1 bigtable数据的存储方式，应该注意些什么：

bigtable是稀疏的，分布式的，持久化的，多维的排序映射。

行键 列键 时间戳 **(row:string, column:string, time:int64)→string**来表示一条键值对记录。

Bigtable的表会根据行键自动划分为片（tablet），片是负载均衡的单元。

行是表的第一级索引，

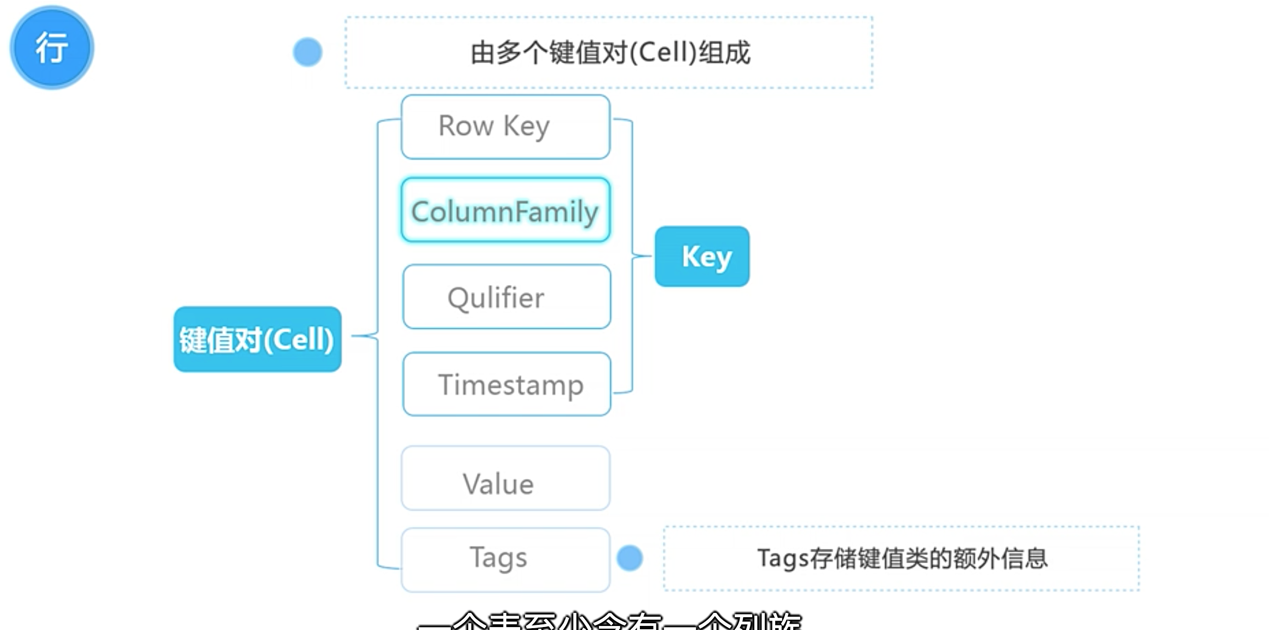
列是第二级索引为了方便管理，列被分为列族，一个列族里的列一般存储相同类型的数据。一行的列族很少变化，但是列族里的列可以随意添加删除。

时间戳是第三级索引。Bigtable允许保存数据的多个版本，版本区分的依据就是时间戳。

Bigtable存储稀疏数据方便，键值对的存储方式，没有值的数据不会被存储。

键-》行键 列族 列 时间戳

tag是额外信息，如访问控制。

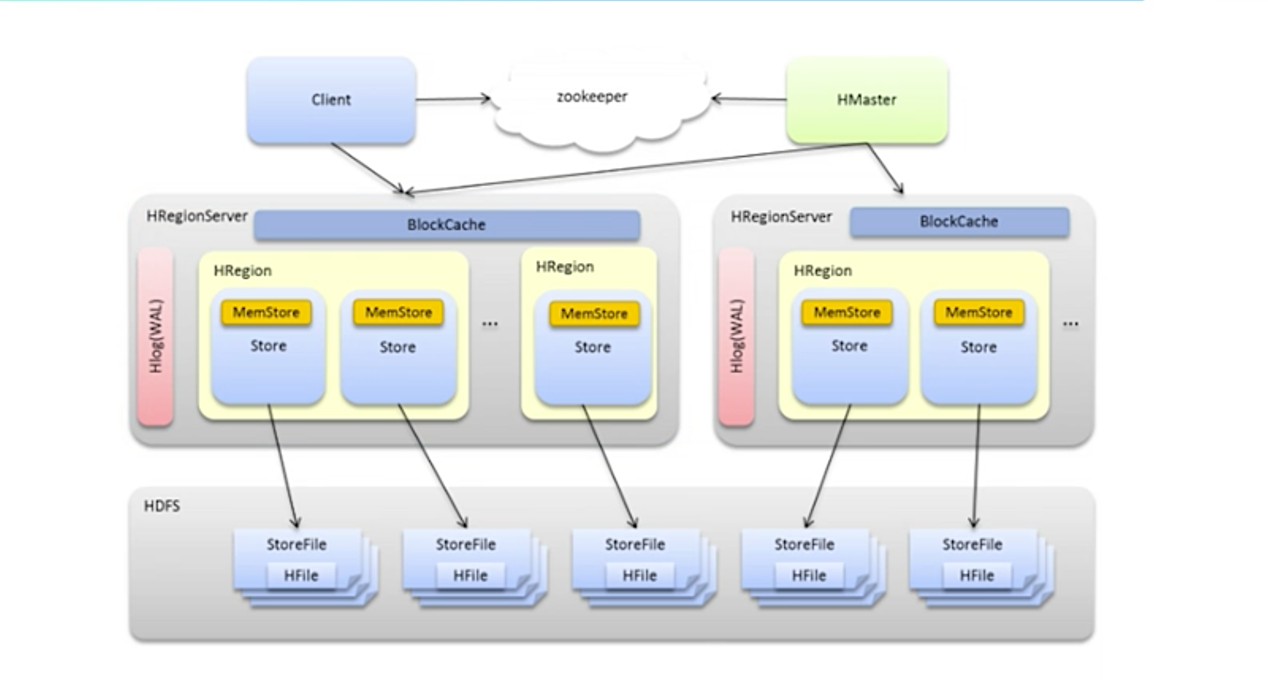


注意点：

1一个表至少含有一个列族，必须在表创建时指定。之后可以修改。

2 同一个列族下可以自定义备份数量和cache大小，将具有相同访问方式或者相同数据类型的数据放在同一个列族中。

3 列族不能太多



HRegionServer的

主要组成部分是Region

HRegionServer

还包括了HLog和BlockCache

一个Region由多个Store组成

每一个列族对应一个Store

HBase的的文件又称为store file

它是HFile的格式

被直接存放在HDFS中

HLog又称为write-ahead log

也就是WAL

Hbase写入

因为HBase新写入的数据

会被保存在内存中

也就是这个memstore中

如果集群发生故障数据就会丢失

所以为了防止数据丢失

所有被写入memstore中的数据

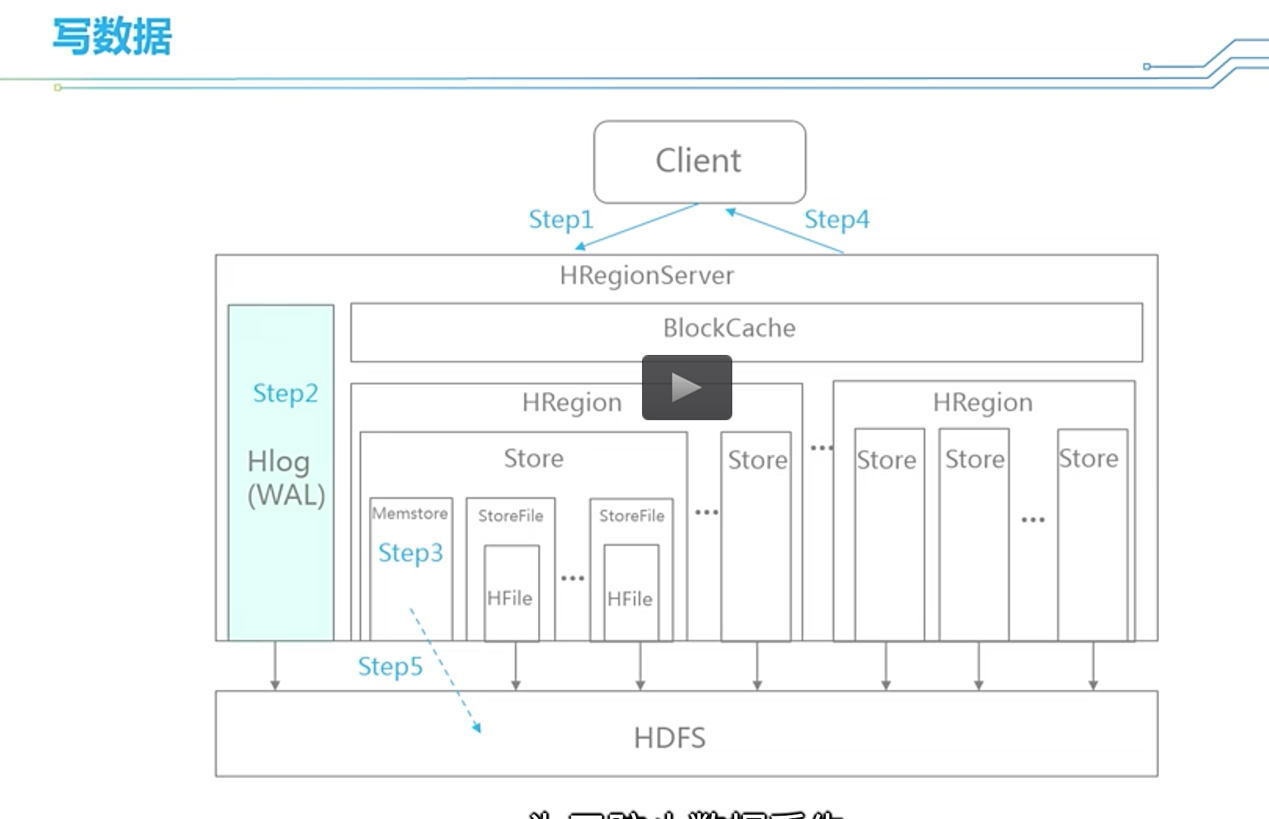
都会被先写入到这些HLog中

这样遇到集群重启

HRegionServer会重新运行

保存在HLog中的记录

恢复集群的数据



写入数据的过程

REgionServer

接收到客户端的数据写入请求

为了防止数据丢失

数据首先被写入HLog当中

接着数据被写入memstore

随后写入请求被返回到客户端

随着memstore被写满

memstore中的数据

会被异步的写入磁盘中

生成一个storefile

PUT实际上是一个upsert

即update加insert

从用户的角度看

就是如果一条记录不存在

就直接写入该记录

若记录存在就更新该记录

DELETE是HBase中的删除操作

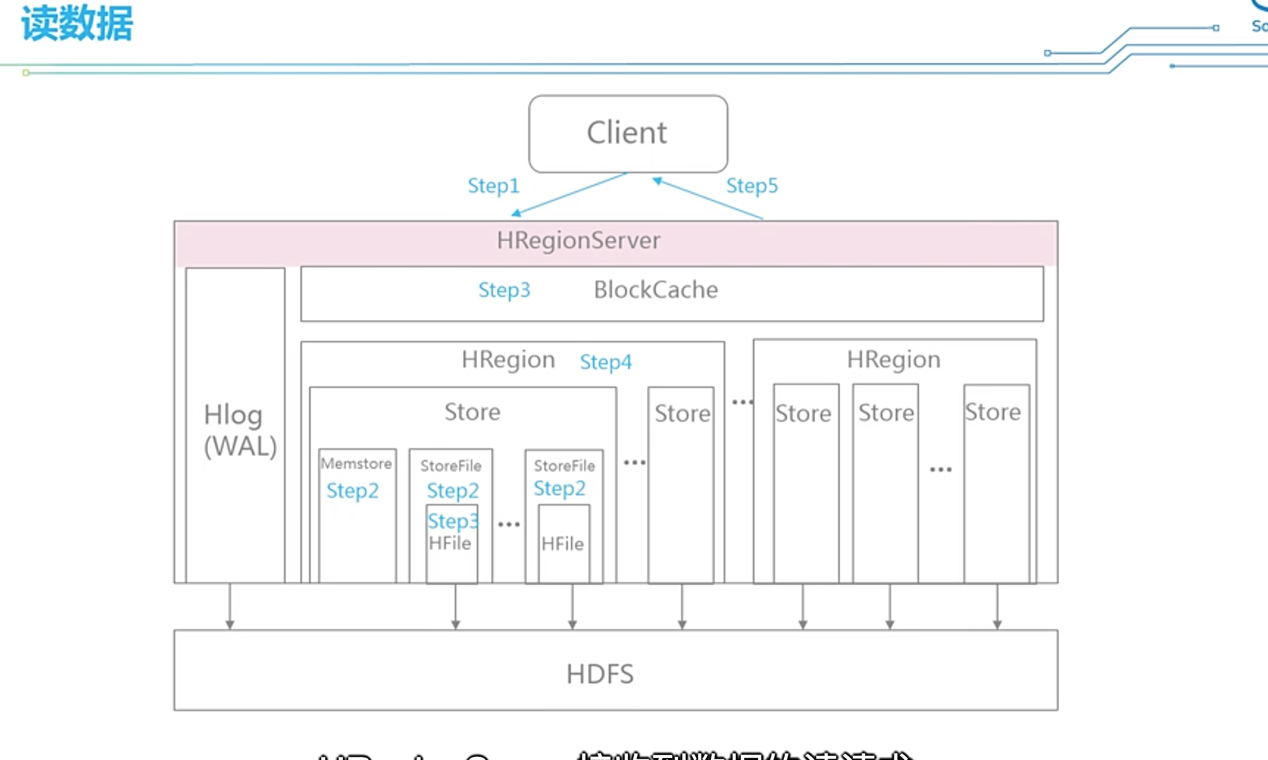
添加一条DELETE的记录

即delete marker

在读取的时候

hbase的scanner会将删除的数据

从结果中过滤掉



查询数据的过程

HRegionServer接收到数据的读请求

通过Region将读请求

发送到相关的Store

StoreScanner打开memstore

和相应的StoreFile

在读取StoreFile的数据块时

Scanner会尝试从BlockCache中

读取数据

若无法查到

则从StoreFile中读取数据

并存入blockcache中

从memstore和storefiles中

查询得到的数据会被进行merge-sort

并加入到最终的结果中

返回到客户端

GET是读取某条记录

Scan是扫描一段连续的记录

之前我们有提到列族最好不要太多

但是在读取时

需要访问很多的磁盘文件

所以hbase会对这些磁盘上的store file

进行合并

生成一个新的更大的文件

当一个store中的storefile

达到某个阀值的时候

默认是3

hbase便会选择部分

或者全部文件进行合并

每个Store都拥有自己的memstore

当一个region拥有的

所有memstore的数据量之和

达到一个阀值的时候

默认是128MB

就会触发memstore的flush

一个列族对应一个store

一个store对应一个storefile

如果一个Region拥有的Store过多

会造成每个写入磁盘的store file过小

随着小文件的增多

也会更加频繁的触发compaction

造成更多额外的I/O负担

所以不建议用户设置过多的列族

compaction：

compaction会优先选择

老的和小的文件来合并

compaction又分为两种

一种叫minor compaction

就是合并部分文件

一种叫major compaction

就是合并所有文件

在major compaction中

delete marker会被删除

HBase并不擅长处理表之间的关系

因为没有外键关联

无法处理如Join之类的操作

但是HBase提供了查询接口

用户如果有需要

可以自己实现相应的join功能

HBase虽然可以保证

同一region数据的读写的事务

但并不能处理传统意义上的事务

一系列写操作中访问的数据可能位于不同的分区服务器，这样的事务就变成[分布式事务](https://www.baidu.com/s?wd=%E5%88%86%E5%B8%83%E5%BC%8F%E4%BA%8B%E5%8A%A1&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkPW0vujfdP17bm1KBPH0Y0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPH6vPHnzPHckPj64P1ckrjfd)，实现原子性需要彼此协调，而协调是耗费时间的，每台机器在一个大事务过程中必须依次确认，这就需要一种协议确保一个事务中没有任何一台机器写操  
作失败。  
这种协调是昂贵的，会增加[延迟时间](https://www.baidu.com/s?wd=%E5%BB%B6%E8%BF%9F%E6%97%B6%E9%97%B4&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkPW0vujfdP17bm1KBPH0Y0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPH6vPHnzPHckPj64P1ckrjfd)，关键问题是，当协调没有完成时，其他操作是不能读取事务中写操作结果的，这是因为事务的all-or-  
nothing原理导致，万一协调过程发现某个写操作不能完成，那么需要将其他写操作成功的进行回滚。针对[分布式事务](https://www.baidu.com/s?wd=%E5%88%86%E5%B8%83%E5%BC%8F%E4%BA%8B%E5%8A%A1&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkPW0vujfdP17bm1KBPH0Y0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPH6vPHnzPHckPj64P1ckrjfd)的分布式协调对整体数据库性能有严重  
影响

比如分布式事务

另外HBase中只有主键索引

不像传统关系型数据那样

拥有二级索引和全文索引

对于非主键的查询并不高效

不得不进行全表扫描

当然现在有很多种途径

去实现二级索引和全文索引

但是在HBase的代码中并没有实现

对大对象的支持

传统关系型数据库中有BLOB

用来存储大对象

HBase对于存储大于50MB的大对象

并不擅长

往往存储几个对象

就需要把内存中的数据写入磁盘

过多的文件会频繁的触发compaction

并会产生大量额外的I/O

影响集群的性能

主键递增的数据

这样会造成一段时间内

某个region成为写入热点

影响数据写入的吞吐量

那么我会首先介绍一下

HDFS的设计原理和架构

然后重点讲一下

HDFS如何保证

数据的可靠性和可用性

那么最后会给大家简单介绍一下

HDFS当中读写数据的流程

首先我会介绍一下

HDFS的主要设计原理和架构

然后重点讲一下

HDFS是如何保证

数据的可靠性以及服务的可用性

还有扩展性的

最后我们一起看一下HDFS当中

基本的读取数据的流程

我们先来看一下HDFS的背景

分布式文件系统就是将文件系统

构建在一个分布式的集群上

数据集的大小

可能会超过单台物理机的容量

这个时候我们就必须把

这些数据分布的存储在多台机器上

HDFS主要是适用于以下三种场景

首先是我们需要存储

TB或者PB级别的大数据集

并且要支持单个的大文件

其次HDFS假设我们的使用模式是

一次写入、多次读取

而且是顺序地读取大量数据的情况

HDFS的随机访问性能并不高

最后HDFS对数据的可靠性提供保障

就不要求我们使用高可靠的服务器

在部分的节点出现故障的时候

可以继续使用

下面这几个场景就不适合使用HDFS

比如HDFS注重的是高吞吐量

而非低延迟

因此要求低延迟的数据访问

就不应该使用HDFS

另外HDFS也不能很好地

应对大量小文件的情况

因为所有的文件的元数据

都需要保存在内存中

因此HDFS可以支持的文件总数量

是受内存限制的

最后HDFS不支持单个文件的并发写入

而且当文件生成以后

对文件的修改也只支持Append操作

下面我们介绍一下HDFS当中的

Block概念

Block概念其实不是HDFS所特有的

对于磁盘而言也有Block的概念

那么对磁盘来说一个Block代表了

最小的读写单元

通常为512个字节

我们常见的文件系统也有Block的概念

大小通常在KB级别

并且是磁盘Block的整数倍

HDFS也有自己的Block

在HDFS当中

每个文件会被切分为多个Block

每个Block是一个独立的存储单元

这样做的好处是单个文件的大小

可以超出任意一台节点的磁盘容量

另外这样做也便于HDFS对文件

进行管理和复制

与常见的文件系统不同的是

HDFS的Block的大小要大的多

默认为128兆

较大的Block可以带来一些好处

首先是在读取数据的时候

较大的Block

减少了磁盘寻道的时间开销

这样读取一个大文件的时候

读取的速率就能近似于磁盘的传输速率

另外较大的Block

可以减少Block的总数

就像我们之前所说的

Block信息也要存储在内存中

因此较少的Block个数

可以减轻内存的压力

但是另一方面

如果Block的大小设置得太大

也会有一些缺点

比如在执行Map Reduce任务的时候

通常一个Map Task

会处理一个Block的数据

这就意味着较大的Block

会导致较少的Map Task

这并不利于充分利用集群的计算资源

因此我们在设置HDFS Block大小的时候

需要做一定的取舍

最后我想说明的是

Block对于HDFS而言是一个逻辑的概念

如果一个文件的大小小于的Block

那么HDFS还是会为这个文件分配一个Block

但是并不代表这个文件会

实际占用一个Block的存储空间

下面我们来介绍一下HDFS的架构

从这张图中可以看出

HDFS采用了master work的模式

一个HDFS集群

由一个NameNode和多个DataNode组成

NameNode负责管理

整个文件系统的命名空间和元数据

是HDFS的master节点

比如这个图中展示的

NameNode记录了home下面

data这个文件的复制份数为3

元数据文件的型式保存在

NameNode的本地磁盘上

主要包含了两个文件

fsimage和edit log

fsimage保存了整个文件系统的命名空间

而edit log记录了在集群运行期间

对文件系统的所有修改操作

比如创建了新的文件

重命名了已有的文件等

另外NameNode还需要管理

所有的DataNode

以及一个文件所有Block的存储位置

最后NameNode还要协调

客户端对数据的访问

我们再来看一下DataNode

Datanode属于HDFS的work节点

负责将HDFS当中的数据

存储在本地的文件系统里

图中绿色的部分就代表了

Datanode所存储的各个Block

Datanode要定时向

Namenode汇报状态

并且接受Namenode的指令

对数据进行管理

另外Datanode还要响应用户的读写请求

客户端就是用户访问HDFS的接口

例如客户端可以向

Namenode请求读取某个文件

或者写入新的文件等

具体的读写步骤

我们会在后面详细的介绍

下面我们来看一下

HDFS是如何保证数据的可靠性的

这里所讲的可靠性是指

当集群中部分的节点出现故障的时候

HDFS要保证用户的数据不会丢失

并且可以继续被访问

可以想到

最简单的保证可靠性的方法

就是对数据进行复制

HDFS当中每一个Block

会有多个副本默认为三份

这些副本分布的存储在

不同的Datanode上

这样部分Datanode出现故障

就不会导致数据丢失

复制的方法实现起来非常的简单

但是因为每一份数据都要被复制多份

因此会有较大的存储开销

为了解决这一缺陷

HDFS又引入了

Erasure Coding的机制

Erasure Coding通过

对数据Block进行编码

可以得到校验的Parity Block

Parity Block和数据Block

分布的存储在集群中

当部分数据丢失的时候

我们可以通过剩余的Block

来恢复丢失的数据

采用Erasure Coding的好处就是

可以在保证数据可靠性的同时

大幅的降低存储开销

当然另一方面

这需要一些额外的编解码的计算

下面我们就来详细看一下这两种机制

这张图展示的

就是通过复制来保证数据的可靠性

假设一个文件由ABC3个Block组成

那么默认情况下

每一个Block会有三个副本

HDFS会将这些副本分布的存储在

不同的Datanode上

如果HDFS拥有集群的网络拓扑信息

那么还会将这些Block尽量的分布在

不同的机柜上

这些措施都是为了保证

部分的Datanode出现故障时

用户的数据不会丢失

在这种机制下

假如我们需要存储的数据为1T

那么实际我们需要3T的存储空间

额外的存储空间开销为200%

那么我们再来看一下Erasure Coding

其实有很多种算法可以实现

Erasure Coding的这个机制

最常用的是Reed-Solomon算法

其他的算法还包括像异或

以及HitchHiker等

我们就以Reed-Solomon算法为例

假设我们有K个数据Block

经过编码后可以得到M个校验Block

我们把这些校验Block称为parity Block

我们将K加M个Block

分布的存储在集群中

在节点出现故障的时候

如果丢失的Block数量小于等于M

我们就可以通过剩余的Block来做解码

并还原出所有的数据

这里需要说明的是

丢失的Block既可以是数据Block

也可以是parity Block

K和M的值可以比较灵活的进行配置

通常情况下我们选K等于6、M等于3

这就意味着Erasure Coding

与复制的机制一样

可以承受最多3台Datanode出现故障

但额外的存储开销只有50%

远远低于我们之前讲的复制

说到Erasure Coding

这里顺便跟大家介绍一下

HDFS的存储布局

在引入Erasure Coding之前

HDFS的Block都是连续存储的

如这张图中上半部门展示的

每一个Block

都连续的存储在一台Datanode上

为了适应Erasure Coding的机制

我们引入了分条的存储布局

目前所有使用

Erasure Coding存储的文件

都是要按照分条来存储的

在分条的布局下

数据会被切分为较小的单元格

我们可以看这个图下半部分

每个单元格的大小是1M

6个单元格就组成了一个条带

并且存储在6台Datanode上

相对于连续存储来说

分条存储的一个优势是说

我们可以进行并发的数据读写

但是它也有一些缺点

比如像MapReduce这样的计算

会失去数据本地化

fsimage文件其实是[Hadoop](https://www.iteblog.com/archives/tag/hadoop/)文件系统元数据的一个永久性的检查点，其中包含[Hadoop](https://www.iteblog.com/archives/tag/hadoop/)文件系统中的所有目录和文件idnode的序列化信息；

、edits文件存放的是[Hadoop](https://www.iteblog.com/archives/tag/hadoop/)文件系统的所有更新操作的路径，文件系统客户端执行的所以写操作首先会被记录到edits文件中。

下面我们来看一下

HDFS如何保证可用性

对于HDFS而言

Namenode是一个单点故障

因此如果Namenode出现错误

无法提供服务的时候

整个集群都无法访问

所以这里所说的可用性

主要就是指的Namenode的可用性

Namenode的可用性

主要面临两种威胁

一个是Namenode出现故障

导致保存的元数据丢失

这种情况下集群将会完全不可用

甚至导致所有的用户数据都会丢失

另一种情况是

Namenode运行一段时间以后

需要重启

重启的原因可能是日常的维护等等

但是这个时候会有比较大的重启开销

导致用户无法使用服务

为了解决以上这些问题呢

我们可以让Namenode

将原数据备份到多个文件系统当中

比如同时写入本地和网络存储

另外HDFS还提供了

Secondary Namenode

和Standby Namenode

来提高可能性

下面我们就来详细看一下这两种方法

就像我们之前提到的

Namenode在运行的过程当中

会将用户所有的修改保存在

edit log中

这就会导致edit log

随着Namenode的运行

而不断增长

当Namenode需要重启的时候

需要读取fsimage

然后将edit log中的修改记录合并进去

这个时候如果edit log的内容太多

就会导致Namenode有很长的启动时间

我们可以用Secondary Namenode

来解决这个问题

通常Secondary Namenode

运行在跟主Namenode不同的节点上

并定期的对主Namenode的fsimage

和edit log进行合并

保证edit log的内容不会过长

这张图就展示了

Secondary Namenode的工作原理

首先Secondary Namenode

会通知主Namenode来更换edit log

主Namenode将当前使用的

edit log文件关闭

并创建新的edit log来记录

后续的修改操作

Secondary Namenode把原来的

edit log和fsimage拷贝到本地

然后进行合并

得到fsimage checkpoint

然后Secondary Namenode把

这份checkpoint拷贝回主Namenode

主Namenode就可以用这个checkpoint

来替换原来的fsimage了

这样就能保证在主Namenode上

edit log的长度不会过长

Secondary Namenode

虽然可以解决主Namenode的

启动开销问题

但是并不能真正带来高可用性

因为当主Namenode出现故障的时候

我们仍然需要重新启动一个Namenode

并把原来备份的fsimage和edit log

加载到内存中

然后还要等足够多的Datanode

向其汇报状态

之后新的Namenode

才能对用户提供服务

在一个规模比较大的集群上

这个过程通常需要至少三十分钟时间

为了克服这个问题

我们在HDFS当中又引入了

Standby Namenode

使用Standby Namenode的时候

集群中会同时存在两台Namenode

在任意一个时间点

只有一台Namenode可以对外提供服务

我们就称之为Active Namenode

另外一台就称之为Standby Namenode

这张图展示了

Standby Namenode的工作模式

其中Active Namenode

和Standby Namenode

通过可靠的共享存储来共享edit log

这就保证了Standby Namenode

拥有跟Active Namenode

一样的最新的状态

而另外所有的Datanode

需要同时向Active

和Standby Namenode回报状态

这样当Active Namenode

出现故障的时候

Standby Namenode可以

很快的成为新的Active Namenode

并对外提供服务

一般来说

从Active Namenode切换到

Standby Namenode

只有分钟级别的延迟

另外 Standby Namenode也具备

Secondary Namenode的功能

也就是说会定期的对

Active Namenode上的

fsimage和edit log进行合并

使用Standby Namenode的一个重点呢

就是要保证任意一个时间点

只有一台Namenode是Active的

否则就可能会导致元数据的损坏

这意味着当我们决定从

Active Namenode切换到

Standby Namenode的时候

我们必须保证原来的Active Namenode

不会再继续写入元数据

为了实现这一点

我们可以要求共享存储

只能授予一台Namenode写权限

或者我们可以远程终止

原来的Active Namenode的进程

关闭其网络端口

甚至是对原来的Active Namenode的节点

进行强行的断电

在客户端这边

我们也需要进行一些简单的配置

来适应Standby Namenode的场景

比如我们需要将HDFS URL

设置为逻辑主机名

这样可以映射到两台Namenode上

接下来我们看一下HDFS的扩展性

像之前我们说的

所有的文件的元数据都要保存在内存中

这就意味着HDFS所能管理的文件总数

是受Namenode内存限制的

为了突破这一限制

HDFS引入了一种叫做federation的机制

就是允许一个集群

可以配置多台Namenode

大家可以看这张图

每个Namenode会管理一个

独立的命名空间

每个命名空间会对应一个Block pool

每个Datanode要向

所有的Namenode注册

并且回报状态

每一个Datanode

可以存储多个Block pool当中的数据

另一方面的扩展性

其实讲的是

Hadoop FileSystem的扩展性

Hadoop定义了FileSystem的抽象的API

而HDFS其实只是其中的一种实现

Hadoop还包含了这张图里

所展示的这些不同的其他的实现

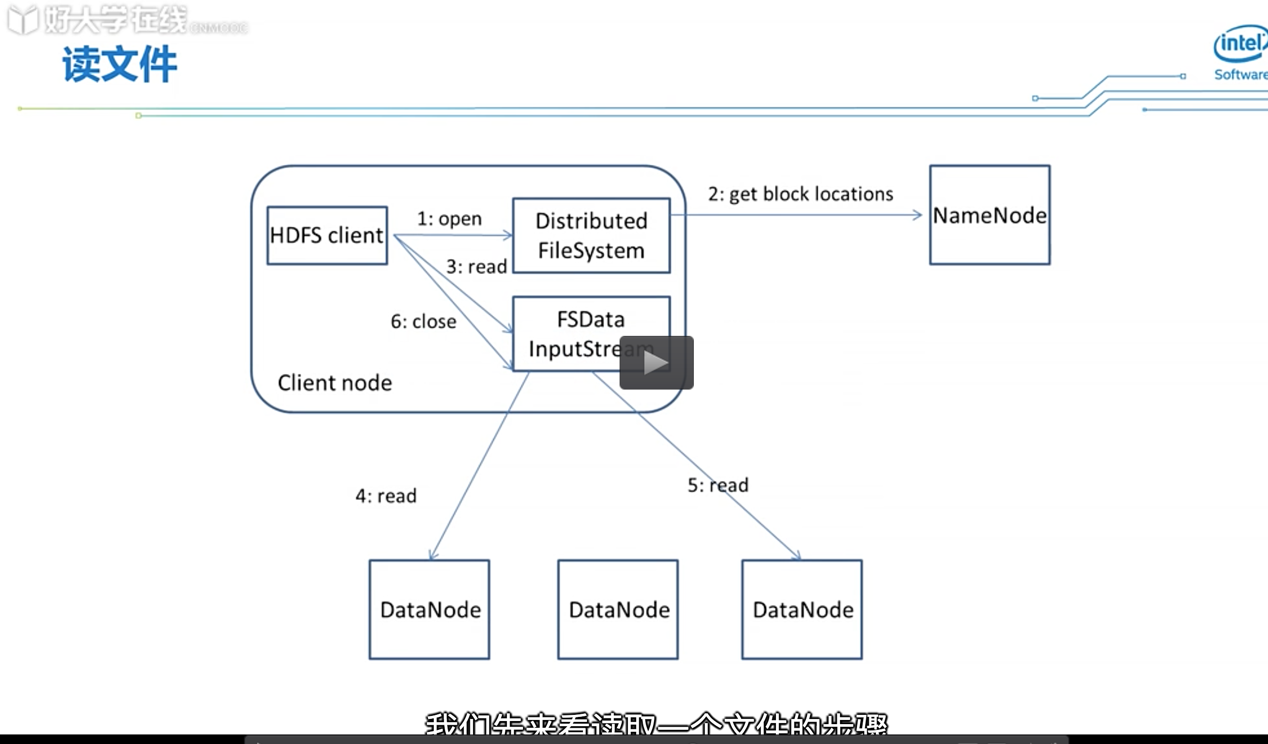
大家如果感兴趣的话

可以自己了解一下

最后我们来看一下

在HDFS当中读写数据的流程

我们先来看读取一个文件的步骤



首先客户端通过文件系统接口

打开要读取的文件

文件系统与Namenode进行通信

然后Namenode会返回

这个文件前几个Block的地址

我们这里所说的Block的地址是指

存储了这个Block的Datanode的列表

而且这个列表当中的Datanode

会按照它们与客户端之间的距离

从近到远进行排序

举个简单的例子

如果客户端所在的节点

也是一台Datanode

并且存储了要读取的Block

那么这个Datanode会出现在

这个列表当中的第一个

文件打开以后

客户端会得到一个inputstream

然后就可以通过这个inputstream

来读取数据了

大家可以看这个图中的第三步

当客户端调用read时

inputstream会根据Block的地址

连接到相应的Datanode上读取数据

当文件的第一批Block读取完毕以后

inputstream会再跟Namenode通信

然后来请求后续的Block地址

在读取一个Block的时候

如果发现地址当中的某个Datanode

出现了故障

就会自动的选择

列表当中的下一个Datanode来读取数据

这一系列的过程

对于客户端来说都是透明的

从用户的角度来看

就是在读取一个连续的数据流

从这一个流程中我们能够看出

数据流是分布在整个集群中的

Namenode只需要提供

文件的元数据信息就可以了

这样就大大减轻了Namenode的压力

下面我们再来看一下写文件的步骤

用户首先通过文件系统接口

创建一个文件

这里会跟Namenode进行通信

Namenode会进行一系列的检查

比如用户是否有权限创建这个文件等等

如果一切顺利的话

Namenode会在命名空间当中

为这个新的文件创建一条记录

文件创建成功以后

用户得到一个outputsteam

通过这个outputsteam

就可以写入数据了

在底层outputsteam通过

data streamer来发送数据

随着用户数据的写入

data streamer会跟Namenode进行通信

请求Namenode分配新的Block

对于每一个Block

data streamer

还会选择一组Datanode来存储这个Block

我们将这样一组Datanode

称为一个pipeline

pipeline当中Datanode的数量

就等于文件的复制份数

如这个图中所示的

默认的pipeline会由三台Datanode组成

之后data streamer会将用户的数据包

发送到pipeline中第一台Datanode上

这台Datanode再负责将数据包

拷贝到pipeline中下一台Datanode上

以此类推

直到pipeline中所有的Datanode都

接收到了数据

每一台Datanode在收到数据包以后

会发送一个ack确认信息

只有当pipeline中

所有的Datanode都确认以后

客户端才认为这个数据包是发送成功的

在写数据的过程当中

如果某一个Datanode出现故障

则会将其从当前的pipeline中移除

客户端会重发已经发送

但是还没有被确认的数据包

这样做的目的是为了保证

pipeline中的后续的Datanode

能够收到数据

在数据写入完成以后

Namenode会将这个文件

标记为复制份数不足

并且在后台进行一个异步的复制

直到文件的副本数达到用户设定的值