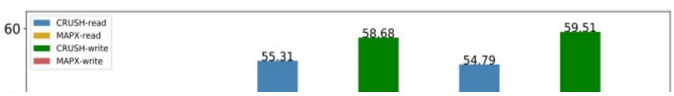
6. 测试

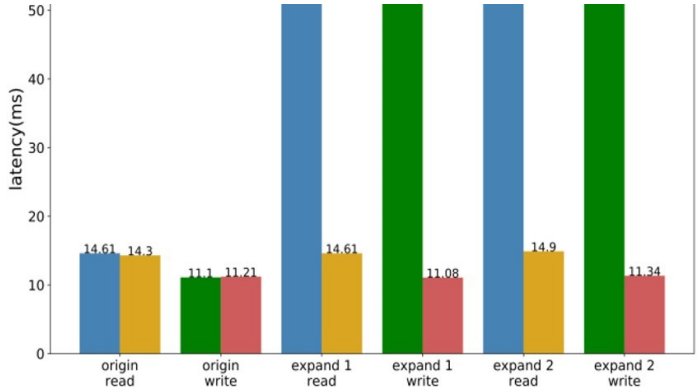
- 比较对象：MAPX 和 传统 CRUSH
- 硬件环境：3台主机，每台主机对应：20 核 Xeon E5-2630 2.20GHz CPU，128G RAM，10GbE NIC，5.5 TB HDDs
- 软件环境：OS：CentOS 7.0，Ceph：12.2 Luminous，BlueStore，Monitor co-located with one of the storage servers

Client：fio benchmark

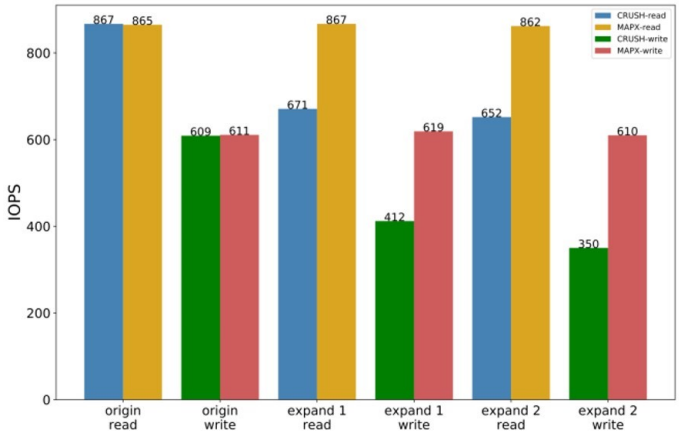
1）扩容过程中的 IO 性能

- 参数使用 Ceph 默认参数，除开 OSD max back fills 前面中提到过 Ceph 自身对 CRUSH 造成的数据迁移的优化是通过设置 OSD max back fills >= 1 来对性能降级过程的严重程度和持续时间来做权衡。该参数默认值为 1，将使得迁移的优先级最低，所以在迁移操作完成之前 PGs 中的对象的数据迁移速度将很慢，相应地将严重延长数据迁移的时间，同时增加节点对应的写负载：因为对 等待迁移的 PGs 的写操作，需要首先在原始 OSD 上执行，然后再异步地迁移到目标 OSD，该方式性能降级现象表现得较为轻微，但会持续很长一段时间。我们将 OSD max back_fills 设为 10，该参数在此次实验中更为合理，因为这样迁移操作才会有更高的优先级来显示 MAPX 和 CRUSH 之间的区别。
- Ceph 初始集群，三台存储机，每台机器两个 OSDs，三副本，128 个 PGs，对应每个 OSD 会负责 64 PGs。创建 40*20GB 的块设备，我们会给每个机器添加一到两个 OSDs，然后测试 MAPX 和 CRUSH 下对应的性能表现（I/O 延迟和 IOPS），I/O 大小为 4KB，FIO 的 iodepth 分别为 1 和 128，对应延迟和 IOPS 测试
- 延迟测试实验结果如图所示，初始集群大小为 6 个 OSDs，然后分别扩容到 9 个和 12 个。云存储通常只关心尾延迟 99th，99.9th or 99.99th，因为尾延迟对于服务等级影响相比于其他比例的延迟最大。MAPX 相比于 CRUSH 降低了 4.25x 的延迟，主要是以内 CRUSH 中的数据迁移将和普通的 IO 请求发生严重的资源竞争。



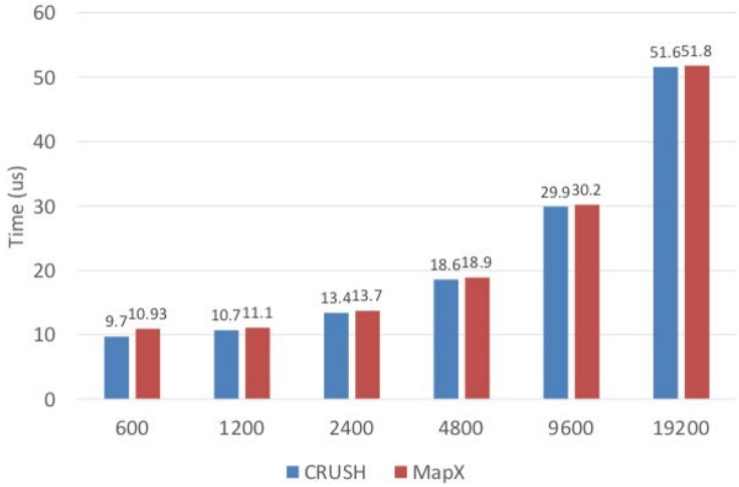


- 下图显示了 IOPS 测试结果，每个结果都 run 了 20 次，忽略了部分误差，因为占比较小。结果显示 MAPX IOPS 优于 CRUSH 约 74.3%



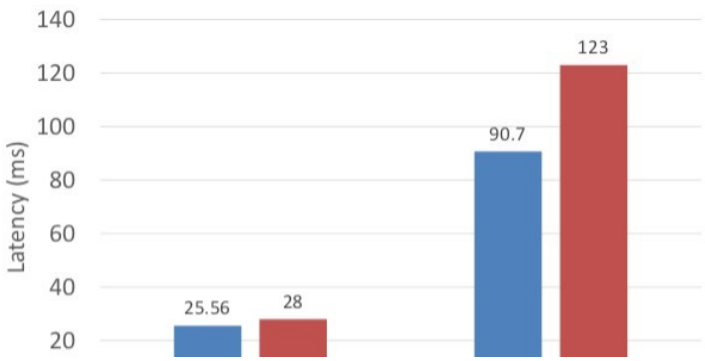
2）计算开销

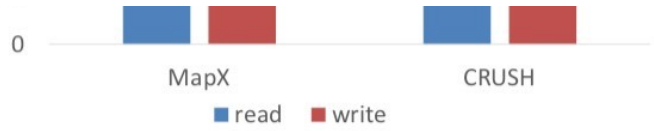
- 模拟大约有 600-19200 个 OSDs 的集群来比较 MAPX 和 CRUSH 的计算开销，CRUSH 和 MAPX 均可以在 10us 内完成对象到 OSD 的映射，MAPX 相比于 CRUSH 略高因为需要时间维度上的映射计算。



3）缩容过程中的 IO 性能

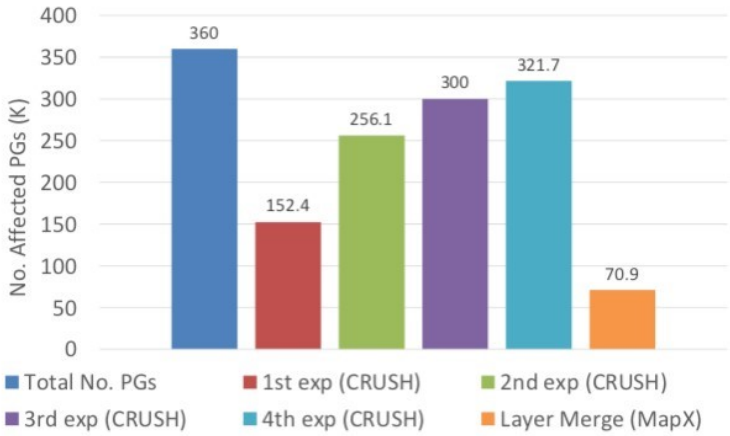
- 三台机器，每个机器三个 OSDs，先给每一台机器加一个 OSD，然后每一个机器减去一个 OSD，为了控制迁移的速度，将并发迁移PG的数量设置为 8
- 图示均显示了 99th 的尾延迟。图示结果并不一定表明缩容过程中 MAPX 的延迟比 CRUSH 低，因为采用了不同的调节机制，因为移除一个 OSD 对于 CRUSH 而言造成的是权重降低然后导致不必要的数据迁移，但对于 MAPX 而言因为根本没有造成之前节点的权重变化，缩容只是 layer 层面上的，根本不会发生数据的迁移。





4）层级之间的合并

- 使用了 CrushTool 来模拟 MAPX 中的层级合并。三副本策略，一个集群对应五个机架，一个机架对应20台物理机，一个机器对应20个OSDs，即100台机器，2000个OSDs，20w PGs，扩容四次，每次加一个新机架，20台物理机，400个OSDs，4w PGs，MAPX将所有新加入的 PGs 映射到 OSDs 上因此没有迁移发生，四次扩容后将第一次扩容和第二次扩容总计 40 台机器的进行了合并，然后测量受影响的 PG 数量。
- 结果表明对应合并的 8w 个 OSDs，其中受影响的 OSDs 约为 70910，由于 CRUSH 没有合并操作，就只是只能对扩容操作影像的 PG 数量进行了了测量。



7. 拓展

1）负载均衡 & 迁移开销

- Ceph 通过降低迁移的优先级来避免扩容造成的突发的数据迁移，但是由于 CRUSH 算法的特性，迁移不可避免，只是时间早晚的问题，保守的迁移控制只会延长迁移的时间，但这又导致了等待迁移的 PGs 的数据写入的复杂性，不可避免低会增加负载。
- 其他的一些去中心化的分布式存储系统使用了分布式哈希表的一致性哈希算法来实现数据路由。相比于CRUSH，DHT不能表示集群的层级结构，需要额外的机制来进行建模，相比于 CRUSH 不够灵活。

2）存储系统

- 去中心化对象存储系统：Ambry，F4。
- 中心化对象存储系统：Haystack，Lustre，HDFS
- 块存储系统：Ursa，Salus，Blizzard，PARIX
- 文件存储系统：GFS，Zebra，BPFS，OptFS

3）Conclusion

- 在大规模存储系统领域关于中心化和去中心化的争论已久，去中心化的 CRUSH 表现出了高可扩展、健壮性和性能上较好，但在扩容过程中会造成不受控制的数据迁移，MAPX引入了额外的时间粒度映射机制，同时继承了 CRUSH 算法的随机性和均匀性。未来将致力于减少对象时间戳存储的开销，以便于将 MAPX 应用于更广泛的基于对象存储的存储系统。