通用可扩展分布式对象存储

Draft v0.0 by Baul

一 需求

CDN / 视频处理 高效存储和查询 写一次，经常读，但很少改写，不频繁删除

挑战

- 几k到几GB，小文件和大文件高效处理
- 海量，并持续增长，要扩展
- 负载不均衡
- 上传要快，高可用性

二 现有技术

分布式文件系统

> 目录结构和元数据导致高负载和延迟。
> gfs / nfs / hdfs / ceph

kv存储

> 只适用于小对象，对几十MB到GB的大对象没有优化；提供了强一致性，对于我们这种需求不必要
> bigtable / dynamo / cassandra等。

haystack / f4/twitter的blob store

> 解决了元数据IO瓶颈，但是具有负载不均衡问题，特别是在扩展时。

三 设计目标

> 大小文件支持，写一次，读多次。

1. 低延迟和高吞吐量

系统page cache / zero copy read / 分块chunk并行读写 / 存储配置策略（大小，压缩，复制因子，复制策略（架构感知 / 跨地区），客户端缓存，磁盘类似（自动分层），等等） / 零成本错误检测机制

2. 跨地区操作支持

数据要复制到多个地区，非中心化的多master架构，数据可以从任一个副本来读写。采用异步机制写到临近地区，然后异步复制到其他地区。利用请求代理把当前地区没有的副本请求转发到其他地区。

3. 可扩展

低成本增量扩展。
对象逻辑位置与物理位置分离设计，透明改变物理位置；非中心化架构，没有主master之分；落盘分段的object indexing和bloom filter 并对最新的indexing分段进行mem缓存。

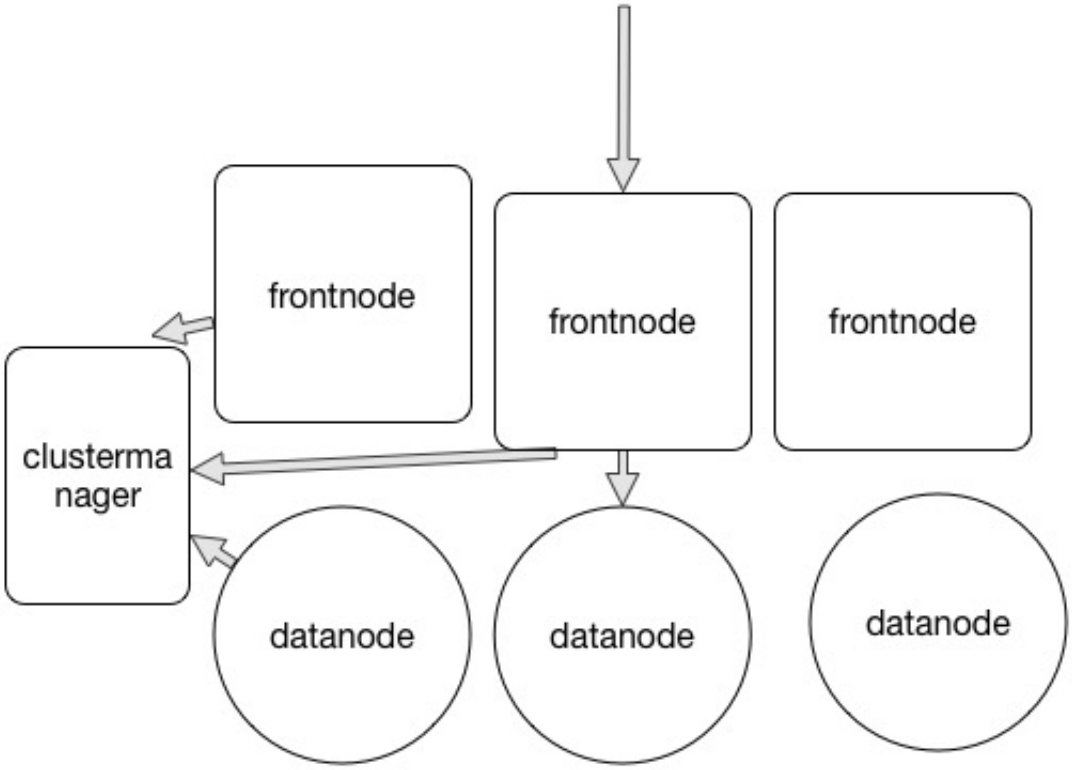
4. 负载均衡

静态集群下，利用chunk和随机选择定位来达到负载均衡，在扩展集群时，启动rebalancing机制。

5. 用户友好的管理接口，基于partition或者virtual disk / container等类似概念提供的多租户，并能应用特定策略（如压缩 / replication机架感知 / 地区感知 / 去重 / 存储分层 / 等等）

6. 为将来功能扩展提供便利比如快照等

四 系统架构概揽



仅表示单个本地集群，采用复制方式与异地集群交互。

1. frontnodes接受并路由请求

接受来自客户或者cdn的put / get / delete等请求，转发到对应的datanode并返回响应，内部实现路由机制。

2. clustermanger维护集群状态

以partions组织数据，由许多对象组成，实现上一个partions就是一个文件，并有多个副本。一旦创建，其状态就为读写，可以附加对象。当这个逻辑分区partions达到其容量，变为只读。cm跟踪partions的状态和其每个副本的位置还有集群的物理布局如节点和磁盘位置。

创建partition / vd。

3. datanodes（存储实际的数据）

frontnode与datanode互相独立，多个clustermanager之间用zookeeper同步。
datanode保存并查询对象数据。每个datanode管理多个磁盘。为了高性能，datanode维护对象在partions上的indexing结构（主要是objectid和在partition里的offset），journal和bloom filter。

五 设计细节

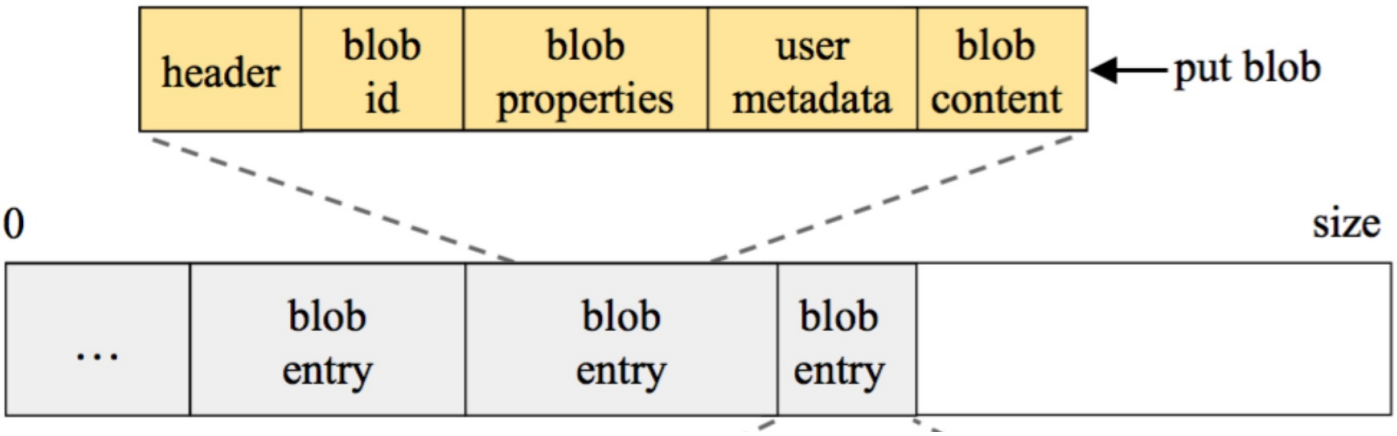
1. 把对象放哪？逻辑partition

不用hash / map方法，而采用逻辑 / 物理物质分离的方法。利用逻辑partition来放置object。
把要写的对象随机归类到一个虚拟组内即partiton里。分区的物理位置由单独的一个进程确定，保证逻辑和物理位置的解耦，有利实现rebalancing。

2. 对象如何存储 （datanode实现）

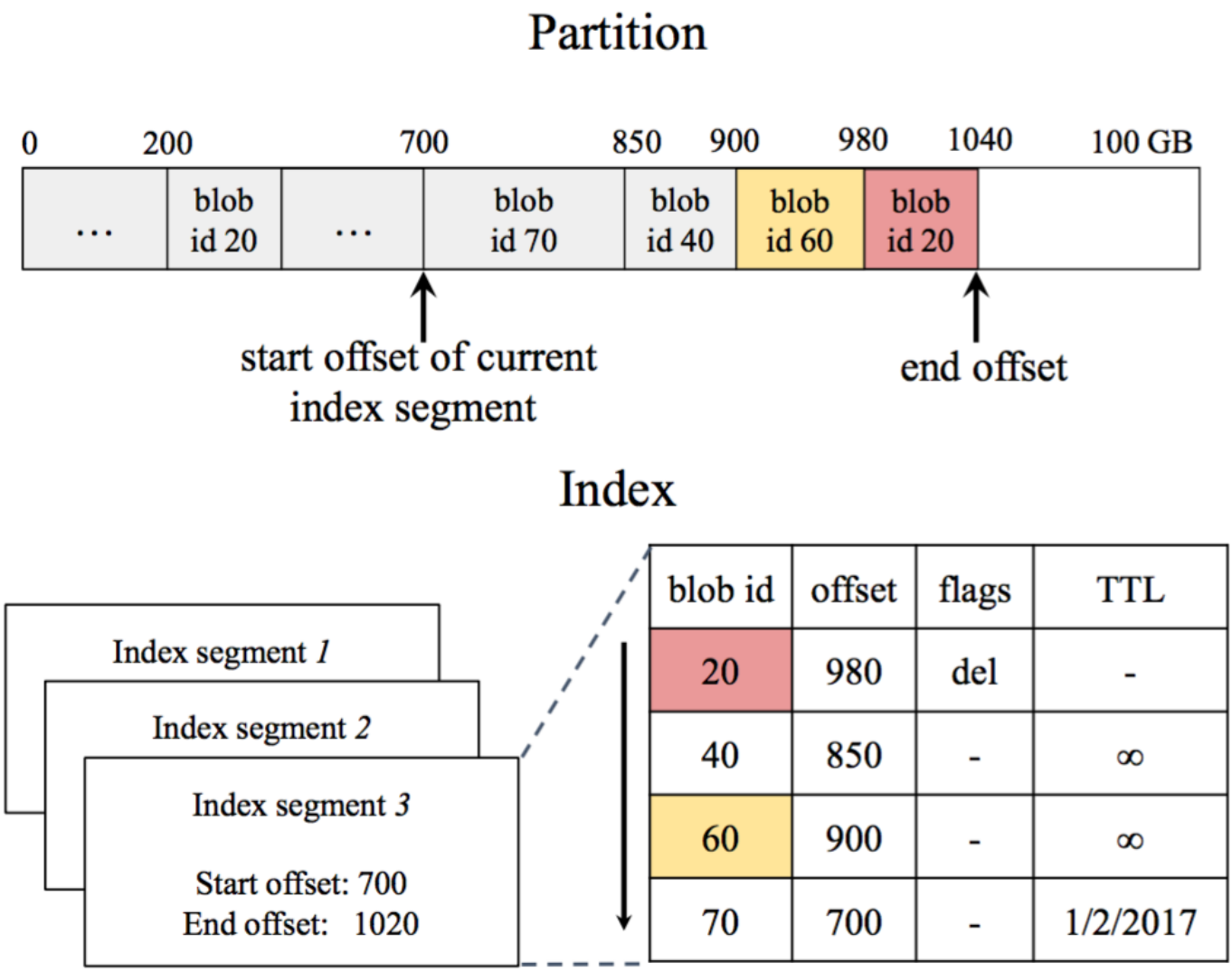
实现上预先分配的一定大小的文件，每个对象顺序附加写（put / delete）的方式添加到其中，直至partion满了。
delete采用设置tag的方式表示删除，原来的对象仍然存在。，随后由定期的compact机制进行实际空间的删除。

partition结构



partition结构如图：每个object entry（blob entry）由heaher（entry中各个域的偏移），ojectid（由frontnode生成，包含partitionid，oject的uuid），预先确定的属性也即系统元数据（如object大小，生命周期，创建时间，内容类型等），用户定义的属性即用户元数据，object内容。这里元数据与数据一起写到文件系统中。实现上也有把元数据单独作为一个实体实现，而不是与数据同处一位置。

object indexing



如图，对分区的object（blob） index，采用index segement分段的方式，针对某个index segement，只维护objectid，offset，flags，TTL等信息。

对小写批处理阻塞写，对大写chunk并发写

缓存partition fd，只打开一次，永久缓存。

3. 如何选择对象位置？frontnode实现

每个地区有自己的一套frontnodes。没有master / slave之分和也没有协调。定期从cm中获取状态，frontnode本身无状态。 具有三个功能

请求处理

- 具体的路由机制,四个过程
  - 对于写，随机选择一个partition，对于读根据objectid来找到partition。再根据客户选择的策略（副本策略 / 其他）进行操作

1. 具体的路由机制： 可以基于libcrush的方法，考虑机架感知，复制数影响，优化rebalancing。

- 分片：对大object分别，小对象合并写），保证chunk大小在4MB－10MB。
- 错误检测：跟踪同一磁盘的连续两个请求，以此来判断失败的存在（零成本错误检测）
- 代理请求：当此frontnode没找到对应的副本时，转发请求到其他frontnodes。

- 可以基于libcrush的方法，考虑机架感知，复制数影响，优化rebalancing。

安全检查

```
    病毒扫描，认证
```

通知改变

```
    可以利用对读写事件的通知，在外层实现一些分析处理，比如用户文件索引，元数据查询系统等。kafka / estatisearch等。
```

4. API支持

```
    s3/swift操作接口兼容
    restful接口支持
    c / python等API支持
```

5. 请求流程

```
    frontnode收到请求后，一些安全检测后，利用routerlibrary，选择partions，与datanode通信，完成请求。对于put写，随机选择partion，确保负载均衡。对于读，根据objectid确定分区。
    操作根据多个master的方式，根据策略，确定此操作涉及多少个副本。
    同步写到本地数据中心或地区，算req成功，然后异步写到其他地区。
    在一个frontnode本地没有要读的object时，可以转发到其他frontnode。
```

6. 负载均衡设计

```
    chunk方法 / 写操作随机选partiton，热点被cdn吸收；
    读写的partiton / 读分区的比例不平衡的情况下，根据磁盘使用率和req速率，进行rebalancing操作。
```

7. 缓存

```
    可以在frontnote节点进行读写缓存
```

8. 自愈过程设计，单独的复制进程

9. cm

```
    cm之间用zookeeper同步。
```

硬件布局

```
  数据中心 / datanode / disk，以及每个磁盘的容量和状态
  hdd/sdd混合  可以自动分层设计
```

逻辑布局

```
  分区副本的逻辑位置到物理位置的映射与分区状态的维护
  cm要定期检查datanode，维护这些信息
```

实现上为了提高吞吐量和性能，多个集群之间的协调通过zookeeper来同步。

```
  对外相当于与一个存储集群。如果内部由四个zookeeper存储集群，写速度会提升四倍。四个集群协调写，更快。
  frontnode利用dht，映射读写到不同的集群，提高吞吐量性能。
```

10. 存储资源池的讨论

```
 保证形成的partition在某个节点故障的情况下，能快速恢复。
```

11. 其他要考虑的问题

- 对象存储后端
  1. 对小对象处理的目前常见采用写大文件然后以附加方式实现如haystack / tfs / twitter blobstore / ambry
  2. 可选的高性能方案类似ceph的bluestore，利用kv存储存储元数据同时写block设备，本质就是裸盘写的过程。但是其中如何处理小对象，需要进一步细化。
  3. 基于file来实现对象的方案很多，如ceph的fielstore / glusterfs的ufo / omino / swift / orangfs
- compact / 空间回收
- 纠错码实现，替换复制的方法
- 实现语言 内嵌并行分布式能力的语言，erlang / rust / go

六 参考资料

```
ambry haystack minio ceph swift hedvig Seaweed-FS etc
```