hdfs的基本原理

**hdfs写流程**

客户端要向HDFS写数据，首先要跟namenode通信以确认可以写文件并获得接收文件block的datanode，然后，客户端按顺序将文件逐个block传递给相应datanode，并由接收到block的datanode负责向其他datanode复制block的副本



**写详细步骤：**

1、根namenode通信请求上传文件，namenode检查目标文件是否已存在，父目录是否存在

2、namenode返回是否可以上传

3、client会先对文件进行切分，比如一个blok块128m，文件有300m就会被切分成3个块，一个128M、一个128M、一个44M请求第一个 block该传输到哪些datanode服务器上

4、namenode返回datanode的服务器

5、client请求一台datanode上传数据（本质上是一个RPC调用，建立pipeline），第一个datanode收到请求会继续调用第二个datanode，然后第二个调用第三个datanode，将整个pipeline建立完成，逐级返回客户端

6、client开始往A上传第一个block（先从磁盘读取数据放到一个本地内存缓存），以packet为单位（一个packet为64kb），当然在写入的时候datanode会进行数据校验，它并不是通过一个packet进行一次校验而是以chunk为单位进行校验（512byte），第一台datanode收到一个packet就会传给第二台，第二台传给第三台；第一台每传一个packet会放入一个应答队列等待应答

7、当一个block传输完成之后，client再次请求namenode上传第二个block的服务器。

**hdfs读流程**

客户端将要读取的文件路径发送给namenode，namenode获取文件的元信息（主要是block的存放位置信息）返回给客户端，客户端根据返回的信息找到相应datanode逐个获取文件的block并在客户端本地进行数据追加合并从而获得整个文件



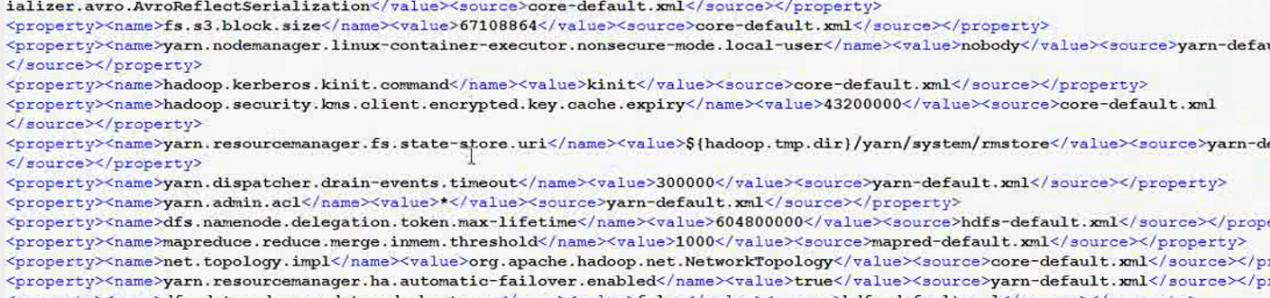
1、跟namenode通信查询元数据（block所在的datanode节点），找到文件块所在的datanode服务器

2、挑选一台datanode（就近原则，然后随机）服务器，请求建立socket流

3、datanode开始发送数据（从磁盘里面读取数据放入流，以packet为单位来做校验）

4、客户端以packet为单位接收，先在本地缓存，然后写入目标文件，后面的block块就相当于是append到前面的block块最后合成最终需要的文件。

hadoop提交作业的总体流程



job端提交流程：

1.客户端向resouce manager申请提交一个application

2.resource manager返回application提交的路径：hdfs://xxx../staging以及application\_id

3.客户端提交job运行所需要的资源

4.客户端资源提交完毕，申请运行MRAPPMaster

5.resource manager将用户请求转成task

6.其中一个node manager领取task任务

7.创建container容器

8.向resource manager申请运行map task的容器

9.剩余的node manager领取task任务，创建容器

10.发送程序运行脚本

11.想resource manager申请运行reduce task程序的容器

12.reduce task向map端获取相应分区的数据

hadoop的shuffle过程

MapReduce运行过程

mapreduce运行过程：

map端：

1.读取数据文件，将文件转成K，V对，对每一个K，V对调用一个map函数

2.收集器收集K，V对到环形缓冲区中，当数据量达到环形缓冲区大小的80%时，就会溢出成文件

3.在溢出之前，会按照key值的hashcode%分区数进行分区，然后对分区内的数据按照key进行排序

4.当磁盘的文件达到阈值后，会对文件进行排序合并成大文件

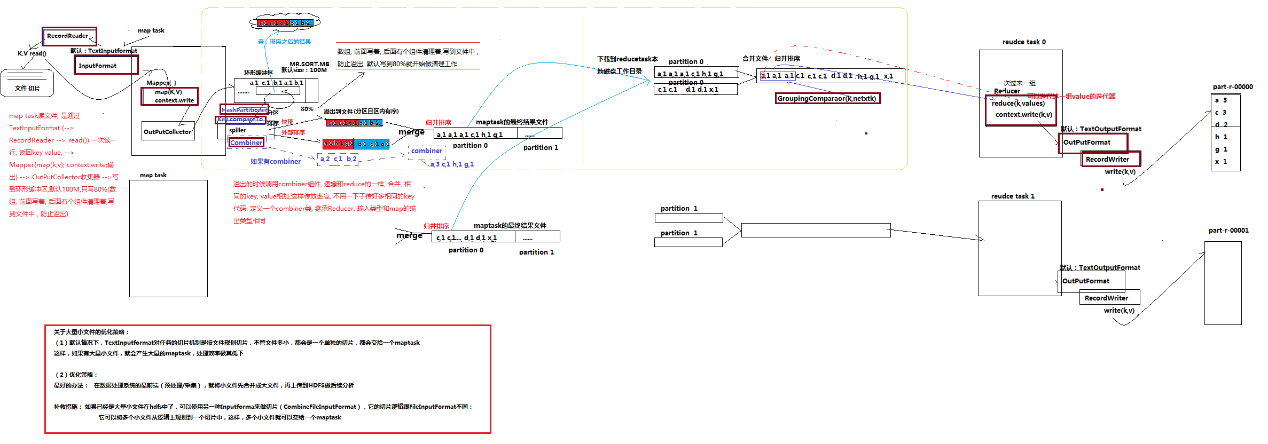
5.这样map端的shuffle过程就完毕了，数据都有序的存在磁盘里，等待reducer来取

reduce端：

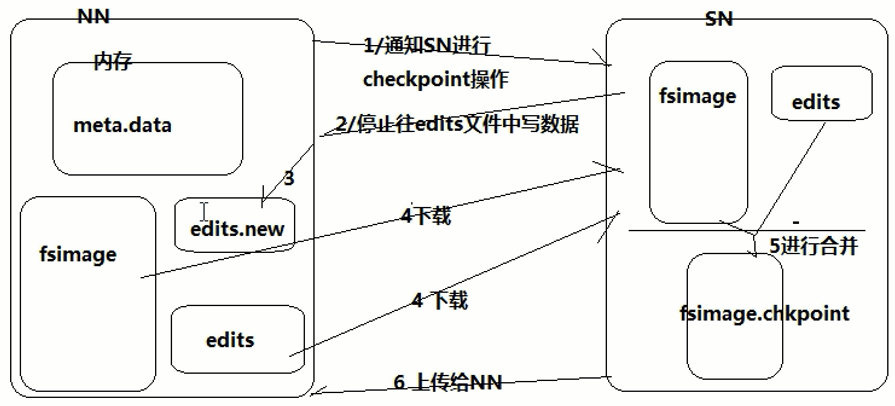
1.将map端的输出文件拷贝到reduce task的本地磁盘工作目录

2.然后进行合并排序，合并成大文件，reduce阶段的shuffle阶段完成了

3.reduce job开始，对key进行分组，对于每一组调用reduce输出到文件中



NameNode，SecondaryNameNode、DataNode的工作职责



SecondaryNameNode的工作流程：

1. secondary 通知namenode切换edits文件
2. secondary从namenode获得fsimage和edits
3. secondary将fsimage载入内存，然后开始合并edits
4. secondary将新的fsimage发回给namenode
5. namenode用新的fsimage替换旧的fsimage

文件系统的管理节点。它维护着整个文件系统的文件目录树，文件/目录的元信息和每个文件对应的数据块列表。接收用户/客户端的操作请求。

文件包括：   
fsimage:元数据镜像文件(保存文件系统的目录树)。并不是实时同步，而是存储某一时段NameNode内存元数据信息。hdfs-site.xml的dfs.name.dir属性。

edits:操作日志文件(针对目录树的修改操作),被写入共享存储系统中 ,比如NFS、JournalNode。

fstime:保存最近一次checkpoint的时间

**NameNode**

Namenode始终在内存中保存metedata，用于处理“读请求”

当有“写请求”到来时，namenode会首先写editlog到磁盘，即向edits文件中写日志，成功返回后，才会修改内存，并且向客户端返回。

Hadoop会维护一个fsimage文件，也就是namenode中metedata的镜像，但是fsimage不会随时与namenode内存中的metedata保持一致，而是每隔一段时间通过合并edits文件来更新内容。Secondary namenode就是用来合并fsimage和edits文件来更新NameNode的metedata的，用新的fsimage替换旧的。

**SecondaryNameNode**

从NameNode上下载元数据信息（fsimage,edits），然后把二者合并，生成新的fsimage，在本地保存，并将其推送到NameNode，替换旧的fsimage.

默认在安装在NameNode节点上，但为了安全性应该分开部署！

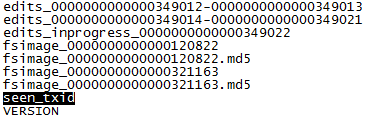
**DataNode**

提供真实文件数据的存储服务。

namenode硬盘损坏，怎样进行恢复数据？

将secondary的元数据目录拷贝给namenode

namenode的工作目录应该配在多块磁盘上



seen\_txid：文件中记录的是edits滚动的序号，每次重启namenode时，namenode就知道要将哪些edits进行加载到内存中。

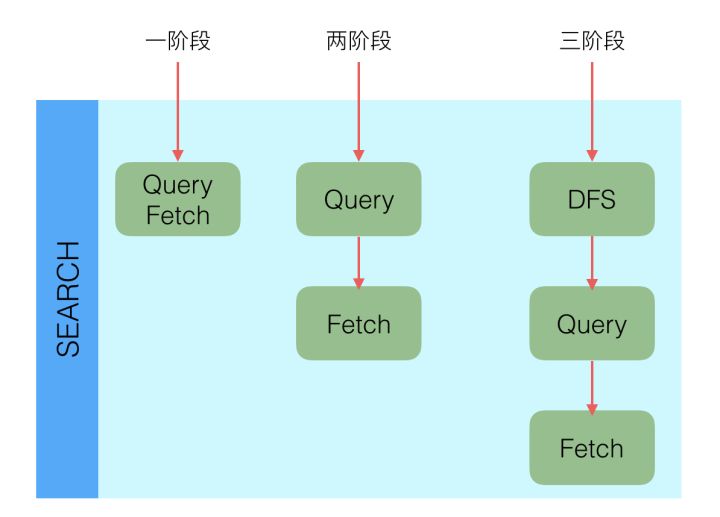
ElasticSearch读流程

1、Get请求：通过id查询指定的Doc

查询的时候先查询内存中的translog，如果找到就立即返回，如果没有找到再查询磁盘上的translog，如果还没找到则再去查询磁盘上的segment。这种查询是实时的。

2、Search请求：通过Query查询匹配Doc。

查询的时候是一起查询内存和磁盘上的segment，最后将结果合并后返回。这种查询是近实时的。



我们都知道es是一个分布式的存储和检索系统，在存储的时候默认是根据每条记录的\_id字段做路由分发的，这意味着es服务端是准确知道每个document分布在哪个shard上的。

相对比于CURD上操作，search一个比较复杂的执行模式，因为我们不知道哪些document会被匹配到，任何一个shard上都有可能，所以一个search请求必须查询一个索引或多个索引里面的所有shard才能完整的查询到我们想要的结果。

找到所有匹配的结果是查询的第一步，来自多个shard上的数据集在分页返回到客户端的之前会被合并到一个排序后的list列表，由于需要经过一步取top N的操作，所以search需要进过两个阶段才能完成，分别是query和fetch。

查询类型

1、query and fetch  
向索引的所有分片（shard）都发出查询请求，各分片返回的时候把元素文档（document）和计算后的排名信息一起返回。这种搜索方式是最快的。因为相比下面的几种搜索方式，这种查询方法只需要去shard查询一次。但是各个shard返回的结果的数量之和可能是用户要求的size的n倍。一般第二种适用于只需要查询一个Shard的请求。  
2、query then fetch（默认的搜索方式）  
如果你搜索时，没有指定搜索方式，就是使用的这种搜索方式。这种搜索方式，大概分两个步骤，第一步，先向所有的shard发出请求，各分片只返回排序和排名相关的信息（注意，不包括文档document)，然后按照各分片返回的分数进行重新排序和排名，取前size个文档。然后进行第二步，去相关的shard取document。这种方式返回的document与用户要求的size是相等的。

（一）query（查询阶段）

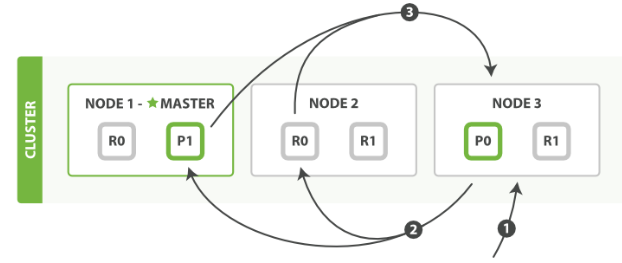
当一个search请求发出的时候，这个query会被广播到索引里面的每一个shard（主shard或副本shard），每个shard会在本地执行查询请求后会生成一个命中文档的优先级队列。

这个队列是一个排序好的top N数据的列表，它的size等于from+size的和，也就是说如果你的from是10，size是10，那么这个队列的size就是20，所以这也是为什么深度分页不能用from+size这种方式，因为from越大，性能就越低。

es里面分布式search的查询流程如下：

1，客户端发送一个search请求到Node 3上，然后Node 3会创建一个优先级队列它的大小=from+size

2，接着Node 3转发这个search请求到索引里面每一个主shard或者副本shard上，每个shard会在本地查询然后添加结果到本地的排序好的优先级队列里面。

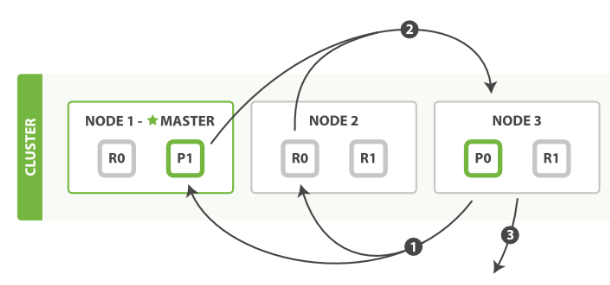
3，每个shard返回docId和所有参与排序字段的值例如\_score到优先级队列里面，然后再返回给coordinating节点也就是Node 3，然后Node 3负责将所有shard里面的数据给合并到一个全局的排序的列表。

上面提到一个术语叫coordinating node，这个节点是当search请求随机负载的发送到一个节点上，然后这个节点就会成为一个coordinating node，它的职责是广播search请求到所有相关的shard上，然后合并他们的响应结果到一个全局的排序列表中然后进行第二个fetch阶段，注意这个结果集仅仅包含docId和所有排序的字段值，search请求可以被主shard或者副本shard处理，这也是为什么我们说增加副本的个数就能增加搜索吞吐量的原因，coordinating节点将会通过round-robin的方式自动负载均衡。

（二）fetch（读取阶段）

query阶段标识了那些文档满足了该次的search请求，但是我们仍然需要检索回document整条数据，这个阶段称为fetch

流程如下：



1，coordinating 节点标识了那些document需要被拉取出来，并发送一个批量的mutil get请求到相关的shard上

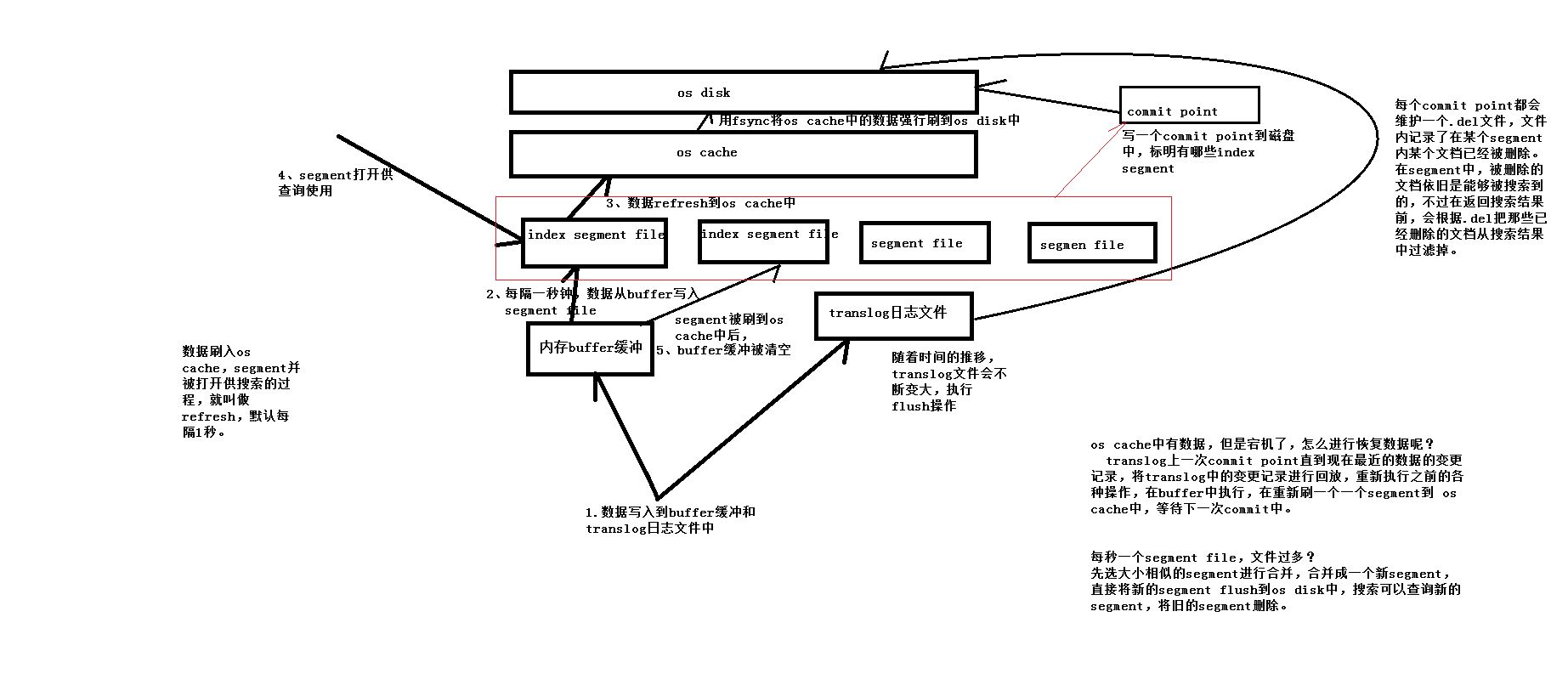
2，每个shard加载相关document，如果需要他们将会被返回到coordinating 节点上

3，一旦所有的document被拉取回来，coordinating节点将会返回结果集到客户端上。

这里需要注意，coordinating节点拉取的时候只拉取需要被拉取的数据，比如from=90，size=10，那么fetch只会读取需要被读取的10条数据，这10条数据可能在一个shard上，也可能在多个shard上所以 coordinating节点会构建一个multi-get请求并发送到每一个shard上，每个shard会根据需要从\_source字段里面获取数据，一旦所有的数据返回，coordinating节点会组装数据进入单个response里面然后将其返回给最终的client。

3、DFS query and fetch  
这种方式比第一种方式多了一个初始化散发(initial scatter)步骤，有这一步，据说可以更精确控制搜索打分和排名。  
4、DFS query then fetch  
比第2种方式多了一个初始化散发(initial scatter)步骤。

ElasticSearch写流程



在每一个Shard中，写入流程分为两部分，先写入Lucene，再写入TransLog。

1. 数据写入到buffer内存和translog日志中（防止断电数据丢失）
2. 每隔一秒钟，buffer中的数据被写入新的segment file，并进入os cache，此时segment被打开并供search使用。

（segment文件被写入cache后，这个segent就可以打开和查询，从而确保在短时间内就可以搜到，而不用执行一个full commit也就是fsync操作，这是一个非常轻量级的处理方式而且是可以高频次的被执行，而不会破坏es的性能。）

1. buffer被清空
2. 重复1~3，新的segment不断添加，buffer不断被清空，而translog中的数据不断累加。
3. 当translog长度达到一定程度的时候，commit操作发生，commit操作包括以下步骤：

（5-1）buffer中现存的所有数据写入一个新的segment，并写入os cache，打开Index segment供搜索使用，然后把buffer被清空

（5-2）把一个commit ponit写入磁盘，这个commit ponit 标明了所有的index segment

（5-3）filesystem cache中的所有index segment file缓存数据，被fsync强行刷到磁盘上

（5-4）现有的translog被清空，创建一个新的translog

fsync+清空translog，就是flush，默认每隔30分钟flush一次，或者当translog过大的时候，也会flush

translog，每隔5秒被fsync一次到磁盘上。在一次增删改操作之后，当fsync在primary shard和replica shard都成功之后，那次增删改操作才会成功

简化版：写入请求到达Shard后，先写Lucene文件，创建好索引，此时索引还在内存里面，接着去写TransLog，写完TransLog后，刷新TransLog数据到磁盘上，写磁盘成功后，请求返回给用户。

新建、索引和删除 请求都是 写 操作， 必须在主分片上面完成之后才能被复制到相关的副本

ElasticSearch数据恢复

基于translog和commit point，实现数据恢复

fsync操作加上清空translog，就是flush操作，默认每隔30分钟flush一次，或者当translog过大的时候，也会执行flush操作，也可以手动执行flush操作。但是建议不要手动执行flush操作，让es默认执行就行。 语法 ： POST /my\_index/\_flush

translog文件每隔5秒会被fsync一次到磁盘上进行保存。在一次增删改操作之后，当fsync在primary shard和replica shard都成功之后，那次增删改操作才会成功。

但是这种在一次增删改时强行fsync translog可能会导致部分操作比较耗时。并且因为是每隔5秒把translog文件保存到磁盘中，所以可能有5秒的translog文件丢失，进而导致5秒的es数据丢失，如果程序可以允许部分数据丢失，那么将es设置异步fsync translog会大大提升性能，

将es设置异步fsync translog的语法

PUT /my\_index/\_settings

{

"index.translog.durability": "async",

"index.translog.sync\_interval": "5s"

}

倒排索引

不可变的好处：

1. 不需要锁，提升并发能力，避免锁的问题
2. 数据不变，一直保存在os cache中，只要cache内存足够
3. Filter cache一直驻留在内存，因为数据不变
4. 可以压缩，节省cpu和io开销

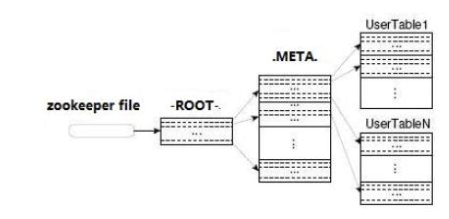
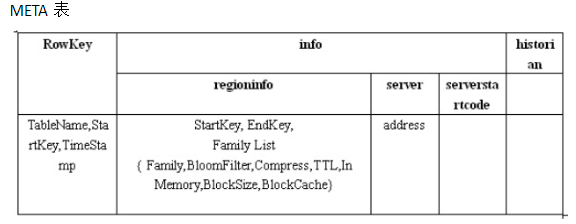
ES常见的优化：

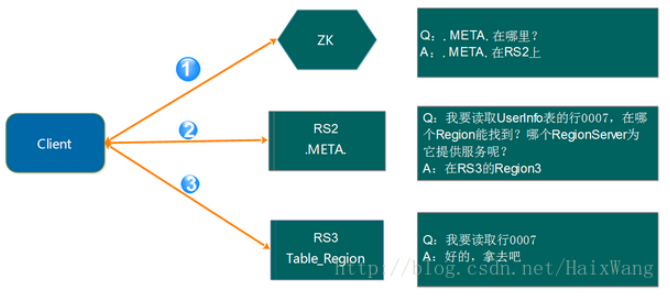
ES进程堆内存使用率高达100%，引发Full GC导致节点脱离集群，较高的内存会导致集群大部门时间都处于Old GC or Full GC，当GC时间达到一定阈值，节点由于无法响应心跳检测会被集群判定连接失败而脱离集群。

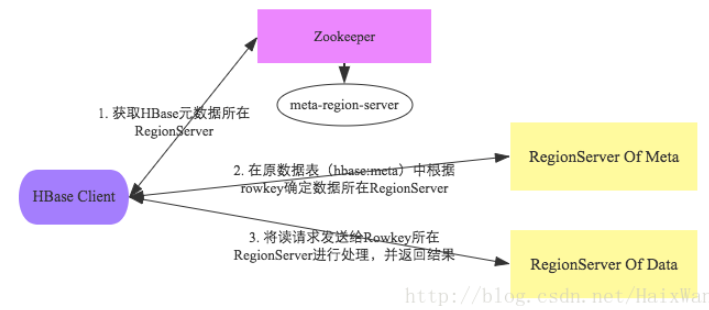
1. 对于string字段，关闭分词功能，并开启doc\_values功能
2. 人名最好不用分词，在mapping中给name字段设置为not\_analyzed， 但是这样不就检索不到人名相关的数据了
3. 强制限制每次请求的数据量（eg1000）
4. 拒绝使用分词字段进行排序、聚合
5. 避免海量数据聚合或者多层嵌套聚合查询
6. 无效索引、定期删除或者关闭
7. 合理进行容量规划，单分片不宜超过100G
8. 禁用文档的\_all字段，节省空间提升索引速度
9. 合理设置 index.refresh\_interval , 实时性和索引性能是互斥的
10. filter拿到相应的doc后不计算score不用排序
11. query会对符合条件的doc计算score并进行排序
12. filter的查询速度比query快很多

HBase读取数据流程

1. Client先访问zookeeper，从meta表读取region的位置，然后读取meta表中的数据。meta中又存储了用户表的region信息。
2. 根据namespace、表名和rowkey在meta表中找到对应的region信息
3. 找到这个region对应的regionserver
4. 查找对应的region
5. 先从MemStore找数据，如果没有，再到StoreFile上读(为了读取的效率)。
6. client访问zk，查找-ROOT-表，获取.META.表信息
7. 从.META.表查找，获取存放数据的region信息（找到region sever地址）
8. 最后通过RegionServer获取查找的数据。
9. 以后查询就直接在内存中查。





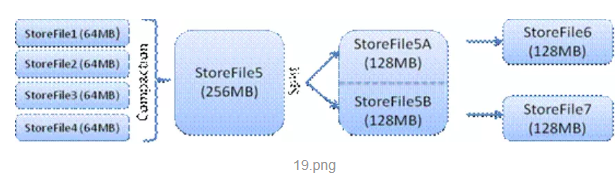


rowKey由三部分组成：TableName, StartKey 和 TimeStamp。

具体流程就是client--->zookeeper--->root---->meta--->region sever这个顺序

HBase写入数据流程

1. Client先访问zookeeper，从meta表获取相应region信息，然后找到meta表的数据
2. 根据namespace、表名和rowkey根据meta表的数据找到写入数据对应的region信息
3. 找到对应的regionserver
4. 把数据分别写到HLog和MemStore上一份
5. MemStore达到一个阈值后则把数据刷成一个StoreFile文件。（若MemStore中的数据有丢失，则可以总HLog上恢复）
6. 当多个StoreFile文件达到一定的大小后，会触发Compact 操作，合并为一个StoreFile，（这里同时进行版本的合并和数据删除。）
7. 当Storefile大小超过一定阈值后，会把当前的Region分割为两个（Split），并由Hmaster分配到相应的HRegionServer，实现负载均衡



Compaction主要有以下几个作用：

1. 合并文件
2. 清除删除、过期、多余版本的数据
3. 提高读写数据的效率

每个 RegionServer 包含多个 Region，每个 Region 包含多个Store，每个 Store 包含一个 MemStore 和多个 StoreFile。

在 Hbase 的表中，每个列族对应 Region 中的一个Store，Region的大小达到阈值时会分裂，因此如果表中有多个列族，则可能出现以下现象：

1）一个Region中有多个Store，如果每个CF的数据量分布不均匀时，比如CF1为100万，CF2为1万，则Region分裂时导致CF2在每个Region中的数据量太少，查询CF2时会横跨多个Region导致效率降低。

2）如果每个CF的数据分布均匀，比如CF1有50万，CF2有50万，CF3有50万，则Region分裂时导致每个CF在Region的数据量偏少，查询某个CF时会导致横跨多个Region的概率增大。

3）多个CF代表有多个Store，也就是说有多个MemStore，也就导致内存的消耗量增大，使用效率下降。

4）Region 中的 缓存刷新 和 压缩 是基本操作，即一个CF出现缓存刷新或压缩操作，其它CF也会同时做一样的操作，当列族太多时就会导致IO频繁的问题。

**Hbase中存储设计主要思想**

LSM树原理把一棵大树拆分成N棵小树，它首先写入内存中，随着小树越来越大，内存中的小树会flush到磁盘中，磁盘中的树定期可以做merge操作，合并成一棵大树，以优化读性能。

以上这些大概就是HBase存储的设计主要思想，这里分别对应说明下：

* 因为小树先写到内存中，为了防止内存数据丢失，写内存的同时需要暂时持久化到磁盘，对应了HBase的MemStore和HLog
* MemStore上的树达到一定大小之后，需要flush到HRegion磁盘中（一般是Hadoop DataNode），这样MemStore就变成了DataNode上的磁盘文件StoreFile，定期HRegionServer对DataNode的数据做merge操作，彻底删除无效空间，多棵小树在这个时机合并成大树，来增强读性能。