# HBase教案

# 12、HBase与MapReduce的集成

HBase当中的数据最终都是存储在HDFS上面的，HBase天生的支持MR的操作，我们可以通过MR直接处理HBase当中的数据，并且MR可以将处理后的结果直接存储到HBase当中去

需求：读取HBase当中一张表的数据，然后将数据写入到HBase当中的另外一张表当中去。注意：我们可以使用TableMapper与TableReducer来实现从HBase当中读取与写入数据

<http://hbase.apache.org/2.0/book.html#mapreduce>

## 需求一：读取myuser这张表当中的数据写入到HBase的另外一张表当中去

这里我们将myuser这张表当中f1列族的name和age字段写入到myuser2这张表的f1列族当中去

### 第一步：创建myuser2这张表

注意：列族的名字要与myuser表的列族名字相同

hbase(main):010:0> create 'myuser2','f1'

### 第二步：创建maven工程，导入jar包

注意：在之前工程导入jar包的基础上，添加以下这些jar包即可

*<!-- https://mvnrepository.com/artifact/org.apache.hbase/hbase-*mapreduce -->  
<dependency>  
 <groupId>org.apache.hbase</groupId>  
 <artifactId>hbase-mapreduce</artifactId>  
 <version>2.0.0</version>  
</dependency>  
  
<dependency>  
 <groupId>org.apache.hadoop</groupId>  
 <artifactId>hadoop-client</artifactId>  
 <version>2.7.5</version>  
</dependency>  
<dependency>  
 <groupId>org.apache.hadoop</groupId>  
 <artifactId>hadoop-hdfs</artifactId>  
 <version> 2.7.5</version>  
</dependency>  
  
<dependency>  
 <groupId>org.apache.hadoop</groupId>  
 <artifactId>hadoop-common</artifactId>  
 <version>2.7.5</version>  
</dependency>

### 第三步：开发MR的程序

#### 定义mapper类

public class HBaseMapper extends TableMapper<Text,Put> {

/\*\*

\* @param key rowkey

\* @param value 封装了我们一行数据

\* @param context

\* @throws IOException

\* @throws InterruptedException

\*/

@Override

protected void map(ImmutableBytesWritable key, Result value, Context context) throws IOException, InterruptedException {

// f1 name age f2 xxx

//获取到我们的rowkey

byte[] bytes = key.get();

Put put = new Put(bytes);

//获取Result当中所有的列

List<Cell> cells = value.listCells();

for (Cell cell : cells) {

//判断属于哪一个列族

byte[] family = CellUtil.cloneFamily(cell);

//获取cell属于哪一个列

byte[] qualifier = CellUtil.cloneQualifier(cell);

if(Bytes.toString(family).equals("f1")){

if(Bytes.toString(qualifier).equals("name") || Bytes.toString(qualifier).equals("age")){

put.add(cell);

}

}

}

if(!put.isEmpty()){

context.write(new Text(Bytes.toString(bytes)),put);

}

}

}

#### 定义reducer类

/\*\*

\* Text key2的类型

\* Put value2类型

\* ImmutableBytesWritable k3的类型

\* V3的类型？？？

\* put 'myuser2','rowkey','f1:name','zhangsan'

\* javaAPI来写通过put对象即可

\*

\*/

public class HBaseReducer extends TableReducer<Text,Put,ImmutableBytesWritable> {

/\*\*

\*

\* @param key 就是我们的key2

\* @param values 就是我们的v2

\* @param context 将我们的数据往外写出去

\* @throws IOException

\* @throws InterruptedException

\*/

@Override

protected void reduce(Text key, Iterable<Put> values, Context context) throws IOException, InterruptedException {

for (Put put : values) {

context.write(new ImmutableBytesWritable(key.toString().getBytes()),put);

}

}

}

#### 定义程序运行main方法

public class HBaseMrMain extends Configured implements Tool {

@Override

public int run(String[] args) throws Exception {

Job job = Job.getInstance(super.getConf(), "hbaseMR");

Scan scan = new Scan();

//使用工具来来初始化我们的mapper类

/\*\*

\* String table, Scan scan,

Class<? extends TableMapper> mapper,

Class<?> outputKeyClass,

Class<?> outputValueClass, Job job

\*/

TableMapReduceUtil.initTableMapperJob("myuser",scan,HBaseMapper.class, Text.class, Put.class,job);

/\*\*

\* String table,

Class<? extends TableReducer> reducer, Job job

\*/

TableMapReduceUtil.initTableReducerJob("myuser2",HBaseReducer.class,job);

boolean b = job.waitForCompletion(true);

return b?0:1;

}

public static void main(String[] args) throws Exception {

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum", "node01:2181,node02:2181");

int run = ToolRunner.run(configuration, new HBaseMrMain(), args);

System.exit(run);

}

}

### 第四步：运行

#### 运行第一种方式：本地运行

直接选中main方法所在的类，运行即可

#### 运行第二种方式：打包集群运行

注意，我们需要使用打包插件，将HBase的依赖jar包都打入到工程jar包里面去

##### 第一步：pom.xml当中添加打包插件

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-shade-plugin</artifactId>

<version>2.4.3</version>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>shade</goal>

</goals>

<configuration>

<minimizeJar>true</minimizeJar>

</configuration>

</execution>

</executions>

</plugin>

##### 第二步：代码当中添加

job.setJarByClass(HBaseMain.**class**);

##### 第三步：使用maven打包

然后打包，将打好的jar包上传到linux服务器，然后执行

yarn jar hbaseStudy-1.0-SNAPSHOT.jar cn.itcast.hbasemr.HBaseMR

或者我们也可以自己设置我们的环境变量，然后运行original那个比较小的jar包

export HADOOP\_HOME=/export/servers/hadoop-2.7.5/

export HBASE\_HOME=/export/servers/hbase-2.0.0/

export HADOOP\_CLASSPATH=`${HBASE\_HOME}/bin/hbase mapredcp`

yarn jar original-hbaseStudy-1.0-SNAPSHOT.jar cn.itcast.hbasemr.HbaseMR

## 需求二：读取HDFS文件，写入到HBase表当中去

读取hdfs路径/hbase/input/user.txt，然后将数据写入到myuser2这张表当中去

### 第一步：准备数据文件

准备数据文件，并将数据文件上传到HDFS上面去

hdfs dfs -mkdir -p /hbase/input

cd /export/servers/

vim user.txt

0007 zhangsan 18

0008 lisi 25

0009 wangwu 20

上传到hdfs上面去

hdfs dfs -put user.txt /hbase/input

### 第二步：开发MR程序

#### 定义mapper类

public class HdfsMapper extends Mapper<LongWritable,Text,Text,NullWritable> {

/\*

我们map阶段没有做任何处理，直接将我们的数据写出去

\*/

@Override

protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

context.write(value,NullWritable.get());

}

}

#### 定义reducer类

public class HBaseWriteReducer extends TableReducer<Text,NullWritable,ImmutableBytesWritable> {

/\*\*

\* @param key

\* @param values

\* @param context

\* @throws IOException

\* @throws InterruptedException

\*/

@Override

protected void reduce(Text key, Iterable<NullWritable> values, Context context) throws IOException, InterruptedException {

String[] split = key.toString().split("\t");

Put put = new Put(split[0].getBytes());

put.addColumn("f1".getBytes(),"name".getBytes(),split[1].getBytes());

put.addColumn("f1".getBytes(),"age".getBytes(),split[2].getBytes());

//ImmutableBytesWritable 可以装我们的rowkey，或者装我们的value值等等

context.write(new ImmutableBytesWritable(split[0].getBytes()),put);

}

}

#### 定义程序运行main方法

public class Hdfs2HBaseMain extends Configured implements Tool {

@Override

public int run(String[] args) throws Exception {

Job job = Job.getInstance(super.getConf(), "hdfsToHbase");

//第一步：读取文件，解析成key，value对

job.setInputFormatClass(TextInputFormat.class);

TextInputFormat.addInputPath(job,new Path("hdfs://node01:8020/hbase/input"));

//第二步：自定义mapper，接收k1 v1转化成新的k2 v2输出

job.setMapperClass(HdfsMapper.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(NullWritable.class);

//第三步：分区

//第四步：排序

//第五步：规约

//第六步：分组

//第七步：reduce逻辑，接收K2 v2 转换成新的k3 v3输出

TableMapReduceUtil.initTableReducerJob("myuser2",HBaseWriteReducer.class,job);

//第八步：输出数据

boolean b = job.waitForCompletion(true);

return b?0:1;

}

public static void main(String[] args) throws Exception {

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum", "node01:2181,node02:2181");

int run = ToolRunner.run(configuration, new Hdfs2HBaseMain(), args);

System.exit(run);

}

}

public class Hdfs2Hbase extends Configured implements Tool{

@Override

public int run(String[] args) throws Exception {

Job job = Job.getInstance(super.getConf(), "hdfs2Hbase");

job.setJarByClass(Hdfs2Hbase.class);

job.setInputFormatClass(TextInputFormat.class);

TextInputFormat.addInputPath(job,new Path("hdfs://node01:8020/hbase/input"));

job.setMapperClass(HdfsMapper.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(NullWritable.class);

TableMapReduceUtil.initTableReducerJob("myuser2",HBaseReducer.class,job);

job.setNumReduceTasks(1);

boolean b = job.waitForCompletion(true);

return b?0:1;

}

public static void main(String[] args) throws Exception {

Configuration configuration = HBaseConfiguration.create();

int run = ToolRunner.run(configuration, new Hdfs2Hbase(), args);

System.exit(run);

}

public static class HdfsMapper extends Mapper<LongWritable,Text,Text,NullWritable>{

@Override

protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

context.write(value,NullWritable.get());

}

}

public static class HBaseReducer extends TableReducer<Text,NullWritable,ImmutableBytesWritable>{

@Override

protected void reduce(Text key, Iterable<NullWritable> values, Context context) throws IOException, InterruptedException {

String[] split = key.toString().split("\t");

Put put = new Put(Bytes.toBytes(split[0]));

put.addColumn("f1".getBytes(),"name".getBytes(),split[1].getBytes());

put.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(Integer.parseInt(split[2])));

context.write(new ImmutableBytesWritable(Bytes.toBytes(split[0])),put);

}

}

}

## 需求三：通过bulkload的方式批量加载数据到HBase当中去

加载数据到HBase当中去的方式多种多样，我们可以使用HBase的javaAPI或者使用sqoop将我们的数据写入或者导入到HBase当中去，但是这些方式不是慢就是在导入的过程的占用Region资源导致效率低下，我们也可以通过MR的程序，将我们的数据直接转换成HBase的最终存储格式HFile，然后直接load数据到HBase当中去即可

HBase中每张Table在根目录（/HBase）下用一个文件夹存储，Table名为文件夹名，在Table文件夹下每个Region同样用一个文件夹存储，每个Region文件夹下的每个列族也用文件夹存储，而每个列族下存储的就是一些HFile文件，HFile就是HBase数据在HFDS下存储格式，所以HBase存储文件最终在hdfs上面的表现形式就是HFile，如果我们可以直接将数据转换为HFile的格式，那么我们的HBase就可以直接读取加载HFile格式的文件，就可以直接读取了

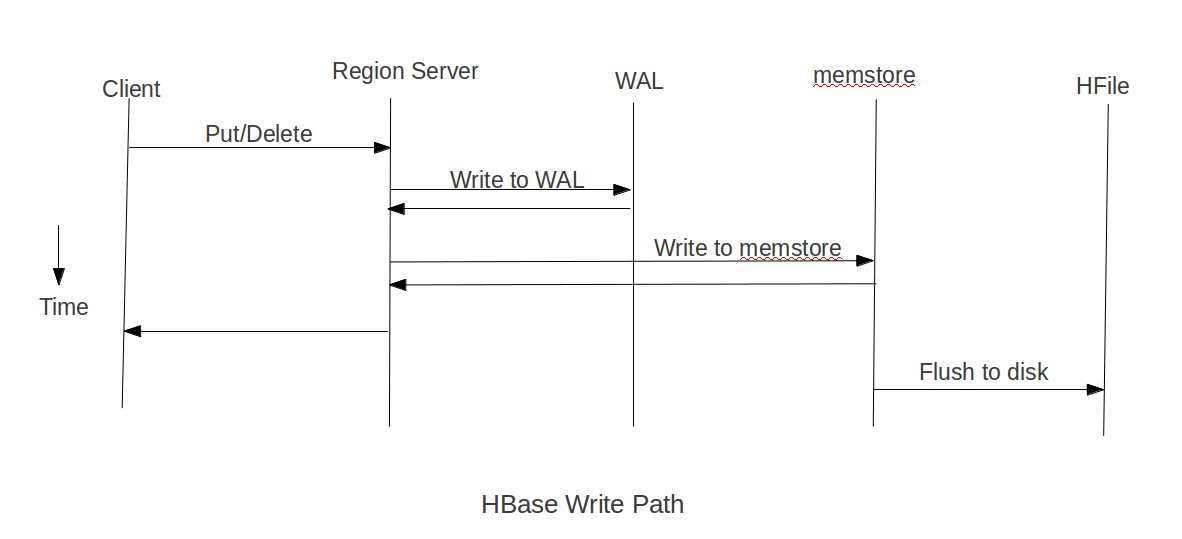
优点：

1.导入过程不占用Region资源

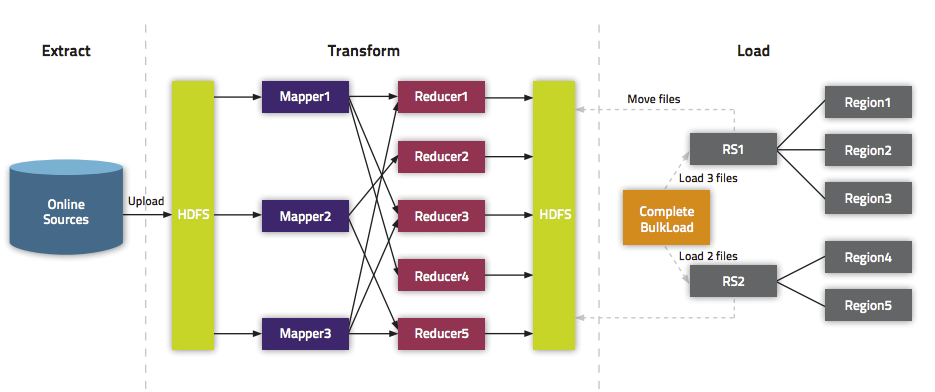
2.能快速导入海量的数据

3.节省内存

HBase数据正常读写流程



使用bulkload的方式将我们的数据直接生成HFile格式，然后直接加载到HBase的表当中去



需求：将我们hdfs上面的这个路径/hbase/input/user.txt的数据文件，转换成HFile格式，然后load到myuser2这张表里面去

### 第一步：定义我们的mapper类

/\*\*

\* LongWritable k1类型

\* Text V1类型

\* ImmutableBytesWritable rowkey

\* Put 插入的对象

\*/

public class BulkLoadMapper extends Mapper<LongWritable,Text,ImmutableBytesWritable,Put> {

@Override

protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

String[] split = value.toString().split("\t");

Put put = new Put(split[0].getBytes());

put.addColumn("f1".getBytes(),"name".getBytes(),split[1].getBytes());

put.addColumn("f1".getBytes(),"age".getBytes(),split[2].getBytes());

context.write(new ImmutableBytesWritable(split[0].getBytes()),put);

}

}

### 第二步：开发我们的main程序入口类

public class BulkLoadMain extends Configured implements Tool {

@Override

public int run(String[] args) throws Exception {

Configuration conf = super.getConf();

Connection connection = ConnectionFactory.createConnection(conf);

Table table = connection.getTable(TableName.valueOf("myuser2"));

Job job = Job.getInstance(conf, "bulkLoad");

//读取文件，解析成key,value对

job.setInputFormatClass(TextInputFormat.class);

TextInputFormat.addInputPath(job,new Path("hdfs://node01:8020/hbase/input"));

//定义我们的mapper类

job.setMapperClass(BulkLoadMapper.class);

job.setMapOutputKeyClass(ImmutableBytesWritable.class);

job.setMapOutputValueClass(Put.class);

//reduce过程也省掉

/\*\*

\* Job job, Table table, RegionLocator regionLocator

\* 使用configureIncrementalLoad来进行配置我们的HFile加载到哪一个表里面的哪一个列族里面去

\*/

HFileOutputFormat2.configureIncrementalLoad(job,table,connection.getRegionLocator(TableName.valueOf("myuser2")));

//设置我们的输出类型，将我们的数据输出成为HFile格式

job.setOutputFormatClass(HFileOutputFormat2.class);

//设置我们的输出路径

HFileOutputFormat2.setOutputPath(job,new Path("hdfs://node01:8020/hbase/hfile\_out"));

boolean b = job.waitForCompletion(true);

return b?0:1;

}

public static void main(String[] args) throws Exception {

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum", "node01:2181,node02:2181");

int run = ToolRunner.run(configuration, new BulkLoadMain(), args);

System.exit(run);

}

}

### 第三步：将代码打成jar包然后进行运行

yarn jar original-hbaseStudy-1.0-SNAPSHOT.jar cn.itcast.hbasemr.HBaseLoad

### 第四步：开发代码，加载数据

将我们的输出路径下面的HFile文件，加载到我们的hbase表当中去

public class LoadData {

public static void main(String[] args) throws Exception {

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.property.clientPort", "2181");

configuration.set("hbase.zookeeper.quorum", "node01,node02,node03");

Connection connection = ConnectionFactory.createConnection(configuration);

Admin admin = connection.getAdmin();

Table table = connection.getTable(TableName.valueOf("myuser2"));

LoadIncrementalHFiles load = new LoadIncrementalHFiles(configuration);

load.doBulkLoad(new Path("hdfs://node01:8020/hbase/output\_hfile"), admin,table,connection.getRegionLocator(TableName.valueOf("myuser2")));

}

}

或者我们也可以通过命令行来进行加载数据

先将hbase的jar包添加到hadoop的classpath路径下

export HBASE\_HOME=/export/servers/hbase-2.0.0/

export HADOOP\_HOME=/export/servers/hadoop-2.7.5/

export HADOOP\_CLASSPATH=`${HBASE\_HOME}/bin/hbase mapredcp`

然后执行以下命令，将hbase的HFile直接导入到表myuser2当中来

yarn jar /export/servers/hbase-2.0.0/lib/hbase-server-1.2.0-cdh5.14.0.jar completebulkload /hbase/hfile\_out myuser2

# 13、HBase与hive的对比

## Hive

### 数据仓库工具

Hive的本质其实就相当于将HDFS中已经存储的文件在Mysql中做了一个双射关系，以方便使用HQL去管理查询。

### 用于数据分析、清洗

Hive适用于离线的数据分析和清洗，延迟较高

### 基于HDFS、MapReduce

Hive存储的数据依旧在DataNode上，编写的HQL语句终将是转换为MapReduce代码执行。

## HBase

### nosql数据库

是一种面向列存储的非关系型数据库。

### 用于存储结构化和非结构话的数据

适用于单表非关系型数据的存储，不适合做关联查询，类似JOIN等操作。

### 基于HDFS

数据持久化存储的体现形式是Hfile，存放于DataNode中，被ResionServer以region的形式进行管理。

### 延迟较低，接入在线业务使用

面对大量的企业数据，HBase可以直线单表大量数据的存储，同时提供了高效的数据访问速度。

### 总结：Hive与HBase

Hive和Hbase是两种基于Hadoop的不同技术，Hive是一种类SQL的引擎，并且运行MapReduce任务，Hbase是一种在Hadoop之上的NoSQL 的Key/vale数据库。这两种工具是可以同时使用的。就像用Google来搜索，用FaceBook进行社交一样，Hive可以用来进行统计查询，HBase可以用来进行实时查询，数据也可以从Hive写到HBase，或者从HBase写回Hive。

# 14、hive与HBase的整合

hive与我们的HBase各有千秋，各自有着不同的功能，但是归根接地，hive与hbase的数据最终都是存储在hdfs上面的，一般的我们为了存储磁盘的空间，不会将一份数据存储到多个地方，导致磁盘空间的浪费，我们可以直接将数据存入hbase，然后通过hive整合hbase直接使用sql语句分析hbase里面的数据即可，非常方便

## 需求一：将hive分析结果的数据，保存到HBase当中去

### 第一步：拷贝hbase的五个依赖jar包到hive的lib目录下

将我们HBase的五个jar包拷贝到hive的lib目录下

hbase的jar包都在/export/servers/hbase-2.0.0/lib

我们需要拷贝五个jar包名字如下

hbase-client-2.0.0.jar

hbase-hadoop2-compat-2.0.0.jar

hbase-hadoop-compat-2.0.0.jar

hbase-it-2.0.0.jar

hbase-server-2.0.0.jar

我们直接在node03执行以下命令，通过创建软连接的方式来进行jar包的依赖

ln -s /export/servers/hbase-2.0.0/lib/hbase-client-2.0.0.jar /export/servers/apache-hive-2.1.0-bin/lib/hbase-client-2.0.0.jar

ln -s /export/servers/hbase-2.0.0/lib/hbase-hadoop2-compat-2.0.0.jar /export/servers/apache-hive-2.1.0-bin/lib/hbase-hadoop2-compat-2.0.0.jar

ln -s /export/servers/hbase-2.0.0/lib/hbase-hadoop-compat-2.0.0.jar /export/servers/apache-hive-2.1.0-bin/lib/hbase-hadoop-compat-2.0.0.jar

ln -s /export/servers/hbase-2.0.0/lib/hbase-it-2.0.0.jar /export/servers/apache-hive-2.1.0-bin/lib/hbase-it-2.0.0.jar

ln -s /export/servers/hbase-2.0.0/lib/hbase-server-2.0.0.jar /export/servers/apache-hive-2.1.0-bin/lib/hbase-server-2.0.0.jar

### 第二步：修改hive的配置文件

编辑node03服务器上面的hive的配置文件hive-site.xml添加以下两行配置

cd /export/servers/apache-hive-2.1.0-bin/conf

vim hive-site.xml

<property>

<name>hive.zookeeper.quorum</name>

<value>node01,node02,node03</value>

</property>

<property>

<name>hbase.zookeeper.quorum</name>

<value>node01,node02,node03</value>

</property>

### 第三步：修改hive-env.sh配置文件添加以下配置

cd /export/servers/apache-hive-2.1.0-bin/conf

vim hive-env.sh

export HADOOP\_HOME=/export/servers/hadoop-2.7.5

export HBASE\_HOME=/export/servers/hbase-2.0.0

export HIVE\_CONF\_DIR=/export/servers/apache-hive-2.1.0-bin/conf

### 第四步：hive当中建表并加载以下数据

#### hive当中建表

进入hive客户端

cd /export/servers/apache-hive-2.1.0-bin/

bin/hive

创建hive数据库与hive对应的数据库表

create database course;

use course;

create external table if not exists course.score(id int,cname string,score int) row format delimited fields terminated by '\t' stored as textfile ;

#### 准备数据内容如下

node03执行以下命令，准备数据文件

cd /export/

vim hive-hbase.txt

1 zhangsan 80

2 lisi 60

3 wangwu 30

4 zhaoliu 70

#### 进行加载数据

进入hive客户端进行加载数据

hive (course)> load data local inpath '/export/hive-hbase.txt' into table score;

hive (course)> select \* from score;

### 第五步：创建hive管理表与HBase进行映射

我们可以创建一个hive的管理表与hbase当中的表进行映射，hive管理表当中的数据，都会存储到hbase上面去

hive当中创建内部表

create table course.hbase\_score(id int,cname string,score int)

stored by 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'

with serdeproperties("hbase.columns.mapping" = "cf:name,cf:score")

tblproperties("hbase.table.name" = "hbase\_score");

通过insert overwrite select 插入数据

insert overwrite table course.hbase\_score select id,cname,score from course.score;

### 第六步：hbase当中查看表hbase\_score

进入hbase的客户端查看表hbase\_score，并查看当中的数据

hbase(main):023:0> list

TABLE

hbase\_score

myuser

myuser2

student

user

5 row(s) in 0.0210 seconds

=> ["hbase\_score", "myuser", "myuser2", "student", "user"]

hbase(main):024:0> scan 'hbase\_score'

ROW COLUMN+CELL

1 column=cf:name, timestamp=1550628395266, value=zhangsan

1 column=cf:score, timestamp=1550628395266, value=80

2 column=cf:name, timestamp=1550628395266, value=lisi

2 column=cf:score, timestamp=1550628395266, value=60

3 column=cf:name, timestamp=1550628395266, value=wangwu

3 column=cf:score, timestamp=1550628395266, value=30

4 column=cf:name, timestamp=1550628395266, value=zhaoliu

4 column=cf:score, timestamp=1550628395266, value=70

4 row(s) in 0.0360 seconds

## 需求二：创建hive外部表，映射HBase当中已有的表模型，

### 第一步：HBase当中创建表并手动插入加载一些数据

进入HBase的shell客户端，手动创建一张表，并插入加载一些数据进去

create 'hbase\_hive\_score',{ NAME =>'cf'}

put 'hbase\_hive\_score','1','cf:name','zhangsan'

put 'hbase\_hive\_score','1','cf:score', '95'

put 'hbase\_hive\_score','2','cf:name','lisi'

put 'hbase\_hive\_score','2','cf:score', '96'

put 'hbase\_hive\_score','3','cf:name','wangwu'

put 'hbase\_hive\_score','3','cf:score', '97'

操作成功结果如下：

hbase(main):049:0> create 'hbase\_hive\_score',{ NAME =>'cf'}

0 row(s) in 1.2970 seconds

=> Hbase::Table - hbase\_hive\_score

hbase(main):050:0> put 'hbase\_hive\_score','1','cf:name','zhangsan'

0 row(s) in 0.0600 seconds

hbase(main):051:0> put 'hbase\_hive\_score','1','cf:score', '95'

0 row(s) in 0.0310 seconds

hbase(main):052:0> put 'hbase\_hive\_score','2','cf:name','lisi'

0 row(s) in 0.0230 seconds

hbase(main):053:0> put 'hbase\_hive\_score','2','cf:score', '96'

0 row(s) in 0.0220 seconds

hbase(main):054:0> put 'hbase\_hive\_score','3','cf:name','wangwu'

0 row(s) in 0.0200 seconds

hbase(main):055:0> put 'hbase\_hive\_score','3','cf:score', '97'

0 row(s) in 0.0250 seconds

### 第二步：建立hive的外部表，映射HBase当中的表以及字段

在hive当中建立外部表，

进入hive客户端，然后执行以下命令进行创建hive外部表，就可以实现映射HBase当中的表数据

CREATE external TABLE course.hbase2hive(id int, name string, score int) STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler' WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,cf:name,cf:score") TBLPROPERTIES("hbase.table.name" ="hbase\_hive\_score");

# 15、HBase的预分区

## 1、为何要预分区？

\* 增加数据读写效率

\* 负载均衡，防止数据倾斜

\* 方便集群容灾调度region

\* 优化Map数量

## 2、如何预分区？

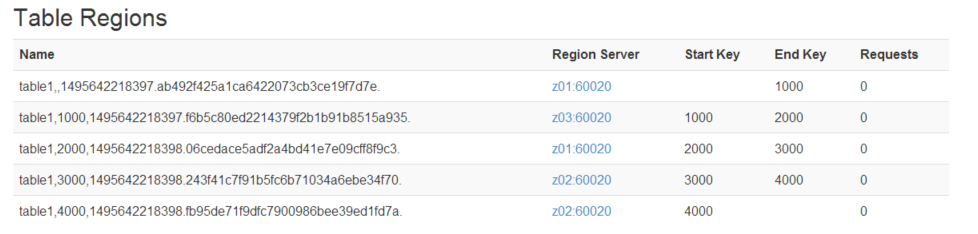
每一个region维护着startRow与endRowKey，如果加入的数据符合某个region维护的rowKey范围，则该数据交给这个region维护。

## 3、如何设定预分区？

### 1、手动指定预分区

hbase(main):001:0> create 'staff','info','partition1',SPLITS => ['1000','2000','3000','4000']

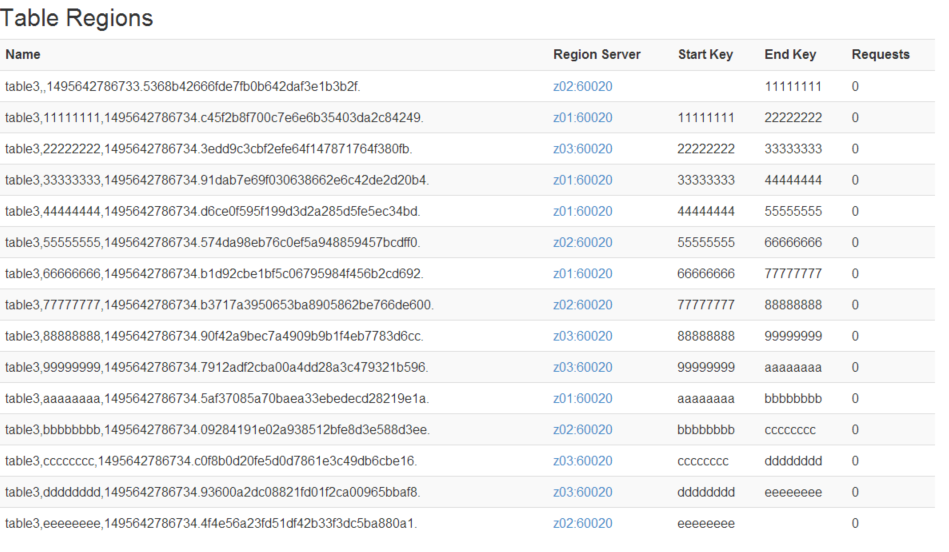
完成后如图：



### 2、使用16进制算法生成预分区

hbase(main):003:0> create 'staff2','info','partition2',{NUMREGIONS => 15, SPLITALGO => 'HexStringSplit'}

完成后如图：



### 3、使用JavaAPI创建预分区

Java代码如下：

/\*\*

\* 通过javaAPI进行HBase的表的创建以及预分区操作

\*/

@Test

public void hbaseSplit() throws IOException {

//获取连接

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum", "node01:2181,node02:2181,node03:2181");

Connection connection = ConnectionFactory.createConnection(configuration);

Admin admin = connection.getAdmin();

//自定义算法，产生一系列Hash散列值存储在二维数组中

byte[][] splitKeys = {{1,2,3,4,5},{'a','b','c','d','e'}};

//通过HTableDescriptor来实现我们表的参数设置，包括表名，列族等等

HTableDescriptor hTableDescriptor = new HTableDescriptor(TableName.valueOf("staff3"));

//添加列族

hTableDescriptor.addFamily(new HColumnDescriptor("f1"));

//添加列族

hTableDescriptor.addFamily(new HColumnDescriptor("f2"));

admin.createTable(hTableDescriptor,splitKeys);

admin.close();

}

# 16、HBase的rowKey设计技巧

HBase是三维有序存储的，通过rowkey（行键），column key（column family和qualifier）和TimeStamp（时间戳）这个三个维度可以对HBase中的数据进行快速定位。

HBase中rowkey可以唯一标识一行记录，在HBase查询的时候，有以下几种方式：

1. 通过get方式，指定rowkey获取唯一一条记录
2. 通过scan方式，设置startRow和stopRow参数进行范围匹配
3. 全表扫描，即直接扫描整张表中所有行记录

### 1 rowkey长度原则

rowkey是一个二进制码流，可以是任意字符串，最大长度64kb，实际应用中一般为10-100bytes，以byte[]形式保存，一般设计成定长。

建议越短越好，不要超过16个字节，原因如下：

* 数据的持久化文件HFile中是按照KeyValue存储的，如果rowkey过长，比如超过100字节，1000w行数据，光rowkey就要占用100\*1000w=10亿个字节，将近1G数据，这样会极大影响HFile的存储效率；
* MemStore将缓存部分数据到内存，如果rowkey字段过长，内存的有效利用率就会降低，系统不能缓存更多的数据，这样会降低检索效率。

### 2 rowkey散列原则

如果rowkey按照时间戳的方式递增，不要将时间放在二进制码的前面，建议将rowkey的高位作为散列字段，由程序随机生成，低位放时间字段，这样将提高数据均衡分布在每个RegionServer，以实现负载均衡的几率。如果没有散列字段，首字段直接是时间信息，所有的数据都会集中在一个RegionServer上，这样在数据检索的时候负载会集中在个别的RegionServer上，造成热点问题，会降低查询效率。

### 3 rowkey唯一原则

必须在设计上保证其唯一性，rowkey是按照字典顺序排序存储的，因此，设计rowkey的时候，要充分利用这个排序的特点，将经常读取的数据存储到一块，将最近可能会被访问的数据放到一块。

### 4什么是热点

HBase中的行是按照rowkey的字典顺序排序的，这种设计优化了scan操作，可以将相关的行以及会被一起读取的行存取在临近位置，便于scan。然而糟糕的rowkey设计是热点的源头。

热点发生在大量的client直接访问集群的一个或极少数个节点（访问可能是读，写或者其他操作）。大量访问会使热点region所在的单个机器超出自身承受能力，引起性能下降甚至region不可用，这也会影响同一个RegionServer上的其他region，由于主机无法服务其他region的请求。

设计良好的数据访问模式以使集群被充分，均衡的利用。为了避免写热点，设计rowkey使得不同行在同一个region，但是在更多数据情况下，数据应该被写入集群的多个region，而不是一个。下面是一些常见的避免热点的方法以及它们的优缺点：

#### 1加盐

这里所说的加盐不是密码学中的加盐，而是在rowkey的前面增加随机数，具体就是给rowkey分配一个随机前缀以使得它和之前的rowkey的开头不同。分配的前缀种类数量应该和你想使用数据分散到不同的region的数量一致。加盐之后的rowkey就会根据随机生成的前缀分散到各个region上，以避免热点。

#### 2哈希

哈希会使同一行永远用一个前缀加盐。哈希也可以使负载分散到整个集群，但是读却是可以预测的。使用确定的哈希可以让客户端重构完整的rowkey，可以使用get操作准确获取某一个行数据。

#### 3反转

第三种防止热点的方法时反转固定长度或者数字格式的rowkey。这样可以使得rowkey中经常改变的部分（最没有意义的部分）放在前面。这样可以有效的随机rowkey，但是牺牲了rowkey的有序性。

反转rowkey的例子以手机号为rowkey，可以将手机号反转后的字符串作为rowkey，这样的就避免了以手机号那样比较固定开头导致热点问题

#### 3时间戳反转

一个常见的数据处理问题是快速获取数据的最近版本，使用反转的时间戳作为rowkey的一部分对这个问题十分有用，可以用 Long.Max\_Value - timestamp 追加到key的末尾，例如 [key][reverse\_timestamp] , [key] 的最新值可以通过scan [key]获得[key]的第一条记录，因为HBase中rowkey是有序的，第一条记录是最后录入的数据。

其他一些建议：

尽量减少行键和列族的大小在HBase中，value永远和它的key一起传输的。当具体的值在系统间传输时，它的rowkey，列名，时间戳也会一起传输。如果你的rowkey和列名很大，这个时候它们将会占用大量的存储空间。

列族尽可能越短越好，最好是一个字符。

冗长的属性名虽然可读性好，但是更短的属性名存储在HBase中会更好。

# 17、HBase的协处理器

<http://hbase.apache.org/book.html#cp>

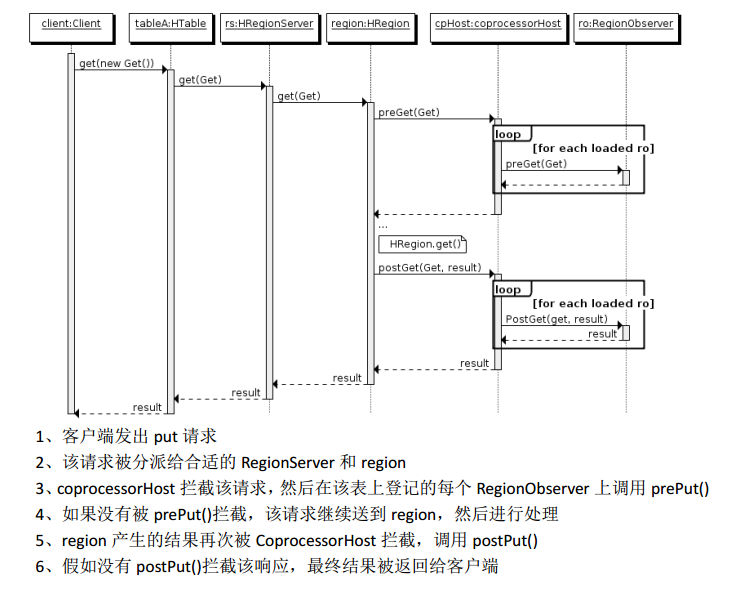
## 1、 起源 Hbase 作为列族数据库最经常被人诟病的特性包括：无法轻易建立“二级索引”，难以执 行求和、计数、排序等操作。比如，在旧版本的(<0.92)Hbase 中，统计数据表的总行数，需 要使用 Counter 方法，执行一次 MapReduce Job 才能得到。虽然 HBase 在数据存储层中集成 了 MapReduce，能够有效用于数据表的分布式计算。然而在很多情况下，做一些简单的相 加或者聚合计算的时候， 如果直接将计算过程放置在 server 端，能够减少通讯开销，从而获 得很好的性能提升。于是， HBase 在 0.92 之后引入了协处理器(coprocessors)，实现一些激动 人心的新特性：能够轻易建立二次索引、复杂过滤器(谓词下推)以及访问控制等。

## 2、协处理器有两种： observer 和 endpoint

  (1) Observer 类似于传统数据库中的触发器，当发生某些事件的时候这类协处理器会被 Server 端调用。Observer Coprocessor 就是一些散布在 HBase Server 端代码中的 hook 钩子， 在固定的事件发生时被调用。比如： put 操作之前有钩子函数 prePut，该函数在 put 操作  
执行前会被 Region Server 调用；在 put 操作之后则有 postPut 钩子函数

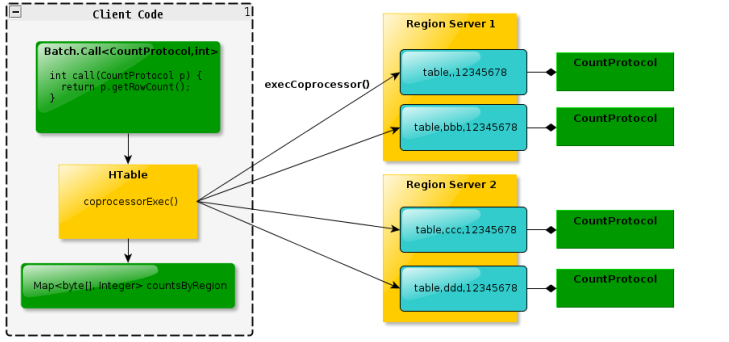
以 Hbase2.0.0 版本为例，它提供了三种观察者接口：  
● RegionObserver：提供客户端的数据操纵事件钩子： Get、 Put、 Delete、 Scan 等。  
● WALObserver：提供 WAL 相关操作钩子。  
● MasterObserver：提供 DDL-类型的操作钩子。如创建、删除、修改数据表等。  
到 0.96 版本又新增一个 RegionServerObserver

下图是以 RegionObserver 为例子讲解 Observer 这种协处理器的原理：



   (2) Endpoint 协处理器类似传统数据库中的存储过程，客户端可以调用这些 Endpoint 协处 理器执行一段 Server 端代码，并将 Server 端代码的结果返回给客户端进一步处理，最常 见的用法就是进行聚集操作。如果没有协处理器，当用户需要找出一张表中的最大数据，即

max 聚合操作，就必须进行全表扫描，在客户端代码内遍历扫描结果，并执行求最大值的 操作。这样的方法无法利用底层集群的并发能力，而将所有计算都集中到 Client 端统一执 行，势必效率低下。利用 Coprocessor，用户可以将求最大值的代码部署到 HBase Server 端，  
HBase 将利用底层 cluster 的多个节点并发执行求最大值的操作。即在每个 Region 范围内 执行求最大值的代码，将每个 Region 的最大值在 Region Server 端计算出，仅仅将该 max 值返回给客户端。在客户端进一步将多个 Region 的最大值进一步处理而找到其中的最大值。  
这样整体的执行效率就会提高很多  
下图是 EndPoint 的工作原理：



  (3)总结

Observer 允许集群在正常的客户端操作过程中可以有不同的行为表现  
Endpoint 允许扩展集群的能力，对客户端应用开放新的运算命令  
observer 类似于 RDBMS 中的触发器，主要在服务端工作  
endpoint 类似于 RDBMS 中的存储过程，主要在 client 端工作  
observer 可以实现权限管理、优先级设置、监控、 ddl 控制、 二级索引等功能  
endpoint 可以实现 min、 max、 avg、 sum、 distinct、 group by 等功能

## 3、协处理器加载方式

     协处理器的加载方式有两种，我们称之为静态加载方式（ Static Load） 和动态加载方式 （ Dynamic Load）。 静态加载的协处理器称之为 System Coprocessor，动态加载的协处理器称 之为 Table Coprocessor  
     1、静态加载

通过修改 hbase-site.xml 这个文件来实现， 启动全局 aggregation，能过操纵所有的表上 的数据。只需要添加如下代码：

<property>

<name>hbase.coprocessor.user.region.classes</name>

<value>org.apache.hadoop.hbase.coprocessor.AggregateImplementation</value>

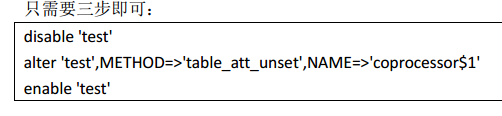
</property>

　　为所有 table 加载了一个 cp class，可以用” ,”分割加载多个 class

    2、动态加载

启用表 aggregation，只对特定的表生效。通过 HBase Shell 来实现。  
disable 指定表。 hbase> disable 'mytable'  
添加 aggregation  
hbase> alter 'mytable', METHOD => 'table\_att','coprocessor'=>  
'|org.apache.Hadoop.hbase.coprocessor.AggregateImplementation||'  
重启指定表 hbase> enable 'mytable'

协处理器卸载



## 4、协处理器Observer应用实战

通过协处理器Observer实现hbase当中一张表插入数据，然后通过协处理器，将数据复制一份保存到另外一张表当中去，但是只取当第一张表当中的部分列数据保存到第二张表当中去

### 第一步：HBase当中创建第一张表proc1

在HBase当中创建一张表，表名user2，并只有一个列族info

cd /export/servers/hbase-2.0.0/

bin/hbase shell

hbase(main):053:0> create 'proc1','info'

### 第二步：Hbase当中创建第二张表proc2

创建第二张表'proc2，作为目标表，将第一张表当中插入数据的部分列，使用协处理器，复制到'proc2表当中来

hbase(main):054:0> create 'proc2','info'

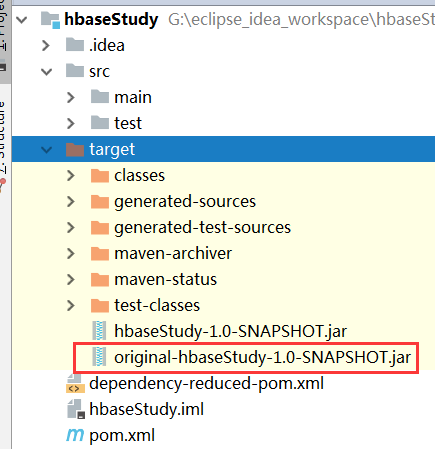
### 第三步：开发HBase的协处理器

开发HBase的协处理器Copo

**public class** MyProcessor **implements** RegionObserver,RegionCoprocessor {  
  
 **static** Connection *connection* = **null**;  
 **static** Table *table* = **null**;  
 **static**{  
 Configuration conf = HBaseConfiguration.*create*();  
 conf.set(**"hbase.zookeeper.quorum"**,**"node01:2181"**);  
 **try** {  
 *connection* = ConnectionFactory.*createConnection*(conf);  
 *table* = *connection*.getTable(TableName.*valueOf*(**"proc2"**));  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 **private** RegionCoprocessorEnvironment **env** = **null**;  
 **private static final** String ***FAMAILLY\_NAME*** = **"info"**;  
 **private static final** String ***QUALIFIER\_NAME*** = **"name"**;  
 *//2.0加入该方法，否则无法生效* @Override  
 **public** Optional<RegionObserver> getRegionObserver() {  
 *// Extremely important to be sure that the coprocessor is invoked as a RegionObserver* **return** Optional.*of*(**this**);  
 }  
 @Override  
 **public void** start(CoprocessorEnvironment e) **throws** IOException {  
 **env** = (RegionCoprocessorEnvironment) e;  
 }  
 @Override  
 **public void** stop(CoprocessorEnvironment e) **throws** IOException {  
 *// nothing to do here* }  
 */\*\*  
 \* 覆写prePut方法，在我们数据插入之前进行拦截，  
 \** ***@param e*** *\** ***@param put*** *put对象里面封装了我们需要插入到目标表的数据  
 \** ***@param edit*** *\** ***@param durability*** *\** ***@throws*** *IOException  
 \*/* @Override  
 **public void** prePut(**final** ObserverContext<RegionCoprocessorEnvironment> e,  
 **final** Put put, **final** WALEdit edit, **final** Durability durability)  
 **throws** IOException {  
 **try** {  
 *//通过put对象获取插入数据的rowkey* **byte**[] rowBytes = put.getRow();  
 String rowkey = Bytes.*toString*(rowBytes);  
 *//获取我们插入数据的name字段的值* List<Cell> list = put.get(Bytes.*toBytes*(***FAMAILLY\_NAME***), Bytes.*toBytes*(***QUALIFIER\_NAME***));  
 **if** (list == **null** || list.size() == 0) {  
 **return**;  
 }  
 *//获取到info列族，name列对应的cell* Cell cell2 = list.get(0);  
  
 *//通过cell获取数据值* String nameValue = Bytes.*toString*(CellUtil.*cloneValue*(cell2));  
 *//创建put对象，将数据插入到proc2表里面去* Put put2 = **new** Put(rowkey.getBytes());  
 put2.addColumn(Bytes.*toBytes*(***FAMAILLY\_NAME***), Bytes.*toBytes*(***QUALIFIER\_NAME***), nameValue.getBytes());  
 *table*.put(put2);  
 *table*.close();  
 } **catch** (Exception e1) {  
 **return** ;  
 }  
 }  
}

### 第四步：将项目打成jar包，并上传到HDFS上面

将我们的协处理器打成一个jar包，此处不需要用任何的打包插件即可，然后上传到hdfs



将打好的jar包上传到linux的/export/servers路径下

cd /export/servers

mv original-hbase-1.0-SNAPSHOT.jar processor.jar

hdfs dfs -mkdir -p /processor

hdfs dfs -put processor.jar /processor

### 第五步：将打好的jar包挂载到proc1表当中去

hbase(main):056:0> **describe 'proc1'**

hbase(main):055:0> **alter 'proc1',METHOD => 'table\_att','Coprocessor'=>'hdfs://node01:8020/processor/processor.jar|cn.itcast.hbasemr.demo4.MyProcessor|1001|'**

再次查看'proc1'表，

hbase(main):043:0> describe 'proc1'

可以查看到我们的卸载器已经加载了

### 第六步：proc1表当中添加数据

进入hbase-shell客户端，然后直接执行以下命令向proc1表当中添加数据

put 'proc1','0001','info:name','zhangsan'

put 'proc1','0001','info:age','28'

put 'proc1','0002','info:name','lisi'

put 'proc1','0002','info:age','25'

向proc1表当中添加数据，然后通过

scan 'proc2'

我们会发现，proc2表当中也插入了数据，并且只有info列族，name列

注意：如果需要卸载我们的协处理器，那么进入hbase的shell命令行，执行以下命令即可

disable 'proc1'

alter 'proc1',METHOD=>'table\_att\_unset',NAME=>'coprocessor$1'

enable 'proc1'

# 18、HBase当中的二级索引的基本介绍

由于HBase的查询比较弱，如果需要实现类似于 select name,salary,count(1),max(salary) from user group by name,salary order by salary 等这样的复杂性的统计需求，基本上不可能，或者说比较困难，所以我们在使用HBase的时候，一般都会借助二级索引的方案来进行实现

HBase的一级索引就是rowkey，我们只能通过rowkey进行检索。如果我们相对hbase里面列族的列列进行一些组合查询，就需要采用HBase的二级索引方案来进行多条件的查询。

1. MapReduce方案

2. ITHBASE（Indexed-Transanctional HBase）方案

3. IHBASE（Index HBase）方案

4. Hbase Coprocessor(协处理器)方案

5. Solr+hbase方案

6. CCIndex（complementalclustering index）方案

常见的二级索引我们一般可以借助各种其他的方式来实现，例如Phoenix或者solr或者ES等

# 19、HBase整合hue

## 1、Hue的介绍

HUE=**Hadoop User Experience**

**在没有HUE的时候，如果我们想要查看Hadoop生态圈各组件的状态，可以通过它们的webconsole，地址分别是：**

**HDFS: NameNode网页 http://ip:50070**

**SecondaryNameNode网页: http://ip:50090**

**Yarn: http://ip:8088**

**HBase: http://ip:16010**

**Hive http://ip:9999/hwi/**

**Spark http://ip:8080**

**如果一个个去查看肯定是可以的，但是…比较耗时间，稍显麻烦。而HUE就是对这些的整合，在HUE一个地方就可以查看上面全部组件的状况和进行一些操作。**

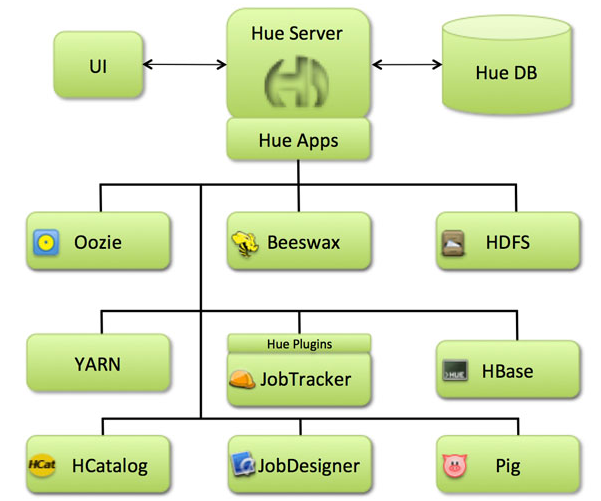
Hue是一个开源的Apache Hadoop UI系统，由Cloudera Desktop演化而来，最后Cloudera公司将其贡献给Apache基金会的Hadoop社区，它是基于Python Web框架Django实现的。

通过使用Hue我们可以在浏览器端的Web控制台上与Hadoop集群进行交互来分析处理数据，例如操作HDFS上的数据，运行MapReduce Job，执行Hive的SQL语句，浏览HBase数据库等等。

### HUE链接

* Site: <http://gethue.com/>
* Github: <https://github.com/cloudera/hue>
* Reviews: [https://review.cloudera.org](https://review.cloudera.org/)

### Hue的架构



### 核心功能

* SQL编辑器，支持Hive, Impala, MySQL, Oracle, PostgreSQL, SparkSQL, Solr SQL, Phoenix…
* 搜索引擎Solr的各种图表
* Spark和Hadoop的友好界面支持
* 支持调度系统Apache Oozie，可进行workflow的编辑、查看

HUE提供的这些功能相比Hadoop生态各组件提供的界面更加友好，但是一些需要debug的场景可能还是需要使用原生系统才能更加深入的找到错误的原因。

HUE中查看Oozie workflow时，也可以很方便的看到整个workflow的DAG图，不过在最新版本中已经将DAG图去掉了，只能看到workflow中的action列表和他们之间的跳转关系，想要看DAG图的仍然可以使用oozie原生的界面系统查看。

1，访问HDFS和文件浏览

2，通过web调试和开发hive以及数据结果展示

3，查询solr和结果展示，报表生成

4，通过web调试和开发impala交互式SQL Query

5，spark调试和开发

7，oozie任务的开发，监控，和工作流协调调度

8，Hbase数据查询和修改，数据展示

9，Hive的元数据（metastore）查询

10，MapReduce任务进度查看，日志追踪

11，创建和提交MapReduce，Streaming，Java job任务

12，Sqoop2的开发和调试

13，Zookeeper的浏览和编辑

14，数据库（MySQL，PostGres，SQlite，Oracle）的查询和展示

一句话总结：Hue是一个友好的界面集成框架，可以集成我们各种学习过的以及将要学习的框架，一个界面就可以做到查看以及执行所有的框架

## 2、Hue的环境准备及安装

Hue的安装支持多种方式，包括rpm包的方式进行安装，tar.gz包的方式进行安装以及cloudera manager的方式来进行安装等，我们这里使用tar.gz包的方式来进行安装

### 第一步：下载依赖包

node03服务器执行一下命令联网下载依赖包

yum install ant asciidoc cyrus-sasl-devel cyrus-sasl-gssapi cyrus-sasl-plain gcc gcc-c++ krb5-devel libffi-devel libxml2-devel libxslt-devel make mysql mysql-devel openldap-devel python-devel sqlite-devel gmp-devel openssl-devel -y

### 第二步：安装配置maven

hue的编译需要使用maven进行下载一些其他jar包，我们这里给node03服务器安装maven即可

node03执行以下命令安装maven

wget http://repos.fedorapeople.org/repos/dchen/apache-maven/epel-apache-maven.repo -O /etc/yum.repos.d/epel-apache-maven.repo

sed -i s/\$releasever/6/g /etc/yum.repos.d/epel-apache-maven.repo

yum install -y apache-maven

mvn --version

配置maven的下载地址

vim /etc/maven/settings.xml

 <mirror>

         <id>alimaven</id>

         <mirrorOf>central</mirrorOf>

         <name>aliyun maven</name>

         <url>http://maven.aliyun.com/nexus/content/groups/public/</url>

     </mirror>

     <mirror>

         <id>ui</id>

         <mirrorOf>central</mirrorOf>

         <name>Human Readable Name for this Mirror.</name>

         <url>http://uk.maven.org/maven2/</url>

     </mirror>

     <mirror>

         <id>jboss-public-repository-group</id>

         <mirrorOf>central</mirrorOf>

         <name>JBoss Public Repository Group</name>

         <url>http://repository.jboss.org/nexus/content/groups/public</url>

     </mirror>

### 第三步：为linux操作系统添加普通用户

hue的安装必须添加普通用户 hue，否则启动报错，直接给node03服务器添加普通用户即可

useradd hue

passwd hue

### 第四步：下载hue压缩包并上传解压

node03服务器执行以下命令下载并解压安装包

cd /export/softwares

wget <http://gethue.com/downloads/releases/4.0.1/hue-4.0.1.tgz>

tar -zxf hue-4.0.1.tgz -C /export/servers/

### 第五步：修改配置文件

修改hue的配置文件hue.ini

node03执行以下命令修改hue的配置文件

cd /export/servers/hue-4.0.0/desktop/conf/

vim hue.ini

#通用配置

[desktop]

secret\_key=jFE93j;2[290-eiw.KEiwN2s3['d;/.q[eIW^y#e=+Iei\*@Mn<qW5o

http\_host=node03.hadoop.com

time\_zone=Asia/Shanghai

server\_user=root

server\_group=root

default\_user=root

default\_hdfs\_superuser=root

#配置使用mysql作为hue的存储数据库,大概在hue.ini的561行左右

[[database]]

engine=mysql

host=node03.hadoop.com

port=3306

user=root

password=123456

name=hue

### 第六步：创建mysql数据库

进入mysql客户端，然后创建mysql数据库

mysql –uroot -p

创建hue数据库

create database hue default character set utf8 default collate utf8\_general\_ci;

### 第七步：对hue进行编译

node03执行以下命令进行编译

cd /export/servers/hue-3.9.0-cdh5.14.0

make apps

注意：如果编译失败，那么需要重新进行编译，多编译几次，网速够快就能编译通过

make clean

make apps

### 第八步：启动hue服务并进行页面访问

node03执行以下命令启动服务

cd /export/servers/hue-3.9.0-cdh5.14.0

build/env/bin/supervisor

页面访问

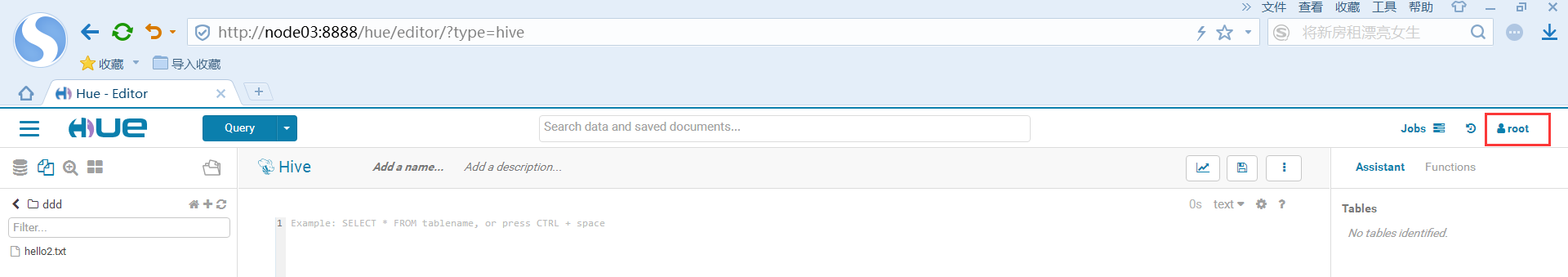
<http://node03:8888>

第一次访问的时候，需要设置管理员用户和密码

我们这里的管理员的用户名与密码尽量保持与我们安装hadoop的用户名和密码一致，

我们安装hadoop的用户名与密码分别是root 123456

初次登录使用root用户，密码为123456



进入之后发现我们的hue页面报错了，这个错误主要是因为hive的原因，因为我们的hue与hive集成的时候出错了，所以我们需要配置我们的hue与hive进行集成，接下里就看看我们的hue与hive以及hadoop如何进行集成

## 3、hue与其他框架的集成

### 3.1、hue与hadoop的HDFS以及yarn集成

#### 第一步：更改所有hadoop节点的core-site.xml配置

记得更改完core-site.xml之后一定要重启hdfs与yarn集群

三台机器更改core-site.xml

<property>

<name>hadoop.proxyuser.root.hosts</name>

<value>\*</value>

</property>

<property>

<name>hadoop.proxyuser.root.groups</name>

<value>\*</value>

</property>

#### 第二步：更改所有hadoop节点的hdfs-site.xml

<property>

<name>dfs.webhdfs.enabled</name>

<value>true</value>

</property>

#### 第三步：重启hadoop集群

在node01机器上面执行以下命令

cd /export/servers/hadoop-2.7.5

sbin/stop-dfs.sh

sbin/start-dfs.sh

sbin/stop-yarn.sh

sbin/start-yarn.sh

#### 第四步：停止hue的服务，并继续配置hue.ini

cd /export/servers/hue-3.9.0-cdh5.14.0/desktop/conf

vim hue.ini

配置我们的hue与hdfs集成

[[hdfs\_clusters]]

[[[default]]]

fs\_defaultfs=hdfs://node01.hadoop.com:8020

webhdfs\_url=http://node01.hadoop.com:50070/webhdfs/v1

hadoop\_hdfs\_home=/export/servers/hadoop-2.7.5

hadoop\_bin=/export/servers/hadoop-2.7.5/bin

hadoop\_conf\_dir=/export/servers/hadoop-2.7.5/etc/hadoop

配置我们的hue与yarn集成

[[yarn\_clusters]]

[[[default]]]

resourcemanager\_host=node01

resourcemanager\_port=8032

submit\_to=True

resourcemanager\_api\_url=http://node01:8088

history\_server\_api\_url=http://node01:19888

### 3.2、配置hue与hive集成

如果需要配置hue与hive的集成，我们需要启动hive的hiveserver2服务

#### 更改hue的配置hue.ini

修改hue.ini

[beeswax]

hive\_server\_host=node03.hadoop.com

hive\_server\_port=10000

hive\_conf\_dir=/export/servers/apache-hive-2.1.0-bin/conf

server\_conn\_timeout=120

auth\_username=root

auth\_password=123456

[metastore]

#允许使用hive创建数据库表等操作

enable\_new\_create\_table=true

#### 启动hive的metastore服务

去node03机器上启动hive的hiveserver2服务

cd /export/servers/apache-hive-2.1.0-bin

nohup bin/hive --service hiveserver2 &

重新启动hue，然后就可以通过浏览器页面操作hive了

### 3.5、hue与HBase的集成

#### 第一步：修改hue.ini

cd /export/servers/hue-3.9.0-cdh5.14.0/desktop/conf

vim hue.ini

[hbase]

hbase\_clusters=(Cluster|node01:9090)

hbase\_conf\_dir=/export/servers/hbase-2.0.0/conf

#### 第二步：启动hbase的thrift server服务

第一台机器执行以下命令启动hbase的thriftserver

cd /export/servers/hbase-2.0.0

bin/hbase-daemon.sh start thrift

#### 第三步：启动hue

第三台机器执行以下命令启动hue

cd /export/servers/hue-3.9.0-cdh5.14.0/

build/env/bin/supervisor

#### 第四步：页面访问

<http://node03:8888/hue/>

# 20、HBase调优

## 1、通用优化

### 1、NameNode的元数据备份使用SSD

### 2、定时备份NameNode上的元数据，每小时或者每天备份，如果数据极其重要，可以5~10分钟备份一次。备份可以通过定时任务复制元数据目录即可。

### 3、为NameNode指定多个元数据目录，使用dfs.name.dir或者dfs.namenode.name.dir指定。一个指定本地磁盘，一个指定网络磁盘。这样可以提供元数据的冗余和健壮性，以免发生故障。

### 4、设置dfs.namenode.name.dir.restore为true，允许尝试恢复之前失败的dfs.namenode.name.dir目录，在创建checkpoint时做此尝试，如果设置了多个磁盘，建议允许。

### 5、NameNode节点必须配置为RAID1（镜像盘）结构。

### 6、补充：什么是Raid0、Raid0+1、Raid1、Raid5



**Standalone**

最普遍的单磁盘储存方式。

**Cluster**

集群储存是通过将数据分布到集群中各节点的存储方式,提供单一的使用接口与界面,使用户可以方便地对所有数据进行统一使用与管理。

**Hot swap**

用户可以再不关闭系统,不切断电源的情况下取出和更换硬盘,提高系统的恢复能力、拓展性和灵活性。

**Raid0**

Raid0是所有raid中存储性能最强的阵列形式。其工作原理就是在多个磁盘上分散存取连续的数据,这样,当需要存取数据是多个磁盘可以并排执行,每个磁盘执行属于它自己的那部分数据请求,显著提高磁盘整体存取性能。但是不具备容错能力,适用于低成本、低可靠性的台式系统。

**Raid1**

又称镜像盘,把一个磁盘的数据镜像到另一个磁盘上,采用镜像容错来提高可靠性,具有raid中最高的数据冗余能力。存数据时会将数据同时写入镜像盘内,读取数据则只从工作盘读出。发生故障时,系统将从镜像盘读取数据,然后再恢复工作盘正确数据。这种阵列方式可靠性极高,但是其容量会减去一半。广泛用于数据要求极严的应用场合,如商业金融、档案管理等领域。只允许一颗硬盘出故障。

**Raid0+1**

将Raid0和Raid1技术结合在一起,兼顾两者的优势。在数据得到保障的同时,还能提供较强的存储性能。不过至少要求4个或以上的硬盘，但也只允许一个磁盘出错。是一种三高技术。

**Raid5**

Raid5可以看成是Raid0+1的低成本方案。采用循环偶校验独立存取的阵列方式。将数据和相对应的奇偶校验信息分布存储到组成RAID5的各个磁盘上。当其中一个磁盘数据发生损坏后,利用剩下的磁盘和相应的奇偶校验信息 重新恢复/生成丢失的数据而不影响数据的可用性。至少需要3个或以上的硬盘。适用于大数据量的操作。成本稍高、储存性强、可靠性强的阵列方式。

RAID还有其他方式，请自行查阅。

### 7、保持NameNode日志目录有足够的空间，这些日志有助于帮助你发现问题。

### 8、因为Hadoop是IO密集型框架，所以尽量提升存储的速度和吞吐量（类似位宽）。

## 2、Linux优化

### 1、开启文件系统的预读缓存可以提高读取速度

$ sudo blockdev --setra 32768 /dev/sda

（尖叫提示：ra是readahead的缩写）

### 2、关闭进程睡眠池

$ sudo sysctl -w vm.swappiness=0

### 3、调整ulimit上限，默认值为比较小的数字

$ ulimit -n 查看允许最大进程数

$ ulimit -u 查看允许打开最大文件数

修改：

$ sudo vi /etc/security/limits.conf 修改打开文件数限制

末尾添加：

\* soft nofile 1024000

\* hard nofile 1024000

Hive - nofile 1024000

hive - nproc 1024000

$ sudo vi /etc/security/limits.d/20-nproc.conf 修改用户打开进程数限制

修改为：

#\* soft nproc 4096

#root soft nproc unlimited

\* soft nproc 40960

root soft nproc unlimited

### 4、开启集群的时间同步NTP，请参看之前文档

### 5、更新系统补丁（尖叫提示：更新补丁前，请先测试新版本补丁对集群节点的兼容性）

## 3、HDFS优化（hdfs-site.xml）

### 1、保证RPC调用会有较多的线程数

属性：dfs.namenode.handler.count

解释：该属性是NameNode服务默认线程数，的默认值是10，根据机器的可用内存可以调整为50~100

属性：dfs.datanode.handler.count

解释：该属性默认值为10，是DataNode的处理线程数，如果HDFS客户端程序读写请求比较多，可以调高到15~20，设置的值越大，内存消耗越多，不要调整的过高，一般业务中，5~10即可。

### 2、副本数的调整

属性：dfs.replication

解释：如果数据量巨大，且不是非常之重要，可以调整为2~3，如果数据非常之重要，可以调整为3~5。

### 3.、文件块大小的调整

属性：dfs.blocksize

解释：块大小定义，该属性应该根据存储的大量的单个文件大小来设置，如果大量的单个文件都小于100M，建议设置成64M块大小，对于大于100M或者达到GB的这种情况，建议设置成256M，一般设置范围波动在64M~256M之间。

## 4、MapReduce优化（mapred-site.xml）

### 1、Job任务服务线程数调整

mapreduce.jobtracker.handler.count

该属性是Job任务线程数，默认值是10，根据机器的可用内存可以调整为50~100

### 2、Http服务器工作线程数

属性：mapreduce.tasktracker.http.threads

解释：定义HTTP服务器工作线程数，默认值为40，对于大集群可以调整到80~100

### 3、文件排序合并优化

属性：mapreduce.task.io.sort.factor

解释：文件排序时同时合并的数据流的数量，这也定义了同时打开文件的个数，默认值为10，如果调高该参数，可以明显减少磁盘IO，即减少文件读取的次数。

### 4、设置任务并发

属性：mapreduce.map.speculative

解释：该属性可以设置任务是否可以并发执行，如果任务多而小，该属性设置为true可以明显加快任务执行效率，但是对于延迟非常高的任务，建议改为false，这就类似于迅雷下载。

### 5、MR输出数据的压缩

属性：mapreduce.map.output.compress、mapreduce.output.fileoutputformat.compress

解释：对于大集群而言，建议设置Map-Reduce的输出为压缩的数据，而对于小集群，则不需要。

### 6、优化Mapper和Reducer的个数

属性：

mapreduce.tasktracker.map.tasks.maximum

mapreduce.tasktracker.reduce.tasks.maximum

解释：以上两个属性分别为一个单独的Job任务可以同时运行的Map和Reduce的数量。

设置上面两个参数时，需要考虑CPU核数、磁盘和内存容量。假设一个8核的CPU，业务内容非常消耗CPU，那么可以设置map数量为4，如果该业务不是特别消耗CPU类型的，那么可以设置map数量为40，reduce数量为20。这些参数的值修改完成之后，一定要观察是否有较长等待的任务，如果有的话，可以减少数量以加快任务执行，如果设置一个很大的值，会引起大量的上下文切换，以及内存与磁盘之间的数据交换，这里没有标准的配置数值，需要根据业务和硬件配置以及经验来做出选择。

在同一时刻，不要同时运行太多的MapReduce，这样会消耗过多的内存，任务会执行的非常缓慢，我们需要根据CPU核数，内存容量设置一个MR任务并发的最大值，使固定数据量的任务完全加载到内存中，避免频繁的内存和磁盘数据交换，从而降低磁盘IO，提高性能。

大概配比：

|  |  |  |  |
| --- | --- | --- | --- |
| CPU CORE | MEM（GB） | Map | Reduce |
| 1 | 1 | 1 | 1 |
| 1 | 5 | 1 | 1 |
| 4 | 5 | 1~4 | 2 |
| 16 | 32 | 16 | 8 |
| 16 | 64 | 16 | 8 |
| 24 | 64 | 24 | 12 |
| 24 | 128 | 24 | 12 |

大概估算公式：

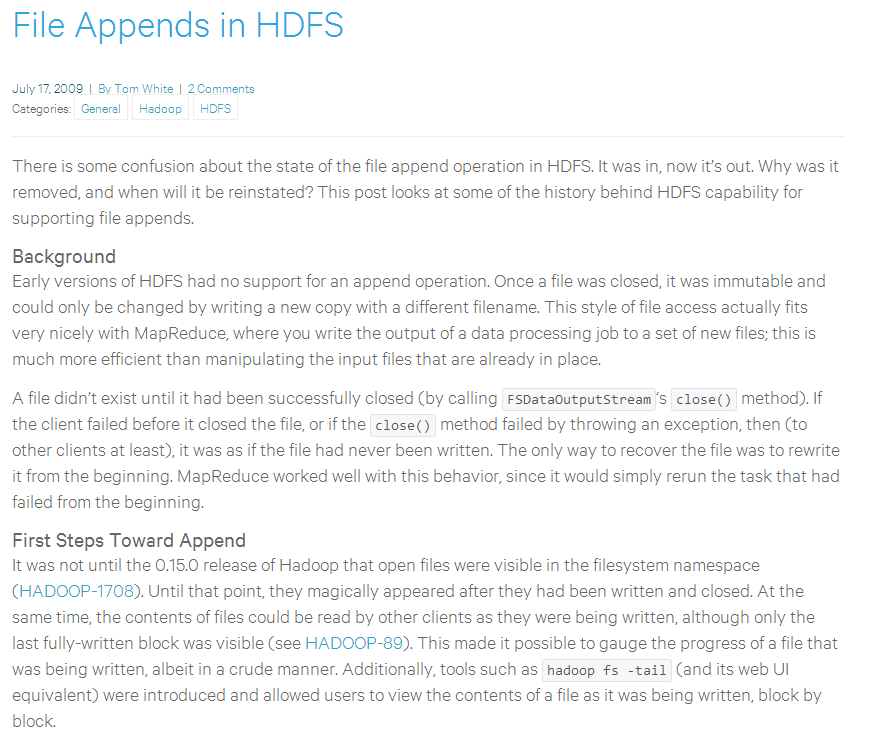
map = 2 + ⅔cpu\_core

reduce = 2 + ⅓cpu\_core

## 5、HBase优化

### 1、在HDFS的文件中追加内容

不是不允许追加内容么？没错，请看背景故事：



属性：dfs.support.append

文件：hdfs-site.xml、hbase-site.xml

解释：开启HDFS追加同步，可以优秀的配合HBase的数据同步和持久化。默认值为true。

### 2、优化DataNode允许的最大文件打开数

属性：dfs.datanode.max.transfer.threads

文件：hdfs-site.xml

解释：HBase一般都会同一时间操作大量的文件，根据集群的数量和规模以及数据动作，设置为4096或者更高。默认值：4096

### 3、优化延迟高的数据操作的等待时间

属性：dfs.image.transfer.timeout

文件：hdfs-site.xml

解释：如果对于某一次数据操作来讲，延迟非常高，socket需要等待更长的时间，建议把该值设置为更大的值（默认60000毫秒），以确保socket不会被timeout掉。

### 4、优化数据的写入效率

属性：

mapreduce.map.output.compress

mapreduce.map.output.compress.codec

文件：mapred-site.xml

解释：开启这两个数据可以大大提高文件的写入效率，减少写入时间。第一个属性值修改为true，第二个属性值修改为：org.apache.hadoop.io.compress.GzipCodec

### 5、优化DataNode存储

属性：dfs.datanode.failed.volumes.tolerated

文件：hdfs-site.xml

解释：默认为0，意思是当DataNode中有一个磁盘出现故障，则会认为该DataNode shutdown了。如果修改为1，则一个磁盘出现故障时，数据会被复制到其他正常的DataNode上，当前的DataNode继续工作。

### 6、设置RPC监听数量

属性：hbase.regionserver.handler.count

文件：hbase-site.xml

解释：默认值为30，用于指定RPC监听的数量，可以根据客户端的请求数进行调整，读写请求较多时，增加此值。

### 7、优化HStore文件大小

属性：hbase.hregion.max.filesize

文件：hbase-site.xml

解释：默认值10737418240（10GB），如果需要运行HBase的MR任务，可以减小此值，因为一个region对应一个map任务，如果单个region过大，会导致map任务执行时间过长。该值的意思就是，如果HFile的大小达到这个数值，则这个region会被切分为两个Hfile。

### 8、优化hbase客户端缓存

属性：hbase.client.write.buffer

文件：hbase-site.xml

解释：用于指定HBase客户端缓存，增大该值可以减少RPC调用次数，但是会消耗更多内存，反之则反之。一般我们需要设定一定的缓存大小，以达到减少RPC次数的目的。

### 9、指定scan.next扫描HBase所获取的行数

属性：hbase.client.scanner.caching

文件：hbase-site.xml

解释：用于指定scan.next方法获取的默认行数，值越大，消耗内存越大。

## 6、内存优化

HBase操作过程中需要大量的内存开销，毕竟Table是可以缓存在内存中的，一般会分配整个可用内存的70%给HBase的Java堆。但是不建议分配非常大的堆内存，因为GC过程持续太久会导致RegionServer处于长期不可用状态，一般16~48G内存就可以了，如果因为框架占用内存过高导致系统内存不足，框架一样会被系统服务拖死。

## 7、JVM优化

涉及文件：hbase-env.sh

### 1、并行GC

参数：-XX:+UseParallelGC

解释：开启并行GC

### 2、同时处理垃圾回收的线程数

参数：-XX:ParallelGCThreads=cpu\_core – 1

解释：该属性设置了同时处理垃圾回收的线程数。

### 3、禁用手动GC

参数：-XX:DisableExplicitGC

解释：防止开发人员手动调用GC

## 8、Zookeeper优化

### 1、优化Zookeeper会话超时时间

参数：zookeeper.session.timeout

文件：hbase-site.xml

解释：In hbase-site.xml, set zookeeper.session.timeout to 30 seconds or less to bound failure detection (20-30 seconds is a good start).该值会直接关系到master发现服务器宕机的最大周期，默认值为30秒，如果该值过小，会在HBase在写入大量数据发生而GC时，导致RegionServer短暂的不可用，从而没有向ZK发送心跳包，最终导致认为从节点shutdown。一般20台左右的集群需要配置5台zookeeper。