毕业设计——海量小文件存储系统设计论文开题报告

赵海凯 16307110258

毕业设计目标是设计一个基于vfs或者fuse(用户态文件系统)的，适用于海量小文件存储，场景是读远大于写的分布式文件系统。

并且要求文件以目录结构作为组织。

第一部分 HayStack 论文思想介绍

facebook在2012年发表了《**Finding a needle in Haystack: Facebook's photo storage**》论文，实现了小文件（图片）的高性能的存储系统。

传统文件系统检索小文件的弊端

传统文件系统用于海量小文件存储的痛点在于访问文件的时候会发生大量的磁盘IO, 访问一个文件至少需要3次磁盘IO：

1.从inode表中找出**文件名与inode的映射关系**

2.从磁盘中加载**inode**到内存中

3.根据inode信息把文件对应block加载到内存中

HayStack 的设计思想

haystack的设计思想可以总结为两点：

1.把小文件集合成大文件，从而减少文件数，从而减少metadata的数目与所占存储空间。以实现在内存中检索metadata。（此处的metadata具体指的是inode）

简单理解就是把若干个图片存储在一个大文件(Volume)

2.尽可能精简inode的信息，只提供基本的POSIX语义信息。

HayStack 的架构

主要有三个组件：

- Haystack Store
- Haystack Directory
- Haystack Cache

简单理解为：存储，目录和缓存。

存储：

有两个概念：physical volumes和logical volumes

Store用于管理小文件的元数据组建。单台主机的存储划分为多个 physical volume ， 单个physical volume可以理解为一个大文件,若干个小文件存储在一个physical Volume中。

不同主机上的若干个physical volumes组成一个logical volume, 一个logical volume下的所有physical volumes互为备份，用于容错和请求的分流。

访问图片的时候，通过**logical volume**的id和offset来快速定位一个小文件。

而且Haystack的核心在于**不需要通过硬盘**访问来获得某个图片的文件名，偏移量和大小。因为**每个物理节点**会一直在内存中维护每个物理卷对应的**文件描述符**和**小文件id到metadata**的映射关系。

下面来讲解volume的数据结构设计：

每个volume就是一个包含superblock和一系列needle的大文件。

每个needle保存图片的元信息和图片本身。

其中flag用于标记图片是否删除，padding用于硬盘块对齐

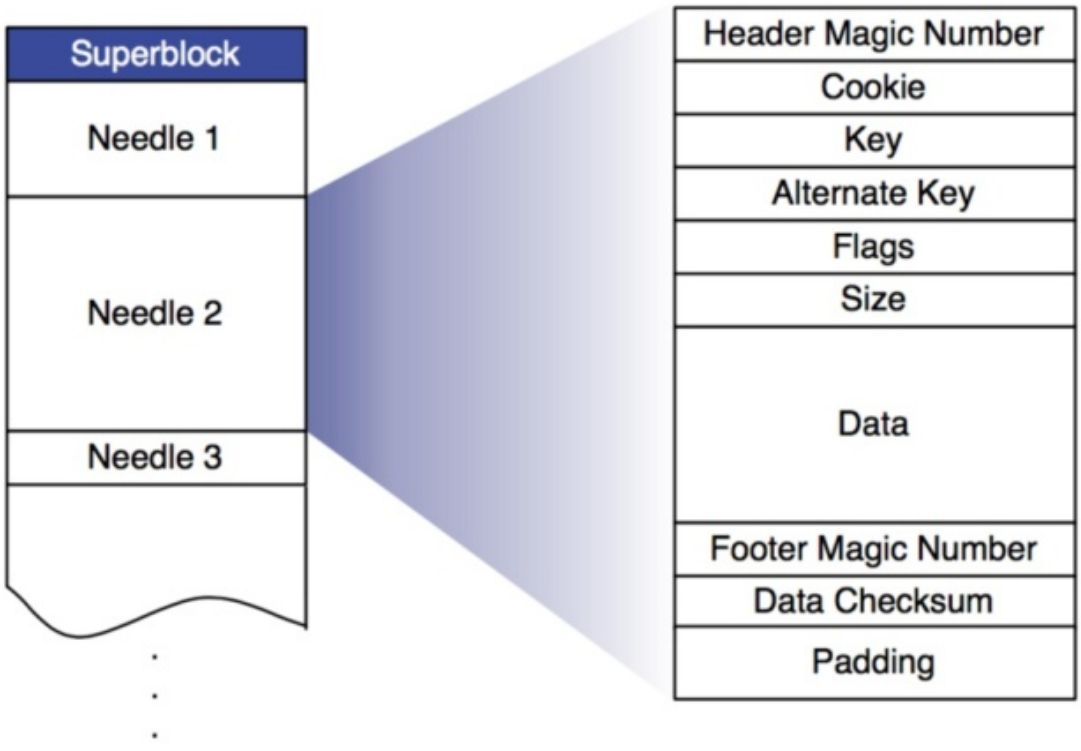
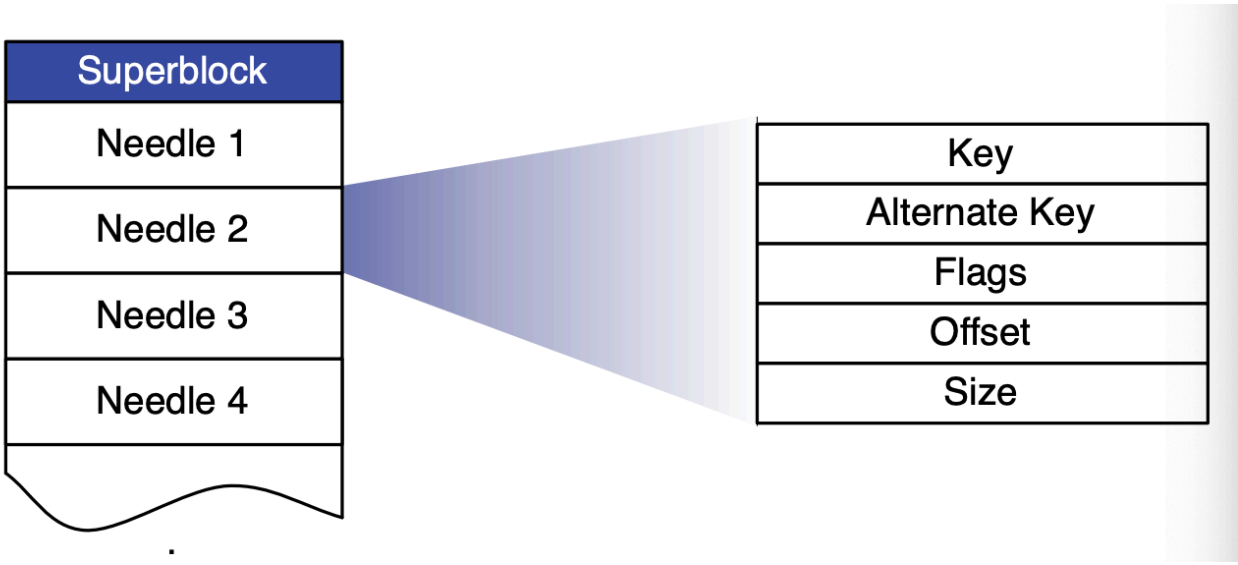


Figure 5: Layout of Haystack Store file

为了加快机器重启后内存中元信息的构建，物理机会将内存中的这些元信息定期做 snapshot 即 index file；其顺序和 store file 保持一致。



目录：

提供logical volume 到physical volume 的映射

以及文件到logical volume的映射

缓存：

可以理解为一个分布式的HashTable,充当内部的CDN.

Key为小文件id,Value为小文件的metadata。

如果Cache未命中，才访问store去寻找metadata。

读写删小文件流程：

- 读

1.来自Cache的读请求包括：(volume id, key, cookie)

2.store物理机会在内存中查找小文件相关的元信息(文件描述符，needle的offset和size)

3.根据offset和size从volume快速定位小文件在磁盘的位置，读出该文件的数据与元信息

4.根据needle的checksum进行校验，随后返回数据

- 写

1.写请求包括:(volume id, key, cookie)

2.每个物理机会把这些信息同步追加到对应的volume中（新增加一个needle）

3.对于update请求也是简单的追加，但是这样会造成重复的key。默认取offset最大的为版本最新。并且定期compact volume，删除过期的needle

- 删

1.同步设置内存和 volume中对应图片元信息的 flag

2.如果请求到某个被删的 Photo，在内存中发现其 flag 被设置了，就报一个异常。被置为已删除的图片所占的空间会暂时不可用

3.在定期的compaction时，会回收这部分空间

总结

HayStack通过引入Volume这一超大文件来同时存储若干个小文件的metadata和实际数据来减少metadata的数目和所占容量，使得（文件名，文件metadata）的映射关系可以存储到内存中。并且从内存中读取metadata后可以直接定位到文件的位置，节省了大量的disk io。

当文件的数目多到连内存也不能完全存储所有文件的metadata的映射关系的时候，这时候可以把映射关系存储到KV数据库中。同时可以通过分布式数据库，通过数据库的**分片（sharding）**来进一步提高metadata的存储能力和进行读取的分流。减轻单台物理机的io和cpu压力。接下来第二部分会介绍一个基于Haystack的开源实现——seaweadfs和我准备选择的几种用于存储metadata的数据库的技术简介。

第二部分 开源实现 & 毕业设计方案

SeaweedFS

https://github.com/chrislusf/seaweedfs

这是一个开源的，基于上述haystack设计的，适用于海量小文件存储的分布式文件系统。

架构简单分为：

- master server

负责接收客户端的请求，维护与多个volume server的通信。

- volume server

存储volume的服务器，若干个文件保存成一个超大的文件——volume，请求从数据库获得文件的metadata后就可以根据volume id和小文件的大小及在volume的偏移量直接从volume中把文件读取出来。

- filer store

存储metadata的数据库服务器，在读取某个文件前，先要访问数据库获取其metadata。**metadata记录着该文件所在的目录等元数据**，可以理解为haystack论文中的Directory

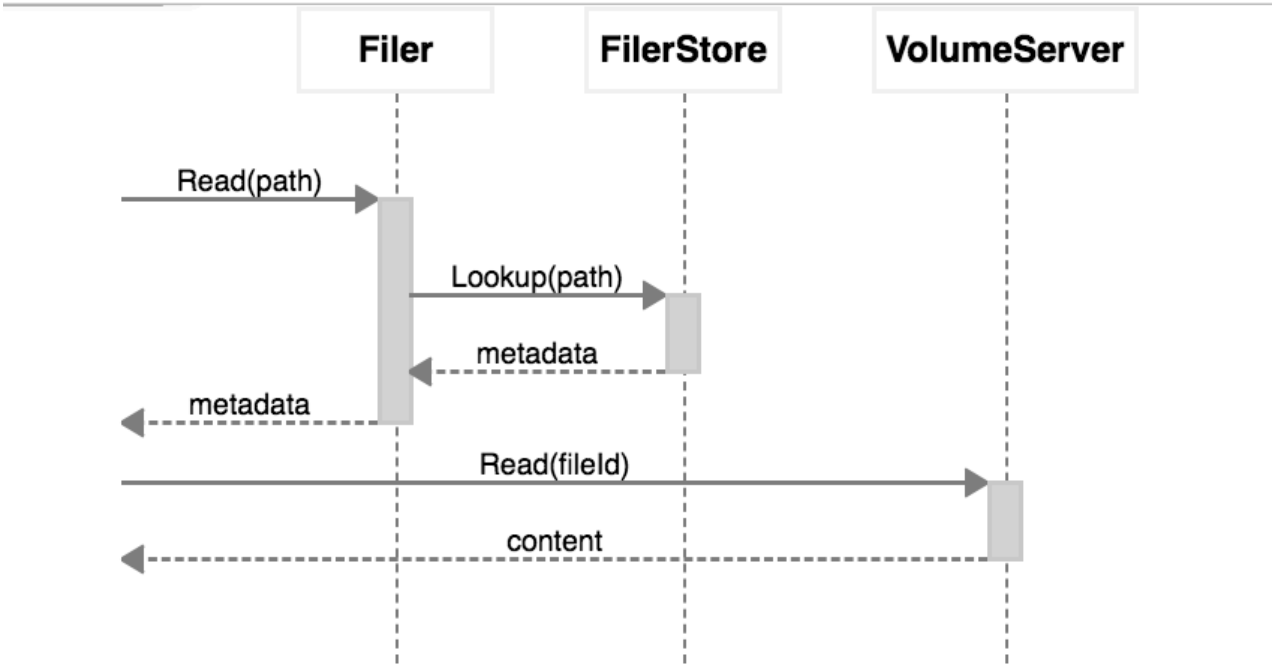
metadata默认都存储在leveldb中，也可以选择保存在其他KV数据库中如:redis,cassandra,elastic search等

目前可以通过开发来添加对其他数据库的支持，目前我在原项目基础上已经添加了对HBase和ClickHouse的支持，后续会尝试使用tikv或者badgerdb（golang版rocksdb），并且尝试针对不同的数据库后端分别进行性能基准测试，然后分析比较。

以文件读取为例：

1.首先从数据库（HBase/LevelDB/etcd/TiKV）等读取metadata

2.根据metadata从volumeServer读取文件的真实数据



以下表格是目前开源支持的用于存储metadata的后端数据库以及横向比较。

Filer Store Name	Lookup	number of entries in a folder	Scalability	Directory Renaming	TTL	Note
memory	O(1)	limited by memory	Local, Fast		Yes	for testing only, no persistent storage
leveldb	O(logN)	unlimited	Local, Very Fast		Yes	Default, fairly scalable
leveldb2	O(logN)	unlimited	Local, Very Fast, faster than leveldb		Yes	Similar to leveldb, part of the lookup key is 128bit MD5 instead of the long full file path
Mongodb	O(logN)	unlimited	Local or Distributed, Fast		Yes	Easy to manage
Redis	O(1)	limited	Local or Distributed, Fastest		Yes	one directory's sub file names are stored in one key~value entry
Cassandra	O(logN)	unlimited	Local or Distributed, Very Fast		Yes	
MySql	O(logN)	unlimited	Local or Distributed, Fast	Atomic	Yes	Easy to manage
Postgres	O(logN)	unlimited	Local or Distributed, Fast	Atomic	Yes	Easy to manage
MemSql	O(logN)	unlimited	Distributed, Fast	Atomic	Yes	Scalable
TiDB	O(logN)	unlimited	Distributed, Fast	Atomic	Yes	Scalable
CockroachDB	O(logN)	unlimited	Distributed, Fast	Atomic	Yes	Scalable
Etcd	O(logN)	~10GB	Distributed, 10,000 writes/sec		Yes	No SPOF. High Availability.
ElasticSearch	O(logN)	unlimited	Distributed, Fast	Atomic	Yes	Scalable, Searchable

同时支持把该系统挂载为FUSE（用户空间文件系统),**允许用户在本地以POSIX语义操作文件，包括增删查改等，并且文件可以以目录格式组织**

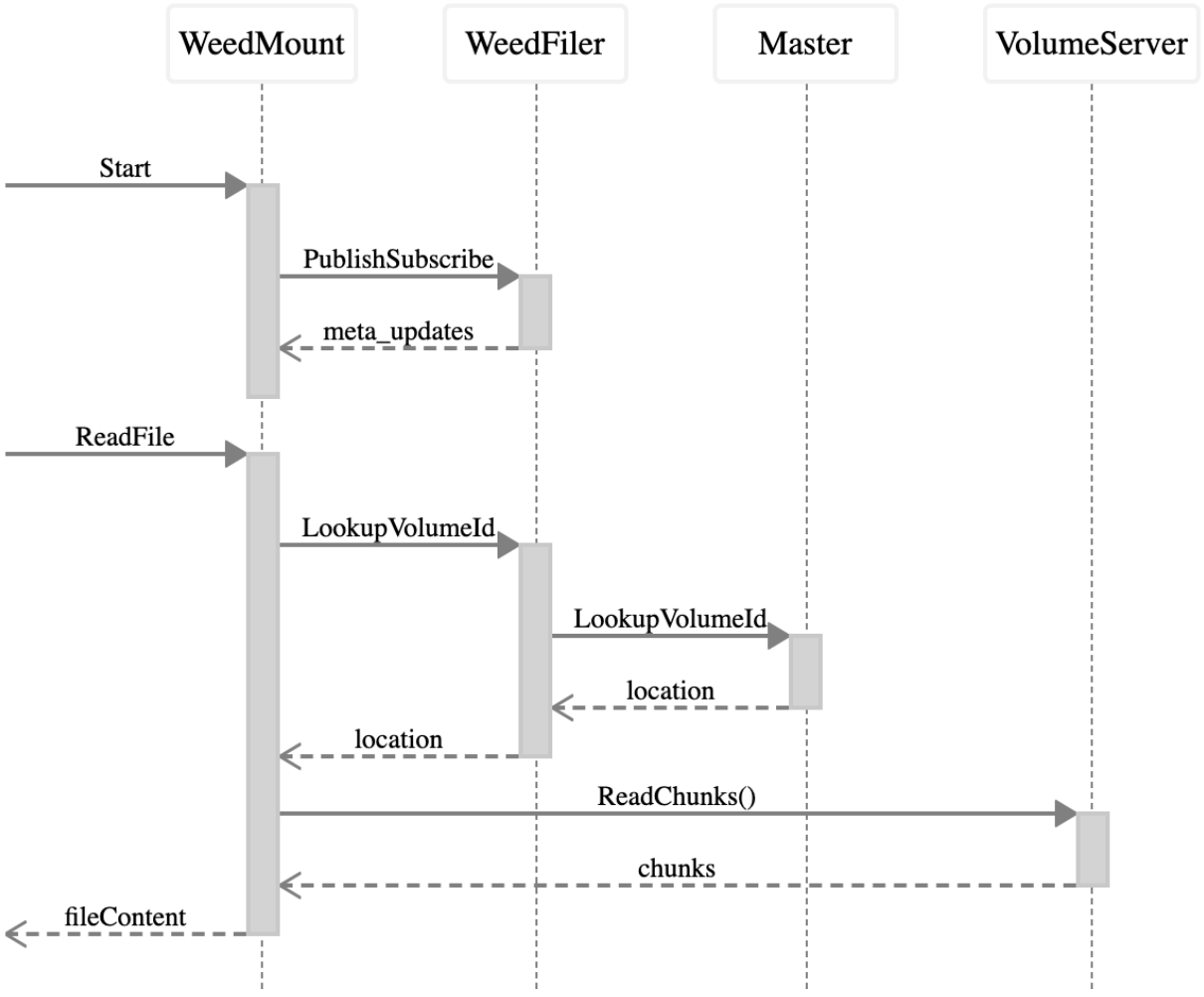
挂载成fuse后读写流程如下所示：

读：

1.挂载层会把请求转发给Filer层，Filer层负责从数据库后端中把文件的metadata捞出来

2.然后挂载层根据metadata得知文件所对应的volume-id, 偏移量和大小，然后向master服务器查询对应的volume上的数据

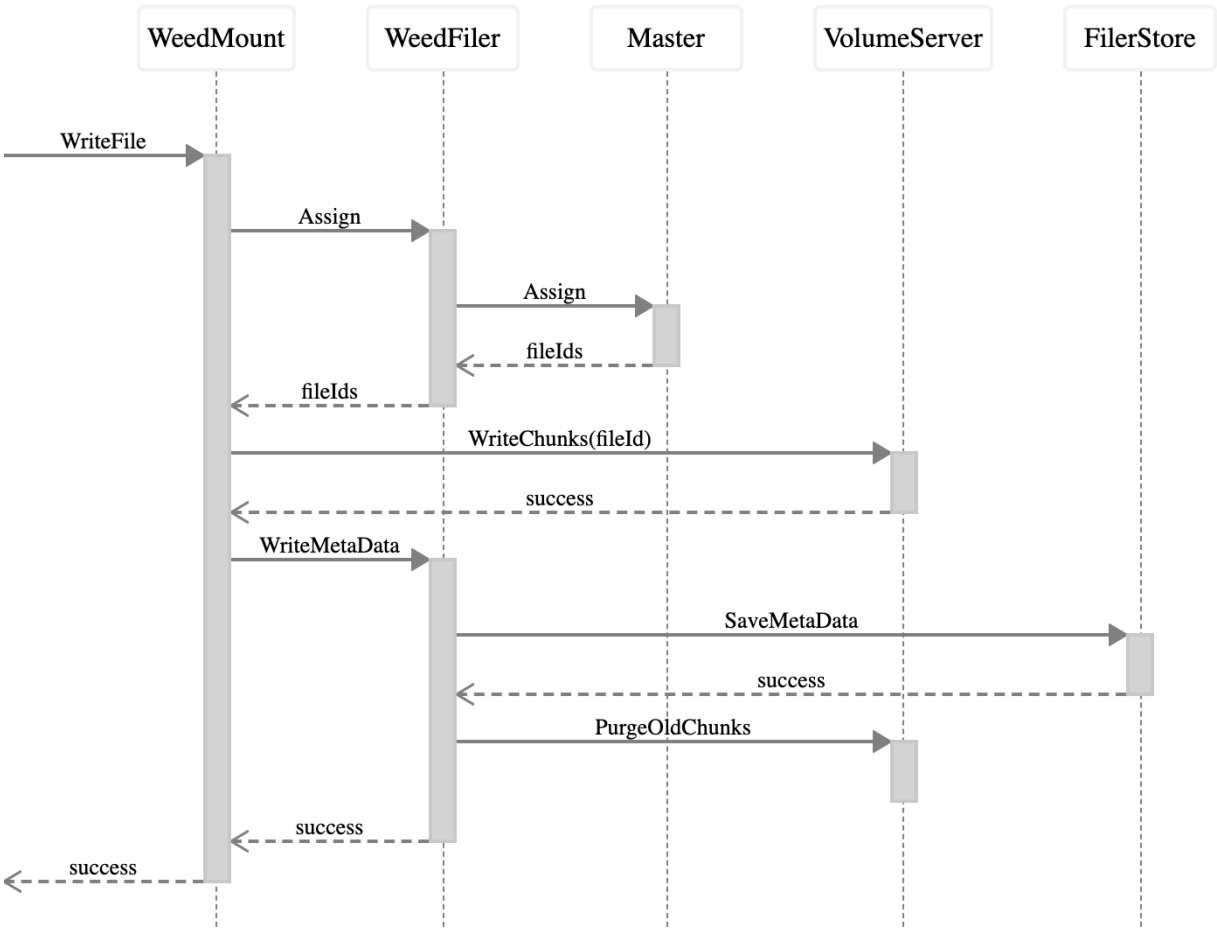
3.volume server把查询得到的数据返回



写：

1.先写文件数据到volume server

2.再在数据库后端更新metadata



毕业设计设计方案

大体参考Haystack的论文和seaweedfs的设计，抽取出seaweedfs最核心的功能 d.md 4个模块自己来实现一个海量小文件存储系统的demo：

1.Master

负责接收客户端请求与维护volume的关系

2.Volume server

负责存储小文件的真实数据。

主要任务要设计needle和volume的具体数据结构，由于是毕设，可以尽可能简化功能设计。

3.Filer(metadata的数据库存储层)

支持多种数据库存储后端，目前我打算尝试的有:Hbase, ClickHouse,TiKV,DagerDB,Redis五种。

数据库用于维护这样的映射关系(文件名:压缩编码后的metadata)

那么系统不需要

4.FUSE

负责把文件系统挂载成用户态文件系统，使得可以像本地文件系统一样操作文件

具体文件的读写流程和volume，needle的数据结构都在前面介绍过了，这里不赘述。

最后的预期成果是一个“SeaWeed FS”的简化版。提供挂载到系统FUSE的功能，实现基本的增删查改功能，并且能够可选地支持若干个数据库后端用于存储metadata。

可能优化与探索：

1.比较多种数据库在不同场景下存储metadata的性价比优劣。

如在文件数量不同，内存容量不同的情况下，设计benchmark来测试海量小文件的增删查改性能与内存/磁盘开销等。并且针对结果比较作出原因分析与探讨。并且根据实验结果给出在生产环境中数据库选择的一些考虑和建议。

2.由于我们的场景是读远大于写，因此一个容易想到的优化就是加一层Cache(例如如果用的是Hbase,可以在filer层前加一层memcached或者redis来挡掉大部分的读请求，进一步减少操作Hbase的Disk IO)

数据库选型

这一部分简单介绍一下我所选择的几种数据库，这几种数据库产品要么都是知名度比较高的：已经在工业界有了非常成熟应用的产品如Hbase,Redis。也有最近在开源届上升势头较猛的badgerDB。这几种都属于分布式KV数据库。除此之外，我还选择了一种近年来在各大数据库排行榜异军突起的关系型列式数据库：ClickHouse，之所以选择ClickHouse是因为这是目前**开源最快**的适用于海量数据分析的列式数据库，核心是极致地压榨cpu性能和追求极致的数据压缩率。

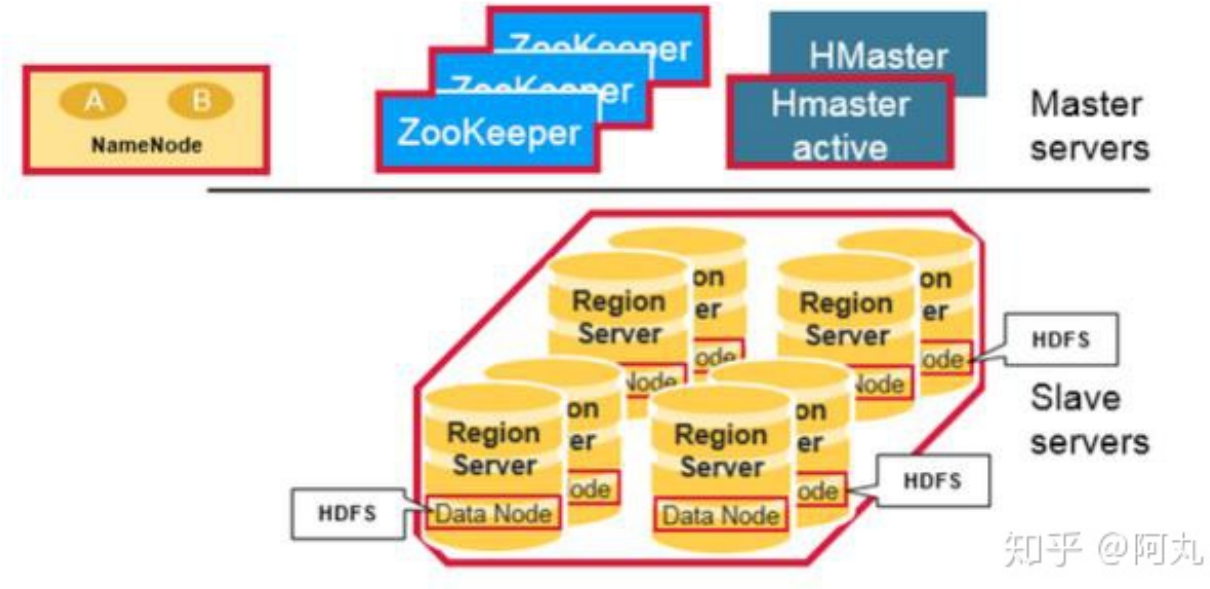
下面我来简单介绍一下这几种数据库的架构：

Hbase

Hbase源于BigTable,是一个**高可靠性、高性能、面向列、可伸缩、 实时读写的分布式数据库**

hbase架构：

- 包含了三种类型server,zookeeper,HMaster和region server

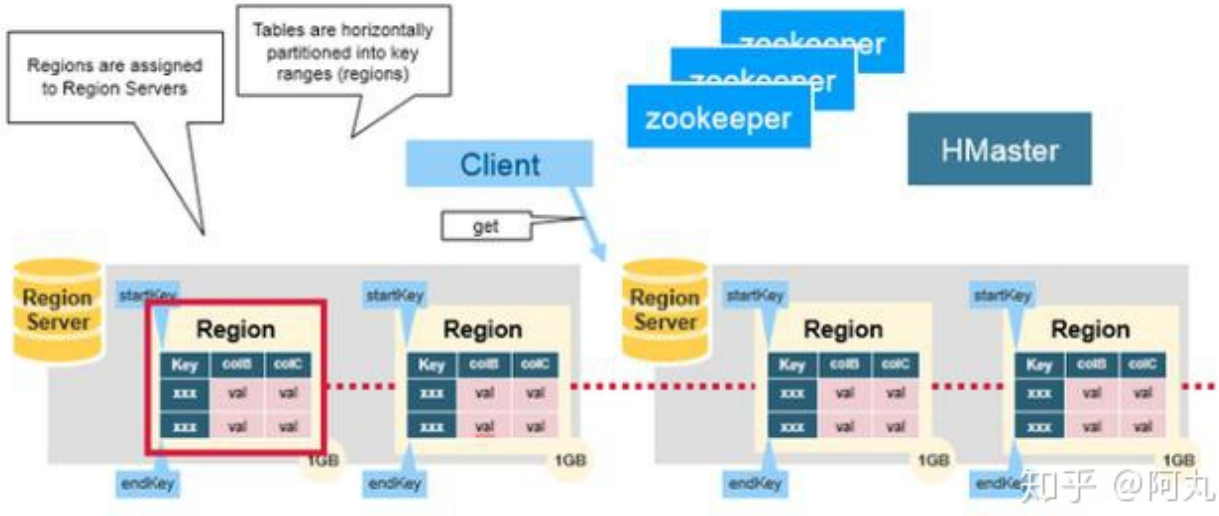


- Region server主要用来服务读和写操作。当用户通过client访问数据时，client会和HBase RegionServer 进行直接通信。
- HMaster主要进行region server的管理、DDL（创建、删除表）操作等。
- Zookeeper是HDFS（Hadoop Distributed File System）的一部分，主要用来维持整个集群的存活，保障了HA，故障自动转移。

而底层的存储，还是依赖于HDFS的。

- Hadoop的DataNode存储了Region Server所管理的数据，所有HBase的数据都是存在HDFS中的。
- Hadoop的NameNode维护了所有物理数据块的metadata。

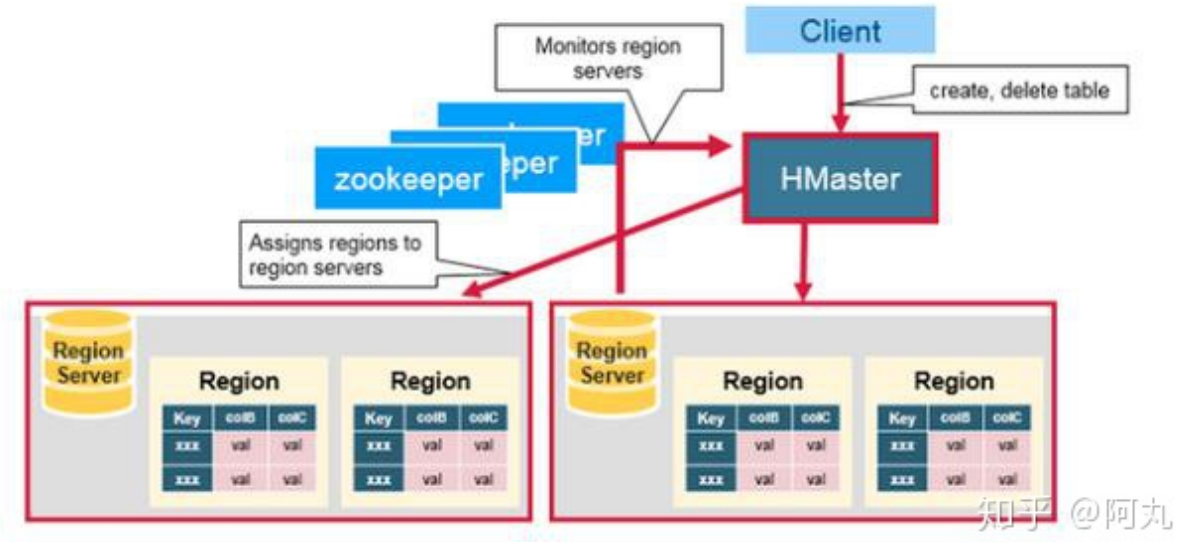
Region Server



HBase 的tables根据rowkey的范围进行水平切分，切分后分配到各个regions。一个region包含一个表在start key和end key所有行。region会被分配到集群中的各个region server，而用户都是跟region server进行读写交互。一个region一般建议大小在5-10G。

HMaster

- 与region server的交互，对region server进行统一管理：
- 启动时region的分配 崩溃后恢复的region重新分配 负载均衡的region重新分配
- Admin相关功能：
- 创建、删除、更新表结构等DDL操作

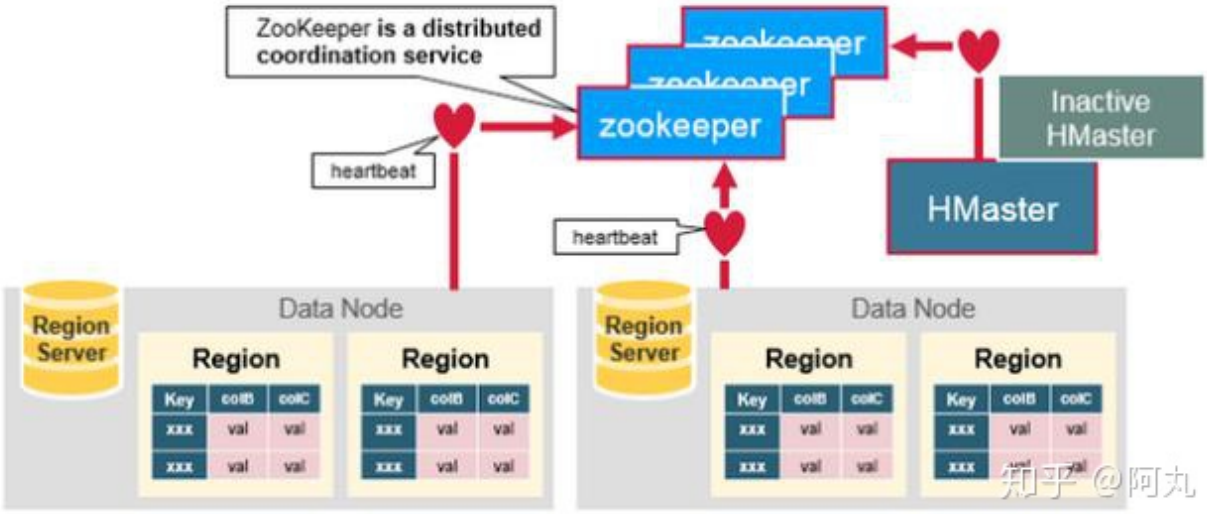


Zookeeper

HBase使用Zookeeper作为分布式协调服务，来维护集群内的server状态。

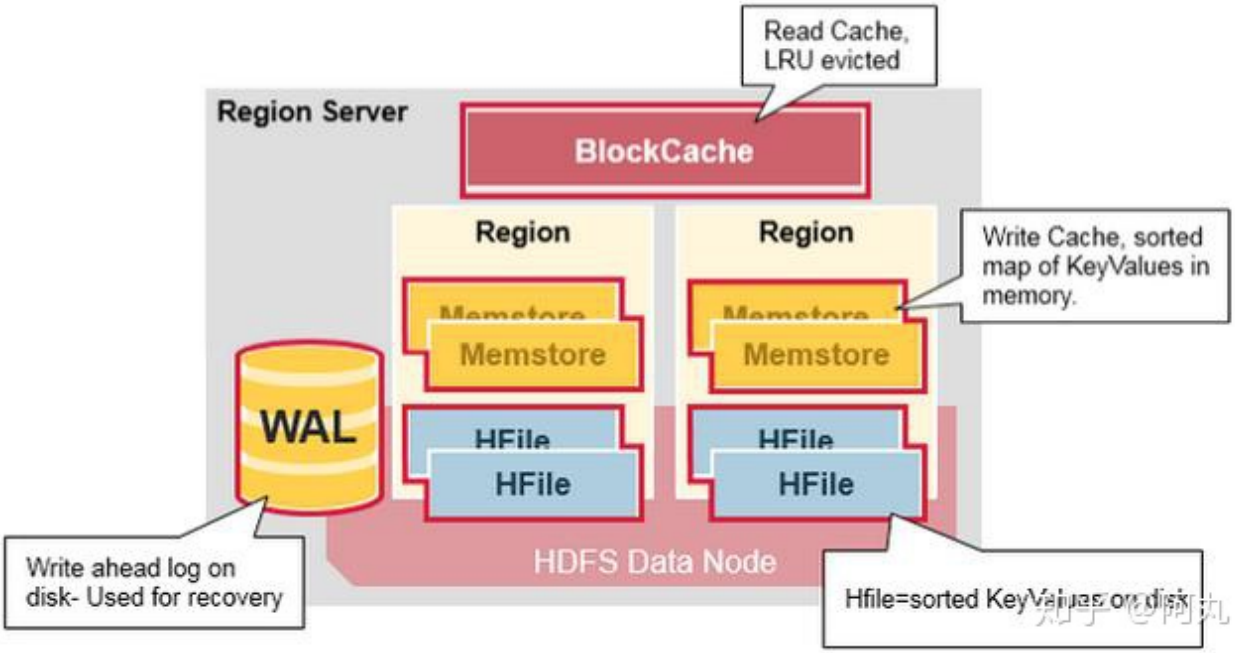
Zookeeper通过 heartbeat 维护了哪些server是存活并可用的，并提供server的故障通知。同时，使用一致性协议来保证各个分布式节点的一致性。

这里，需要特别关注，zookeeper负责来HMaster的选举工作，如果一个HMater节点宕机了，就会选择另一个HMaster节点进入active状态。



具体每个region server会运行在HDFS的data node上。

读写过程都是分布式的



Hbase的数据模型

Row Key	TimeStamp	CF1	CF2	CF3
“RK000001”	T1		CF2:q1=val3	CF3:q2=val2
	T2	CF1:q3=val4		
	T3		CF2:q4=val5	

这是Hbase中一行数据的数据模型，包含了三个Cell,每个Cell是由行和列坐标来共同确定的，并且每个Cell都有对应的时间戳版本。

以我们文件系统为例，可以这样来建模：

Row Key为文件的绝对路径名或者id

列族只有一个，命名为cf

然后每一个row key对应的列族的单元格存储的就是序列化后**metadata**字节序列

那么在按文件名来读取文件的时候，就可以直接从HBase中读取出该文件对应的**metadata**,然后反序列化出来后得到具体信息，最后从Volume Server把文件数据读取出来。

Redis

Redis是一个key-value存储系统。和Memcached类似，它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash(哈希类型)。

与Memcached一样，为了保证效率，数据都是缓存在内存中。区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件。

Redis与Memcached区别在于,Redis用来作为缓存，也可以做存储。而Memcached只能缓存数据。

以我们文件系统为例，可以这样来建模：

(Key: Value) -> (文件绝对路径名: 序列化后的metadata数据)

但由于是内存数据库，所以单台机器存储的数据量是有限的，意味着我们的单台机器所能存储的文件metadata是有限的，这意味着我们的能创建的目录/文件数量是有限的 所以我的想法是Redis主要还是用作其他后端数据库的缓存层。

ClickHouse

ClickHouse是面向联机分析处理的列式数据库，支持SQL查询，且查询性能好，特别是基于大宽表的聚合分析查询性能非常优异，比其他分析型数据库速度快一个数量级。

主要特性包括：

- 数据压缩比高。
- 多核并行计算。
- 向量化计算引擎。
- 支持嵌套数据结构。
- 支持稀疏索引。

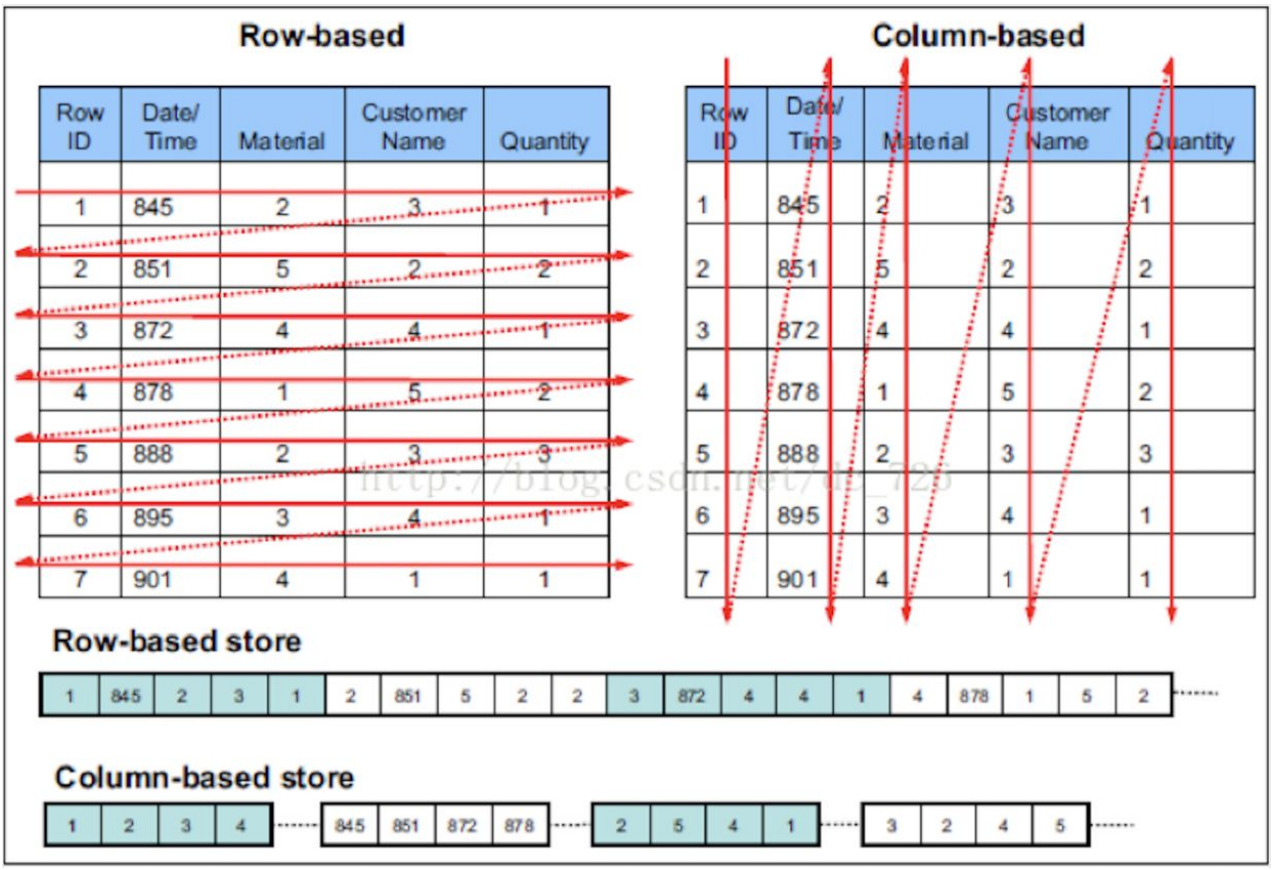
为什么我会选择一款OLAP场景下的数据库，是因为其实我们的文件访问场景：

读远多于写，数据不更新或者少更新，读大量行但是少量列，结果集较小

就是一个典型的OLAP数据分析场景。

然后下面简单介绍一下ClickHouse的列式存储以及架构

列式存储



1） 分析场景中往往需要读大量行但是少数几个列。在行存模式下，数据按行连续存储，所有列的数据都存储在一个block中，不参与计算的列在IO时也要全部读出，读取操作被严重放大。而列存模式下，只需要读取参与计算的列即可，极大的减低了IO cost，加速了查询。

2）同一列中的数据属于同一类型，压缩效果显著。列存往往有着高达十倍甚至更高的压缩比，节省了大量的存储空间，降低了存储成本。

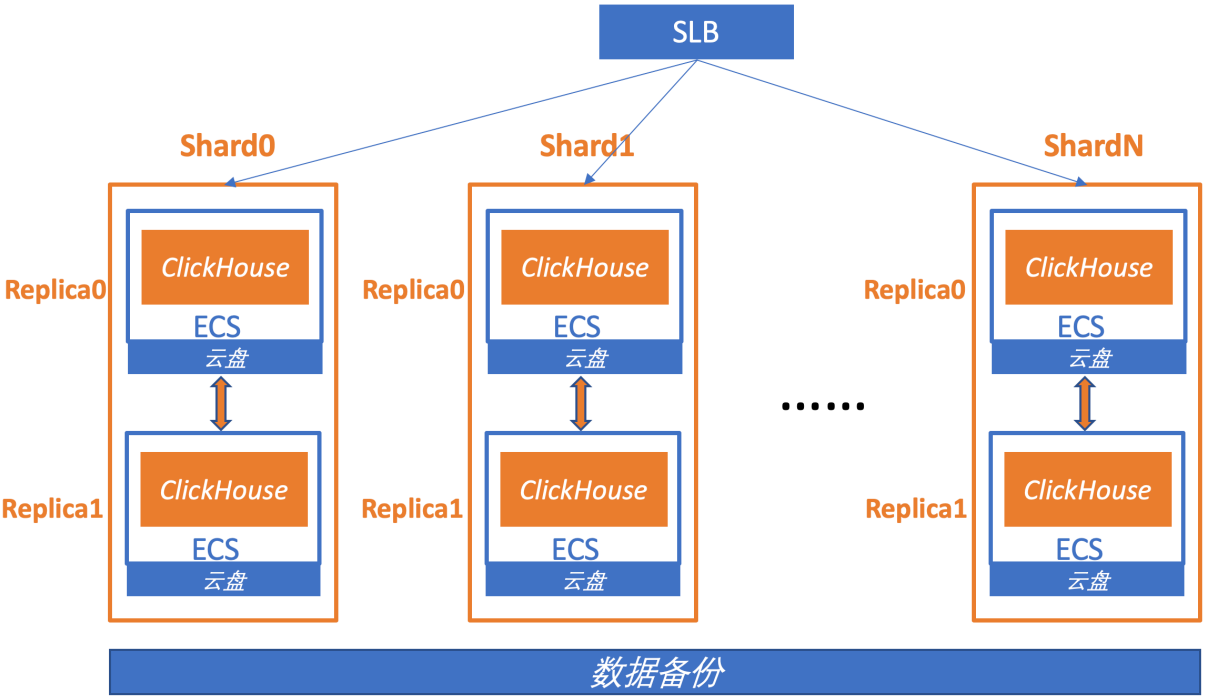
3）更高的压缩比意味着更小的data size，从磁盘中读取相应数据耗时更短。

4）自由的压缩算法选择。不同列的数据具有不同的数据类型，适用的压缩算法也就不尽相同。可以针对不同列类型，选择最合适的压缩算法。

5）高压缩比，意味着同等大小的内存能够存放更多数据，系统cache效果更好。

架构

ClickHouse支持单机模式，也支持分布式集群模式。在分布式模式下，ClickHouse会将数据分为多个分片（shard），并且分布到不同节点上。每个分片又可以设置多个副本来保持高可用。数据分片，让ClickHouse可以充分利用整个集群的大规模并行计算能力，快速返回查询结果。



总结

ClickHouse凭借其列式存储来极大减少Disk IO的开销以及通过分片机制来实现线性拓展，提供海量数据的处理能力。从处理场景来看是非常适合我们的读远大于写，文件修改少的场景的。

BadgerDB

用go语言编写的Key-Value数据库，针对SSD做大量优化。在性能评测上随机读的性能是RocksDB的3.5倍以上。

BadgerDB的key value是分开存储的，并且主要场景在value远比key大的情况：

这样做的好处是key和value不用同时保存在LSM Tree上。我们知道LSM Tree的compaction操作（即把多个sst文件读取到内存中进行归并排序然后重新生成一个大的sst文件）涉及到大量的disk io，写放大会比较严重。

那么如果key和value数据同时保存在一起，而且value数据比较大（例如key只有16B, Value有16kB）那么其实compcation要操作的文件会很大,开销是非常大的。而BadgerDB比较聪明的就是key和value分开存储,key存储在LSM Tree, 而value 存储在一个叫value log 的 WAL文件,直接利用SSD的随机读把文件读取出来。

在LSM Tree中紧随着key保存的是一个指向value的指针，换而言之，我们构造出来的LSM Tree所占空间其实会非常小，意味着LSM Tree的Level会更少，相应的读写放大都会减少。同时LSM Tree还能保存在内存中，极大地加快了key的遍历速度。

总结

BadgerDB是一个用go语言编写的对标RocksDB的Key-Value数据库，核心是通过key和value数据的分开存储来压缩存储引擎LSM Tree的所占空间，使得其能够保存在内存中，以加快Key的遍历速度。

ref

https://blog.gopheracademy.com/advent-2014/fuse-zipfs/