# 分布式存储的元数据设计

李道兵 <lidaobing@gmail.com>
七牛云存储
2015-04 北京

# 引子

- 面试新人时最经常被问到的一句话：七牛的云存储真的是自己写的么，是不是基于 Hadoop 的?

# Outline

- 无中心的存储设计: glusterfs
- 有中心的存储设计： hadoop
- 基于数据库的存储设计: gridfs, hbase
- 绕过问题的存储设计: fastdfs

# glusterfs

- 设计目标
  - 兼容 POSIX 文件系统: 你的应用程序无需修改就可以放到 glusterfs 上来运行
    - 无中心节点: 性能瓶颈没有了，单点故障也没有了
    - 扩展能力: 大容量存储都需要这个
- 我们在这里不讨论 POSIX 兼容的优劣，集中通过 glusterfs 来讨论无中心节点设计中普遍遇到的一些难点

# glusterfs

- glusterfs 是一系列无中心存储的设计的代表
- 无中心暗示着我们可以通过内容的 key 推算出来这个 key 的存储位置
- 在绝大部分实践中，在这种设计下如果出现单盘故障，处理模式如下
  - 去掉坏盘，换上新盘
  - 从老盘拷贝数据到新盘
  - 这个模式最大的问题是修复时间，4T盘在 100MB/s 的修复速度下需要至少 11 个小时，这么长的修复时间会导致数据的可靠性降低（在这11个小时内另外两块盘的损坏概率）
- 另外一种修复模式是在读的时候发现有坏块，然后触发修复，在这种模式下修复只会更糟糕。

# glusterfs

- 如何回避掉这个问题呢?
  - 引入中间层记录分区和物理设备的关系，这样磁盘损坏不用等换盘就可以开始修复
  - 一个磁盘分成多个区，每个区可以到不同的盘上去修复，那么可以大幅度缩短修复时间，比如分到 50 个区（每个区 80GB), 那么修复时间就可以缩小到 13分钟左右。

# glusterfs

- 扩容
  - 在无中心设计中，扩容往往伴随着数据的再平衡，再平衡会带来如下的挑战
    - 网络拥塞：可以使用独立的迁移网络来改善
    - 迁移时间长且迁移期间数据读写逻辑变得更复杂：多加测试改善代码质量

# glusterfs

- 不支持异构存储
  - 比如小文件经常伴随着很高的 iops 需求，针对小文件我们可以引入SAS或者SSD盘来得到更高的 iops, 但对于无中心存储来讲，这种方法很难实施。
  - 类似的异构需求还包括某些客户数据只想存两份，而其他客户数据则想多存几份的情况，这些在无中心存储中都是很难解决的。
  - 小文件的问题针对读取的部分可以通过缓存层来改善，但对于高频率的写入没有太好的解决方案
  - 这儿也存在一个基于 hash 碰撞的攻击方案，不过影响不大。

# glusterfs

- 数据不一致的问题
  - 比如我们要覆盖一个 key，但在覆盖过程中出现意外，导致只覆盖了三个副本中的两个或者一个。这个时候就很容易读到错误的数据。

  - 在写入文件时，先写临时文件，最后再重命名能改善这个问题，但仍然不完美。

# glusterfs

- 问题总结
  - 坏盘修复问题：元数据+分区可以改善这个问题
  - 扩容动作大: 忍
  - 小文件高IOPS: 劣势，集群足够大的话可以靠规模来抗住
  - 数据不一致的问题: 复杂度会变高

# Hadoop

- 设计目标
  - 大文件
  - offline 使用
  - 可伸缩
- 元数据(NameNode)设计
  - 主备模式，各一台机器
  - 数据尽量加载到内存，提高性能
  - 放弃高可用，进一步提高元数据的性能 (NameNode 的变更不是同步更新到从机，而是通过定期合并的方式来更新）

# Hadoop 优点

- Hadoop 为大文件服务:
  - 意味着 NameNode 不会太大，比如 64M 的块大小, 10PB 文件只需要存储 1.6亿条数据，如果每条数据 200B, 那么需要 32GB 左右的内存。
  - 元信息的 qps 也不用太高，如果每次 qps 能提供一个文件块的读写，那么 1000qps 就能达到 512Gb/s 的读写速度，满足绝大部分数据中心的需求。
- Hadoop 为offline业务服务: 高可用可以部分牺牲
- Hadoop 为可伸缩服务: 伸缩的是存储节点，元信息节点无需伸缩。

# Hadoop 为什么不能当公有云?

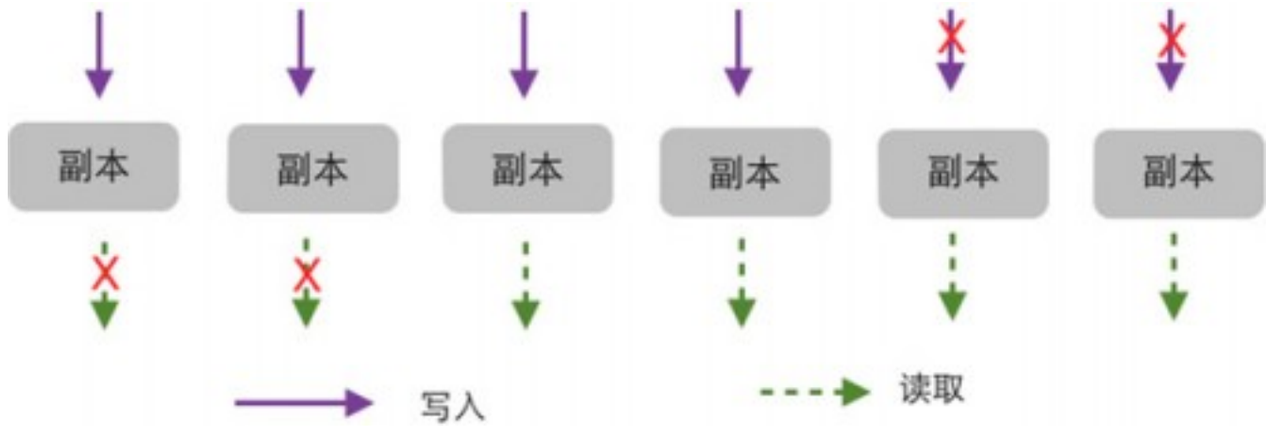
- 元信息容量太小：1.6 亿条数据就占掉 32GB，100 亿的数据需要 2000GB 内存，这个完全没法接受

- 元信息节点无法伸缩: 元信息限制在单台，1000qps 甚至 15000qps 的单机容量远不能达到公有云的需求。

- 高可用不完美: NameNode 问题

# 其他有中心设计

- WRN算法




- 写入了 W 份才算成功
- 读取时成功读取 R 份才算成功
- W + R > N (其中N 为总副本数)

图片来自：莫华枫的《云存储的黑暗面：元数据保障》

# WRN算法

- W,R,N 的选择
  - 比如2,2,3这种情况，写入两份就算成功，但如果其中一台机器下线，这个数据就可能读不出来了
  - 所以446 或者 669 这样的选择会更好。但机器越多，响应会越差。

# WRN算法

- 失败的写入会污染数据
  - 比如 446 的场景，如果写入只成功了3份，那么这次写入是失败的，但如果是覆盖写入，那么也就意味着现在有三份正确的数据，三份错误的数据，哪一个是正确就无从判别了。
  - 写入数据带版本(不覆盖，只是追加)能改善这个问题，但带来了一个攻击点：反复覆盖同一个文件，导致数据库出现性能瓶颈

# 有元数据的存储

- Hadoop
  - NameNode 不是高可用
  - NameNode 容量不足
- WRN支持的元数据
  - 响应性差
  - 有丢失数据可能性 (覆盖写) / 有攻击点 (带版本写)

# 基于数据库的分布式存储方案

- GridFS (基于 MongoDB)
- HBase
- HBase + Hadoop

# GridFS

- 基于 MongoDB
- 分块存储，每块大小为255KB
- 数据直接放在两个表里边
  - chunks: 存储数据，加上元信息后单条记录在256KB 以内
  - files: 存储文件元信息

# GridFS 优点

- 两个需求（数据库和文件都需要持久化），一次满足
- 拥有MongoDB的全部优点: 在线存储，高可用，可伸缩(*)，跨机房备份，…
- 支持 Range GET，删除时可以释放空间(需要用 mongodb 的定期维护来释放空间)

# GridFS 的缺点

- oplog 耗尽:
  - oplog 是 mongodb 上一个固定大小的表，用于记录 mongodb 上的每一步操作，MongoDB 的 ReplicaSet 的同步依赖于 oplog。
  - 一般情况下 oplog 在 5GB-50GB 附近，足够支撑 24 小时的数据库修改操作。
  - 但如果用于 GridFS，几个大文件的写入就会导致 oplog 迅速耗尽，很容易引发 secondary 机器没有跟上，需要手工修复，而且MongoDB 的修复非常费力。
- 简单来说就是防冲击能力差，这个跟数据库的设计思路有关。
- 除了前面提到手工修复的问题外，冲击还会造成主从数据库差异拉大，对于读写分离，或者双写后再返回的场景带来不小的挑战。

# GridFS 的缺点

- 滥用内存
  - mongodb 使用 mmap 来把磁盘文件映射到内存，对于 gridfs 来说，大部分场景都是文件只需读写一次，对于这种场景没法做优化，内存浪费巨大，会挤出那些需要正常使用内存的数据。
- 设计阻抗失配带来的另外一个问题。

# GridFS 的缺点

- 伸缩性
  - 需要伸缩性就必须引入 mongodb sharding
  - sharding 的情况下你需要使用 files_id 作为 sharding key
    - 如果你不修改程序的话files_id 是递增的，也就是说所有的写入都会压入同一个集群，而不是均匀分散。
- 在这种情况下你需要改写你的驱动，引入一个新的 files_id 生成方法。
- 另外，MongoDB Sharding在高容量高压力下的运维很痛苦（大家可以参考百度网盘组之前的一些 PPT)

# GridFS

- 低压力: 没问题，挺好用的
- 中压力: 如果单台机器能抗住你的存储，建议分离数据库和GridFS, 使用独立的机器资源
- 高压力: 不建议使用 GridFS

# HBase

- 前面提到 Hadoop 因为 NameNode 容量问题所以不合适用来做小文件存储，那么 HBase 是否合适呢?

# HBase 的优点

- 伸缩性，高可用都在底层帮你解决了
- 容量很大,几乎没有上限。

# HBase 缺点

- 微妙的可用性问题
  - 首先是 Hadoop NameNode 的高可用问题
  - HBase 的数据放在 Region 上，Region 会有分裂的问题，在分裂和合并的过程中，这个 Region 会不可用
  - 我们可以采用预分裂来回避这个问题，但这就要求 预先知道整体规模，并且key 的分布是近均匀的
  - 在多租户的场景下，key 均匀分布很难做到（除非舍弃掉 key 必须按顺序这个需求）

# HBase 的缺点

- 大文件支持
  - 10MB以上的大文件支持不好
  - 一个改良方案是把数据拼装成大文件，然后 hbase 只存储文件名，offset 和 size
  - 这个改良方案其实挺实用的，不过如果要做到空间回收就需要补很多开发了。

# HBase方案

- HBase存元数据，Hadoop 存数据算一个可用方案，但是
  - Hadoop是 offline 设计的，NameNode的高可用考虑不充分
  - HBase的 Region 分拆和合并会造成短暂的不可用，如果可以的话最好做预拆，但预拆也有问题
  - 如果对可用性要求低的话问题不大

# 绕过问题也是解决问题的方式: fastdfs

- fastdfs:
  - hadoop的问题是NameNode 压力过高，那么 fastdfs 的思路就是给 NameNode 减压。
  - 减压的方法就是把 NameNode 的信息编码到key里边
  - 范例URL: group1/M00/00/00/rBAXr1AJGF_3rC-ZAAAAEc45MdM850_big.txt
  - 也就是说 NameNode 只需做一件事情，把 group1 翻译成具体的机器名字

# fastdfs 的优点

- 结构简单，元数据节点压力低
- 扩容简单，扩容后数据无需重新平衡

# fastdfs 缺点

- 不能自定义 key: 这个对多租户是致命的打击，自己使用也会减低灵活性
- 修复速度:
  - 磁盘镜像分布，修复速度取决于磁盘写入速度，比如 4TB 的盘, 100MB/s 的写入速度，那么需要至少 11个小时
- 大文件容易造成冲击
  - 首先是文件大小有限制(不能超过磁盘大小)
  - 其次是大文件没有分片，导致大文件的读写都由单块盘来承担，所以对磁盘的网络冲击很大

| | 优点 | 缺点 |
| --- | --- | --- |
| 无中心设计 | 无高压力节点 | 修复慢，扩容难<br>无异构支持<br>数据不一致 |
| 有中心设计 | 扩容，修复更灵活 | 中心节点难设计 |
| 基于数据库的设计 | 简单，易上手 | 设计失配<br>容量有限 |
| fastdfs | 中心压力小，易扩容 | key不能随便重命名<br>大文件支持差 |

# Q&A