目录

[使用MR访问Hbase 1](#_Toc520923442)

[MR访问Hbase的方法 2](#_Toc520923443)

[案例需求 2](#_Toc520923444)

[转化需求 3](#_Toc520923445)

[Hbase的存储结构 4](#_Toc520923446)

[宏观结构 4](#_Toc520923447)

[RegionServer组成 4](#_Toc520923448)

[Region内部 4](#_Toc520923449)

[Store内部 4](#_Toc520923450)

[总结 5](#_Toc520923451)

[Hbase组件的功能和意义 5](#_Toc520923452)

[WAL（write-ahead log）预写日志 5](#_Toc520923453)

[WAL可以关 5](#_Toc520923454)

[WAL滚动 5](#_Toc520923455)

[WAL最后怎么样了？ 6](#_Toc520923456)

[MemStore 6](#_Toc520923457)

[MemStore的工作过程 6](#_Toc520923458)

[MemStore的设计意义 6](#_Toc520923459)

[HFile介绍 6](#_Toc520923460)

[数据在hbase上是怎么存的 7](#_Toc520923461)

[数据在hbase上是怎么取的 7](#_Toc520923462)

[设计原则 7](#_Toc520923463)

[HBase列族的设计原则 7](#_Toc520923464)

[rowkey设计原则 8](#_Toc520923465)

[A唯一原则 8](#_Toc520923466)

[B长度原则 8](#_Toc520923467)

[C热点问题 8](#_Toc520923468)

# 使用MR访问Hbase

mr居然还能访问hbase有没感觉很神奇？因为Hbase速度确实是快啊，二很适合小条目的存储。我们最常用的MR分析起来还是很灵活的，若能用MR去计算Hbase上的数据是什么感觉？

比如在hbase上有这样一个表，行键是人，有一个列是这个人发表过的帖子。我想统计他的帖子里词频或是一些关键字的个数等等。或者想group by 等复杂的hbase中扫描器和过滤器不好完成的需求使用MR适合，再加上hbase的速度很快，嗯完美。

## MR访问Hbase的方法

你只需要记住这么几个点就可以了。首先你要会写mr嗯。然后有这么几个地方和正常的MR不一样。

A config的设置，需要结合hbase做一些设置，比如要连上hbase

B job的Mapper的设置，和之前不太一样，因为这次我们是从hbase上读取数据

C Mapper的写法，同理读取的都是Hbase上的内容Mapper写法略有区别

D Mapper写法的具体区别[

1 继承TableMapper<outKey,outValue> 而不是Mapper

2 输入的内容是hbase上的行键和Result

]

### 案例需求

统计hbase上user表中所有单词在每个行键内的的词频  
探索步骤

首先我们先研究一下我们的简单MR程序的语法，看看它的输入输出都是什么样的感觉。

所以我们发现，Mapper的键和值其实就是hbase记录的键和值。知道这个关键信息之后我们就可以进行计算了。

public static class ForMapper extends TableMapper<Text,Text>{  
 @Override  
 protected void map(ImmutableBytesWritable key, Result value, Context context) throws IOException, InterruptedException {  
 System.*out*.println(key+","+value);  
 }  
}  
public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException, URISyntaxException {  
 //配置job访问hbase  
 Configuration config=new Configuration();  
 config.set("hbase.zookeeper.quorum","192.168.208.150");  
 config.set("hbase.zookeeper.property.clientPort", "2181");  
 Job job= Job.*getInstance*(config);  
 Scan scan=new Scan();  
 scan.addColumn(Bytes.*toBytes*("info"),Bytes.*toBytes*("text"));  
 //初始化Mapreducejob通过它将扫描器和mr进行结合  
 TableMapReduceUtil.*initTableMapperJob*("user", scan, ForMapper.class,Text.class,Text.class,job);  
 String output="file:///E:/output";  
 FileSystem fileSystem= FileSystem.*get*(new URI(output),new Configuration());  
 if(fileSystem.exists(new Path(output))){  
 fileSystem.delete(new Path(output), true);  
 }  
 FileOutputFormat.*setOutputPath*(job, new Path("file:///E:/output"));  
  
 job.waitForCompletion(true);  
  
}

### 转化需求

我们分析一下需求发现要计算的是单词在行键内的词频这其实就是倒排索引嘛，统计每个单词在不同的行中的次数

public class ForInverse {  
 public static class MyMapper extends TableMapper<Text,Text> {  
 private static Text *out*=new Text();  
 @Override  
 // hbase 表中的key 根据scan得到的value  
 protected void map(ImmutableBytesWritable key, Result value, Context context) throws IOException, InterruptedException {  
 String strValue=new String(value.value());//获得hbase表中的内容  
 String[] words=strValue.split(" ");  
 for(String s:words){  
 context.write(new Text(s+"->"+new String(key.get())),new Text());  
 }  
 }  
 }  
 public static class MyCombiner extends Reducer<Text,Text,Text,Text> {  
 public static Text *out*=new Text();  
 @Override  
 protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {  
 int count=0;  
 for(Text v:values){  
 count++;  
 }  
 System.*out*.println(key.toString());  
 String [] strs=key.toString().split("->");  
 System.*out*.println(Arrays.*toString*(strs[1].getBytes())+","+strs[1].equals("a")+"----------------------");  
 context.write(new Text(strs[0]),new Text(strs[1]+","+count));  
 }  
  
 }  
 public static class MyReducer extends Reducer<Text,Text,Text,Text>{  
 @Override  
 protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {  
 System.*out*.println(Arrays.*toString*(key.getBytes())+"==========================");  
 StringBuilder sb=new StringBuilder();  
 for(Text text:values){  
 sb.append(text.toString()+"#");  
 }  
 context.write(key,new Text(sb+""));  
 }  
 }  
  
  
 public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException, URISyntaxException {  
 Configuration config=new Configuration();  
 config.set("hbase.zookeeper.quorum","192.168.208.150");  
 config.set("hbase.zookeeper.property.clientPort", "2181");  
 Job job= Job.*getInstance*(config);  
 job.setCombinerClass(MyCombiner.class);  
 job.setReducerClass(MyReducer.class);  
 Scan scan=new Scan();  
 scan.addColumn(Bytes.*toBytes*("info"),Bytes.*toBytes*("text"));  
 TableMapReduceUtil.*initTableMapperJob*("user1", scan, MyMapper.class,Text.class,Text.class,job);  
 String output="file:///E:/output";  
 FileSystem fileSystem= FileSystem.*get*(new URI(output),new Configuration());  
 if(fileSystem.exists(new Path(output))){  
 fileSystem.delete(new Path(output), true);  
 }  
 FileOutputFormat.*setOutputPath*(job, new Path("file:///E:/output"));  
  
 job.waitForCompletion(true);  
 }  
  
}

# Hbase的存储结构

我们从宏观结构和具体的组件内部这两个方面来了解hbase的存储架构的设计。

## 宏观结构

关键词：©和RegionServer

hbase由HMaster机器和RegionServer机器构成，基本架构也是主从式的。但是HMaster的作用并不像Hadoop上的NameNode那么关键。如果HMaster挂了系统其实还是可以正常运行的，因为HMaster并不负责数据的写入和读取。但是HMaster也不能挂的时间太长因为，因为一些创建表的那个DDL的操作还需要HMaser

而RegionServer才是数据存储的关键 一个RegionServer里会有多个Region，而Region

里还有很多内容，这我们就需要探寻一下Hbase内部了。

## RegionServer组成

RegionServer是数据存储的关键位置，我们先来了解一下一个RegionServer里都有什么。一共有这样几个东东。

1 一个WAL（write-ahead-log）预写日志

2 多个Region每个Region都有起始rowkey和结束rowkey

## Region内部

Region的内部又有多个Store每个region内部都有多个store实例，一个store对应一个列族的数据，如果一个表有两列族那一个Region里就有两个store

### Store内部

一个store里又有MemStore和HFile两部分组成。这两东西是干嘛的我们马上会说到。

## 总结

Hbase的宏观结构有两大核心一个是HMaster一个是RegionServer。其实RegionServer是关于数据存储 的我们还要具体研究它。

RegionServer的内部又分为一个预写日志，和多个Region。

Region也不是省油的灯内部还有组件。一个Region包含多个Store，每个Store对应多个列族。Store里还有组件分别是MemStore和HFile。

所以关系如下：

HMaster

RegionServer\*N

WAL

Region

Store\*N

MemStore

HFile

# Hbase组件的功能和意义

我们上面只是介绍了Hbase中都有哪些组件，以及这些组件之间的关系。下面我们以数据的流向顺序介绍一些上面那些组件的具体功能和设计的目的。

## WAL（write-ahead log）预写日志

数据来了先到WAL它是直接存储在HDFS上的。所以数据到达Region先写入WAL的目的是解决宕机之后的数据恢复问题。因为它直接写在HDFS上所以就算是Region挂了，数据依然存在。

### WAL可以关

根据上面的信息我们不难想到每次写入WAL必定有开销，所以Hbase允许把它关掉，但是这样就不能保证数据可以恢复了。我们可以通过setDurability(Durability.SKIP\_WAL)来关掉它。也可以延时同步Mutation.setDuribility(Durability.ASYNC\_WAL)默认是一秒钟。这样可减少对WAL的操作。

### WAL滚动

还记得shuffle中的环形缓冲区嘛？环形的好处是写入效率高而且大小稳定。WAL也是这样的。WAL会检查实际的操作和持久化到HDFS中的实际情况。已经持久化过的会装到oldlogs里，以保证WAL的大小。触发WAL滚动的条件有两个一个是WAL所在的文件块要满了（一个就是大小到阀值了。这个阀值的公式的意思就是块大小乘以一个系数，默认是95%，到块的百分之九十五就滚动。

### WAL最后怎么样了？

WAL早晚要被归档，或是被删除。具体什么时候归档删除很简单。如果数据已经持久化了就是操作完成了，那就归档，进入oldlog文件夹。如果已经没有任何引用指向这个WAL的时候Master会定期的将它删除。

## MemStore

数据的下一站是MemStore，每个Region有多个Store，每个Store里有一个Memstore。数据写入WAL后会写入MemStore而MemStore见名知意是内存的对象，这货放在内存里。

### MemStore的工作过程

数据写入到WAl之后会被写入到MemStore它是放在内存中的。当MemStore满了的时候就会被刷写到HFile上。而Hfile是存储在HDFS上的。HFile就厉害了是数据存储的实体一会再说它。现在的流程是WAL（hdfs）->MemStore(memory)->HFile(hdfs)。所以从Hdfs到内存再到hdfs这是折腾啥呢？

### MemStore的设计意义

HDFS不适合修改。为什说这句话？因为hbase是数据库，数据就要支持随机存取。而且，hbase的底层使用的是hdfs。hdfs上的文件是不可修改的。所以就要求hbase在存储数据时尽量将数据按顺序存储起来这样才方便管理。所以MemStore功能一就是将数据进行排序整理，使用的是LSM树的算法。

二，数据在内存中处理一定是快的。这样让输在Mestore里先存储一段时间等MemStore满了之后再写入HDFS这样可以减少一些对HDFS的操作以提高工作效率。

hbase应该是长在hdfs上的。这句话我先放在这里埋伏笔

好了，我们开始。

## HFile介绍

HFile几乎是最终文件存储的位置了，其实HFile里还会再次细分，但是有些太细了我们暂不用关心。我们需要知道的是HFile是数据的真正载体就可以了。然后我们需要知道的是hbas在CRUD时对HFile真正的操作是怎样的。hbase是数据库需要随机存储。但是HDFS却不是随机存储系统。要么添加，要么整个删除，没法局部修改，所以Hbase要怎么做呢？添加！

hbase添加是在hdfs上添加一条

hbase删除数据还是在hdfs上添加一条

hbase修改还是在hdfs上添加一条，都是添加操作。但是永远都是添加那不会爆掉嘛？不会hbase会对HFile每隔一段时间进行一次合并。在合并的时候会发现已经删除的数据等。所以会将多个HFile文件合并成一个。就这样完成操作。Hbase表结构和Rowkey设计原则

## 数据在hbase上是怎么存的

A 数据存储：我们之前有介绍过，hbase上存储数据的关键是列族。每个列族中的内容都存储在一个文件中。多个列族可以存储在一个region中，一个region只能运行在一个机器上。一个机器可以运行多个region。这是对我们之前的认识的一个回顾。

B 记录存储：每条记录可能有多个版本（时间戳不同）多个版本的数据是连续存储的，多个版本存储时需要添加时间戳，这些时间戳按降序排列，所以这样可以使用户取到的是最新的记录。

C 数据存储在Region中会越来越大性能也随之越来越差，所以Region会进行拆分，当然由于数据的删除会导致每个region变小所以Region也会进行合并。

## 数据在hbase上是怎么取的

A 存储顺序： rowkey>family>qualifier>timestamp >value

B 查询规则：

i:用户可以按行键检索一行数据，这样可以有效地减少查询特定行和行范围的时间

ii:指定列族可以减少查询文件的时间

iii:指定列限定符因为系统要将数据送到过滤器上进行检查所以只能小幅度提高查询效率

iii:指定值查询很常用但是系统也需要通过过滤的方式检索，由于查询效率依照存储顺序从左到右递减，所以再设计的时候可以考虑将重要的经常使用的信息向左移动，从而提高查询效率。

# 设计原则

## [HBase列族的设计原则](http://blog.csdn.net/javajxz008/article/details/51892967)

A 长度短 列族其实是可以没有实际意义的，因为实际上列族也不会太多，有个文档对照一下也完全可以。所以不像mysql数据库表一样要把列名起的清楚。Hbase中不需要

B 业务上相似的内容放入同一列族下，业务上相似的内容一般在一起查询的概率要高一些，所以放在同一列族下可以提高查询效率。

## rowkey设计原则

### A唯一原则

不用多说行键一定是要保证唯一的通过行键是可以唯一确定一行数据的。

### B长度原则

Rowkey理论上长度能达到64k实际中一般是10-100kb一般是越短越好，原因是这样的。hbase存的是大量数据，如果rowkey大的话仅rowkey就占用了大量空间，比如100b的rowkey 存100条就是1k 一千条就是10k一万条就是100k十万条就是1M，一亿条就是一个G。所以row可以应该在能保证唯一和功能的前提下越短越好。

还有一个问题是rowkey的长度最好是一致的。比如7并不会排在11的前面而是后面，但是07会排在11的前面。

### C热点问题（散列原则）

由于region存储是按行键存储的，如果行键的值都是类似的那就容易导致数据都堆在一个region上产生了热点问题。

所以可以通过，rowkey应该尽量的散列化。

1 加盐在rowkey前生成随机数2 反转把最没有用的放在前面以增加随机性，比如手机号，一个号段太多反转就可以了，或者使用MD5加密rowkey，解决热点问题。