# HBase教案

# HBase基本介绍

**简介**

hbase是bigtable的开源java版本。是建立在hdfs之上，提供高可靠性、高性能、列存储、可伸缩、实时读写nosql的数据库系统。

它介于nosql和RDBMS之间，仅能通过主键(row key)和主键的range来检索数据，仅支持单行事务(可通过hive支持来实现多表join等复杂操作)。

主要用来存储结构化和半结构化的松散数据。

Hbase查询数据功能很简单，不支持join等复杂操作，不支持复杂的事务（行级的事务）

Hbase中支持的数据类型：byte[]

与hadoop一样，Hbase目标主要依靠横向扩展，通过不断增加廉价的商用服务器，来增加计算和存储能力。

HBase中的表一般有这样的特点：

* 大：一个表可以有上十亿行，上百万列
* 面向列:面向列(族)的存储和权限控制，列(族)独立检索。
* 稀疏:对于为空(null)的列，并不占用存储空间，因此，表可以设计的非常稀疏。

HBase的发展历程

HBase的原型是Google的BigTable论文，受到了该论文思想的启发，目前作为Hadoop的子项目来开发维护，用于支持结构化的数据存储。

官方网站：<http://hbase.apache.org>

\* 2006年Google发表BigTable白皮书

\* 2006年开始开发HBase

\* 2008 HBase成为了 Hadoop的子项目

\* 2010年HBase成为Apache顶级项目

# 2、HBase与Hadoop的关系

## 1、HDFS

\* 为分布式存储提供文件系统

\* 针对存储大尺寸的文件进行优化，不需要对HDFS上的文件进行随机读写

\* 直接使用文件

\* 数据模型不灵活

\* 使用文件系统和处理框架

\* 优化一次写入，多次读取的方式

## 2、HBase

\* 提供表状的面向列的数据存储

\* 针对表状数据的随机读写进行优化

\* 使用key-value操作数据

\* 提供灵活的数据模型

\* 使用表状存储，支持MapReduce，依赖HDFS

\* 优化了多次读，以及多次写

# 3、RDBMS与HBase的对比

## 1、关系型数据库

**结构：**

\* 数据库以表的形式存在

\* 支持FAT、NTFS、EXT、文件系统

\* 使用Commit log存储日志

\* 参考系统是坐标系统

\* 使用主键（PK）

\* 支持分区

\* 使用行、列、单元格

**功能：**

\* 支持向上扩展

\* 使用SQL查询

\* 面向行，即每一行都是一个连续单元

\* 数据总量依赖于服务器配置

\* 具有ACID支持

\* 适合结构化数据

\* 传统关系型数据库一般都是中心化的

\* 支持事务

\* 支持Join

## 2、HBase

**结构：**

\* 数据库以region的形式存在

\* 支持HDFS文件系统

\* 使用WAL（Write-Ahead Logs）存储日志

\* 参考系统是Zookeeper

\* 使用行键（row key）

\* 支持分片

\* 使用行、列、列族和单元格

**功能：**

\* 支持向外扩展

\* 使用API和MapReduce来访问HBase表数据

\* 面向列，即每一列都是一个连续的单元

\* 数据总量不依赖具体某台机器，而取决于机器数量

\* HBase不支持ACID（Atomicity、Consistency、Isolation、Durability）

\* 适合结构化数据和非结构化数据

\* 一般都是分布式的

\* HBase不支持事务

\* 不支持Join

# 4、HBase特征简要

**1）海量存储**

Hbase适合存储PB级别的海量数据，在PB级别的数据以及采用廉价PC存储的情况下，能在几十到百毫秒内返回数据。这与Hbase的极易扩展性息息相关。正式因为Hbase良好的扩展性，才为海量数据的存储提供了便利。

**2）列式存储**

这里的列式存储其实说的是列族存储，Hbase是根据列族来存储数据的。列族下面可以有非常多的列，列族在创建表的时候就必须指定。

**3）极易扩展**

Hbase的扩展性主要体现在两个方面，一个是基于上层处理能力（RegionServer）的扩展，一个是基于存储的扩展（HDFS）。  
通过横向添加RegionSever的机器，进行水平扩展，提升Hbase上层的处理能力，提升Hbsae服务更多Region的能力。

备注：RegionServer的作用是管理region、承接业务的访问，这个后面会详细的介绍通过横向添加Datanode的机器，进行存储层扩容，提升Hbase的数据存储能力和提升后端存储的读写能力。

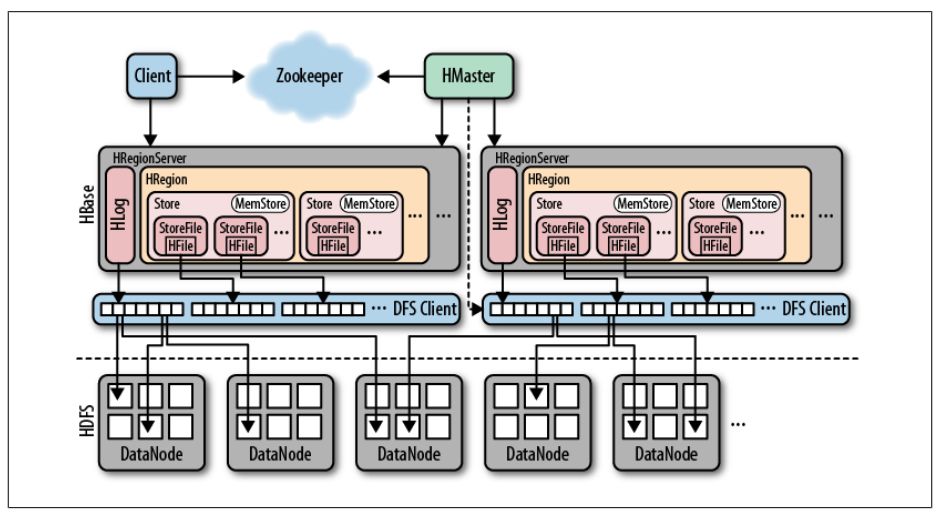
**4）高并发**

由于目前大部分使用Hbase的架构，都是采用的廉价PC，因此单个IO的延迟其实并不小，一般在几十到上百ms之间。这里说的高并发，主要是在并发的情况下，Hbase的单个IO延迟下降并不多。能获得高并发、低延迟的服务。

**5）稀疏**

稀疏主要是针对Hbase列的灵活性，在列族中，你可以指定任意多的列，在列数据为空的情况下，是不会占用存储空间的。

# 5、HBase的基础架构



## 1、HMaster

**功能：**

1) 监控RegionServer

2) 处理RegionServer故障转移

3) 处理元数据的变更

4) 处理region的分配或移除

5) 在空闲时间进行数据的负载均衡

6) 通过Zookeeper发布自己的位置给客户端

## 2、RegionServer

**功能：**

1) 负责存储HBase的实际数据

2) 处理分配给它的Region

3) 刷新缓存到HDFS

4) 维护HLog

5) 执行压缩

6) 负责处理Region分片

**组件：**

**1) Write-Ahead logs**

HBase的修改记录，当对HBase读写数据的时候，数据不是直接写进磁盘，它会在内存中保留一段时间（时间以及数据量阈值可以设定）。但把数据保存在内存中可能有更高的概率引起数据丢失，为了解决这个问题，数据会先写在一个叫做Write-Ahead logfile的文件中，然后再写入内存中。所以在系统出现故障的时候，数据可以通过这个日志文件重建。

**2) HFile**

这是在磁盘上保存原始数据的实际的物理文件，是实际的存储文件。

**3) Store**

HFile存储在Store中，一个Store对应HBase表中的一个列族。

**4) MemStore**

顾名思义，就是内存存储，位于内存中，用来保存当前的数据操作，所以当数据保存在WAL中之后，RegsionServer会在内存中存储键值对。

**5) Region**

Hbase表的分片，HBase表会根据RowKey值被切分成不同的region存储在RegionServer中，在一个RegionServer中可以有多个不同的region。

# 6、HBase的集群环境搭建

注意事项：HBase强依赖zookeeper和hadoop，安装HBase之前一定要保证zookeeper和hadoop启动成功，且服务正常运行

## 第一步：下载对应的HBase的安装包

所有关于CDH版本的软件包下载地址如下

<http://archive.cloudera.com/cdh5/cdh/5/>

HBase对应的版本下载地址如下

<http://archive.cloudera.com/cdh5/cdh/5/hbase-1.2.0-cdh5.14.0.tar.gz>

## 第二步：压缩包上传并解压

将我们的压缩包上传到node01服务器的/export/softwares路径下并解压

cd /export/softwares/

tar -zxvf hbase-1.2.0-cdh5.14.0.tar -C ../servers/

## 第三步：修改配置文件

第一台机器进行修改配置文件

cd /export/servers/hbase-1.2.0-cdh5.14.0/conf

### 修改第一个配置文件hbase-env.sh

注释掉HBase使用内部zk

vim hbase-env.sh

export JAVA\_HOME=/export/servers/jdk1.8.0\_141

export HBASE\_MANAGES\_ZK=false

### 修改第二个配置文件hbase-site.xml

修改hbase-site.xml

vim hbase-site.xml

<configuration>

<property>

<name>hbase.rootdir</name>

<value>hdfs://node01:8020/hbase</value>

</property>

<property>

<name>hbase.cluster.distributed</name>

<value>true</value>

</property>

<!-- 0.98后的新变动，之前版本没有.port,默认端口为60000 -->

<property>

<name>hbase.master.port</name>

<value>16000</value>

</property>

<property>

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

<value>node01:2181,node02:2181,node03:2181</value>

</property>

<property>

<name>hbase.zookeeper.property.dataDir</name>

<value>/export/servers/zookeeper-3.4.5-cdh5.14.0/zkdatas</value>

</property>

</configuration>

### 修改第三个配置文件regionservers

vim regionservers

node01

node02

node03

### 创建back-masters配置文件，实现HMaster的高可用

cd /export/servers/hbase-1.2.0-cdh5.14.0/conf

vim backup-masters

node02

## 第四步：安装包分发到其他机器

将我们第一台机器的hbase的安装包拷贝到其他机器上面去

cd /export/servers/

scp -r hbase-1.2.0-cdh5.14.0/ node02:$PWD

scp -r hbase-1.2.0-cdh5.14.0/ node03:$PWD

## 第五步：三台机器创建软连接

因为hbase需要读取hadoop的core-site.xml以及hdfs-site.xml当中的配置文件信息，所以我们三台机器都要执行以下命令创建软连接

ln -s /export/servers/hadoop-2.6.0-cdh5.14.0/etc/hadoop/core-site.xml /export/servers/hbase-1.2.0-cdh5.14.0/conf/core-site.xml

ln -s /export/servers/hadoop-2.6.0-cdh5.14.0/etc/hadoop/hdfs-site.xml /export/servers/hbase-1.2.0-cdh5.14.0/conf/hdfs-site.xml

## 第六步：三台机器添加HBASE\_HOME的环境变量

vim /etc/profile

export HBASE\_HOME=/export/servers/hbase-1.2.0-cdh5.14.0

export PATH=:$HBASE\_HOME/bin:$PATH

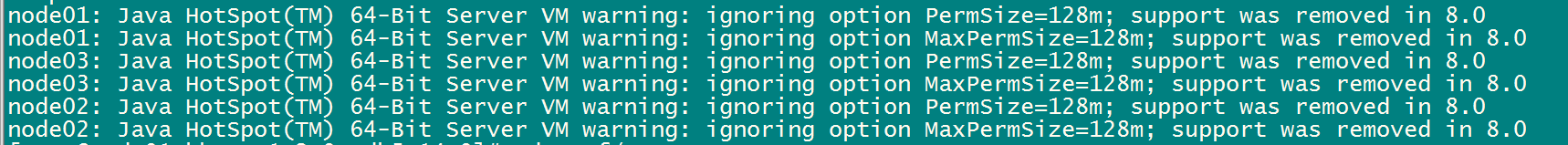
## 第七步：HBase集群启动

第一台机器执行以下命令进行启动

cd /export/servers/hbase-1.2.0-cdh5.14.0

bin/start-hbase.sh

警告提示：HBase启动的时候会产生一个警告，这是因为jdk7与jdk8的问题导致的，如果linux服务器安装jdk8就会产生这样的一个警告



我们可以只是掉所有机器的hbase-env.sh当中的

“HBASE\_MASTER\_OPTS”和“HBASE\_REGIONSERVER\_OPTS”配置 来解决这个问题。不过警告不影响我们正常运行，可以不用解决

我们也可以执行以下命令单节点进行启动

启动HMaster命令

bin/hbase-daemon.sh start master

启动HRegionServer命令

bin/hbase-daemon.sh start regionserver

为了解决HMaster单点故障问题，我们可以在node02和node03机器上面都可以启动HMaster节点的进程，以实现HMaster的高可用

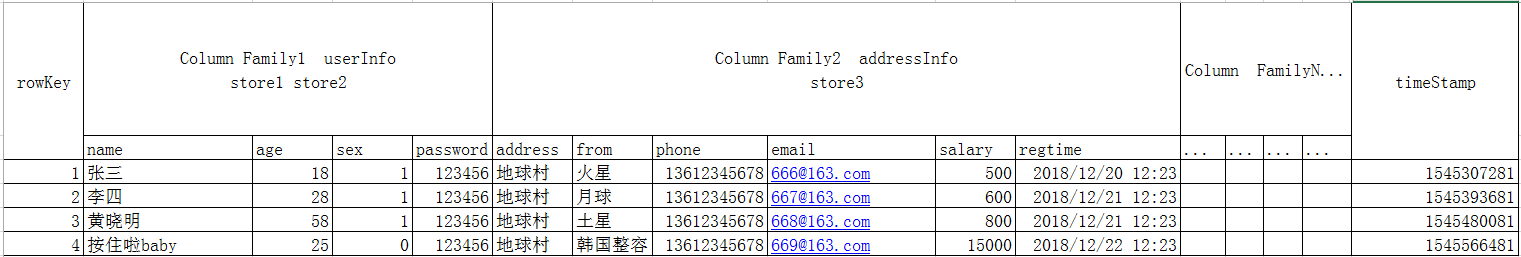
bin/hbase-daemon.sh start master

## 第七步：页面访问

浏览器页面访问

<http://node01:60010/master-status>

## HBASE的表模型基本介绍



# 7、HBase常用shell操作

## 1、进入HBase客户端命令操作界面

$ bin/hbase shell

## 2、查看帮助命令

hbase(main):001:0> help

## 3、查看当前数据库中有哪些表

hbase(main):002:0> list

## 4、创建一张表

创建user表，包含info、data两个列族

hbase(main):010:0> create 'user', 'info', 'data'

或者

hbase(main):010:0> create 'user', {NAME => 'info', VERSIONS => '3'}，{NAME => 'data'}

## 5、添加数据操作

向user表中插入信息，row key为rk0001，列族info中添加name列标示符，值为zhangsan

hbase(main):011:0> put 'user', 'rk0001', 'info:name', 'zhangsan'

向user表中插入信息，row key为rk0001，列族info中添加gender列标示符，值为female

hbase(main):012:0> put 'user', 'rk0001', 'info:gender', 'female'

向user表中插入信息，row key为rk0001，列族info中添加age列标示符，值为20

hbase(main):013:0> put 'user', 'rk0001', 'info:age', 20

向user表中插入信息，row key为rk0001，列族data中添加pic列标示符，值为picture

hbase(main):014:0> put 'user', 'rk0001', 'data:pic', 'picture'

## 6、查询数据操作

### 1、通过rowkey进行查询

获取user表中row key为rk0001的所有信息

hbase(main):015:0> get 'user', 'rk0001'

### 2、查看rowkey下面的某个列族的信息

获取user表中row key为rk0001，info列族的所有信息

hbase(main):016:0> get 'user', 'rk0001', 'info'

### 3、查看rowkey指定列族指定字段的值

获取user表中row key为rk0001，info列族的name、age列标示符的信息

hbase(main):017:0> get 'user', 'rk0001', 'info:name', 'info:age'

### 4、查看rowkey指定多个列族的信息

获取user表中row key为rk0001，info、data列族的信息

hbase(main):018:0> get 'user', 'rk0001', 'info', 'data'

或者你也可以这样写

hbase(main):019:0> get 'user', 'rk0001', {COLUMN => ['info', 'data']}

或者你也可以这样写，也行

hbase(main):020:0> get 'user', 'rk0001', {COLUMN => ['info:name', 'data:pic']}

### 4、指定rowkey与列值查询

获取user表中row key为rk0001，cell的值为zhangsan的信息

hbase(main):030:0> get 'user', 'rk0001', {FILTER => "ValueFilter(=, 'binary:zhangsan')"}

### 5、指定rowkey与列值模糊查询

获取user表中row key为rk0001，列标示符中含有a的信息

hbase(main):031:0> get 'user', 'rk0001', {FILTER => "(QualifierFilter(=,'substring:a'))"}

继续插入一批数据

hbase(main):032:0> put 'user', 'rk0002', 'info:name', 'fanbingbing'

hbase(main):033:0> put 'user', 'rk0002', 'info:gender', 'female'

hbase(main):034:0> put 'user', 'rk0002', 'info:nationality', '中国'

hbase(main):035:0> get 'user', 'rk0002', {FILTER => "ValueFilter(=, 'binary:中国')"}

### 6、查询所有数据

查询user表中的所有信息

scan 'user'

### 7、列族查询

查询user表中列族为info的信息

scan 'user', {COLUMNS => 'info'}

scan 'user', {COLUMNS => 'info', RAW => true, VERSIONS => 5}

scan 'user', {COLUMNS => 'info', RAW => true, VERSIONS => 3}

### 8、多列族查询

查询user表中列族为info和data的信息

scan 'user', {COLUMNS => ['info', 'data']}

scan 'user', {COLUMNS => ['info:name', 'data:pic']}

### 9、指定列族与某个列名查询

查询user表中列族为info、列标示符为name的信息

scan 'user', {COLUMNS => 'info:name'}

### 10、指定列族与列名以及限定版本查询

查询user表中列族为info、列标示符为name的信息,并且版本最新的5个

scan 'user', {COLUMNS => 'info:name', VERSIONS => 5}

### 11、指定多个列族与按照数据值模糊查询

查询user表中列族为info和data且列标示符中含有a字符的信息

scan 'user', {COLUMNS => ['info', 'data'], FILTER => "(QualifierFilter(=,'substring:a'))"}

### 12、rowkey的范围值查询

查询user表中列族为info，rk范围是[rk0001, rk0003)的数据

scan 'user', {COLUMNS => 'info', STARTROW => 'rk0001', ENDROW => 'rk0003'}

### 13、指定rowkey模糊查询

查询user表中row key以rk字符开头的

scan 'user',{FILTER=>"PrefixFilter('rk')"}

### 14、指定数据范围值查询

查询user表中指定范围的数据

scan 'user', {TIMERANGE => [1392368783980, 1392380169184]}

## 7、更新数据操作

### 1、更新数据值

更新操作同插入操作一模一样，只不过有数据就更新，没数据就添加

### 2、更新版本号

将user表的f1列族版本号改为5

hbase(main):050:0> alter 'user', NAME => 'info', VERSIONS => 5

## 8、删除数据以及删除表操作

### 1、指定rowkey以及列名进行删除

删除user表row key为rk0001，列标示符为info:name的数据

hbase(main):045:0> delete 'user', 'rk0001', 'info:name'

### 2、指定rowkey，列名以及字段值进行删除

删除user表row key为rk0001，列标示符为info:name，timestamp为1392383705316的数据

delete 'user', 'rk0001', 'info:name', 1392383705316

### 3、删除一个列族

删除一个列族：

alter 'user', NAME => 'info', METHOD => 'delete'

或 alter 'user', 'delete' => 'info'

### 4、清空表数据

hbase(main):017:0> truncate 'user'

### 5、删除表

首先需要先让该表为disable状态，使用命令：

hbase(main):049:0> disable 'user'

然后才能drop这个表，使用命令：

hbase(main):050:0> drop 'user'

(注意：如果直接drop表，会报错：Drop the named table. Table must first be disabled)

## 9、统计一张表有多少行数据

hbase(main):053:0> count 'user'

# 8、HBase的高级shell管理命令

## 1、status

例如：显示服务器状态

hbase(main):058:0> status 'node01'

## 2、whoami

显示HBase当前用户，例如：

hbase> whoami

## 3、list

显示当前所有的表

## 4、count

统计指定表的记录数，例如：

hbase> count 'user'

## 5、describe

展示表结构信息

## 6、exists

检查表是否存在，适用于表量特别多的情况

## 7、is\_enabled、is\_disabled

检查表是否启用或禁用

## 8、alter

该命令可以改变表和列族的模式，例如：

**为当前表增加列族：**

hbase> alter 'user', NAME => 'CF2', VERSIONS => 2

**为当前表删除列族：**

hbase(main):002:0> alter 'user', 'delete' => 'CF2'

## 9、disable/enable

禁用一张表/启用一张表

## 10、drop

删除一张表，记得在删除表之前必须先禁用

## 11、truncate

禁用表-删除表-创建表

# 9、HBase的java代码开发

熟练掌握通过使用java代码实现HBase数据库当中的数据增删改查的操作，特别是各种查询，熟练运用

## 第一步：创建maven工程，导入jar包

<repositories>

<repository>

<id>cloudera</id>

<url>https://repository.cloudera.com/artifactory/cloudera-repos/</url>

</repository>

</repositories>

<dependencies>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-client</artifactId>

<version>2.6.0-mr1-cdh5.14.0</version>

</dependency>

<dependency>

<groupId>org.apache.hbase</groupId>

<artifactId>hbase-client</artifactId>

<version>1.2.0-cdh5.14.0</version>

</dependency>

<dependency>

<groupId>org.apache.hbase</groupId>

<artifactId>hbase-server</artifactId>

<version>1.2.0-cdh5.14.0</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.testng</groupId>

<artifactId>testng</artifactId>

<version>6.14.3</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>maven-compiler-plugin</artifactId>

<version>3.0</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

<encoding>UTF-8</encoding>

<!-- <verbal>true</verbal>-->

</configuration>

</plugin>

<plugin>

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

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

<version>2.2</version>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>shade</goal>

</goals>

<configuration>

<filters>

<filter>

<artifact>\*:\*</artifact>

<excludes>

<exclude>META-INF/\*.SF</exclude>

<exclude>META-INF/\*.DSA</exclude>

<exclude>META-INF/\*/RSA</exclude>

</excludes>

</filter>

</filters>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

## 第二步：开发javaAPI操作HBase表数据

### 1、创建表myuser

@Test

public void createTable() throws IOException {

//创建配置文件对象，并指定zookeeper的连接地址

Configuration configuration = HBaseConfiguration.create();

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

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

//集群配置↓

//configuration.set("hbase.zookeeper.quorum", "101.236.39.141,101.236.46.114,101.236.46.113");

configuration.set("hbase.master", "node01:60000");

Connection connection = ConnectionFactory.createConnection(configuration);

Admin admin = connection.getAdmin();

//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("myuser"));

//添加列族

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

//添加列族

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

//创建表

boolean myuser = admin.tableExists(TableName.valueOf("myuser"));

if(!myuser){

admin.createTable(hTableDescriptor);

}

//关闭客户端连接

admin.close();

}

### 2、向表中添加数据

/\*\*

\* 插入数据

\*/

@Test

public void addDatas() throws IOException {

//获取连接

Configuration configuration = HBaseConfiguration.create();

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

Connection connection = ConnectionFactory.createConnection(configuration);

//获取表

Table myuser = connection.getTable(TableName.valueOf("myuser"));

//创建put对象，并指定rowkey

Put put = new Put("0001".getBytes());

put.addColumn("f1".getBytes(),"id".getBytes(), Bytes.toBytes(1));

put.addColumn("f1".getBytes(),"name".getBytes(), Bytes.toBytes("张三"));

put.addColumn("f1".getBytes(),"age".getBytes(), Bytes.toBytes(18));

put.addColumn("f2".getBytes(),"address".getBytes(), Bytes.toBytes("地球人"));

put.addColumn("f2".getBytes(),"phone".getBytes(), Bytes.toBytes("15874102589"));

//插入数据

myuser.put(put);

//关闭表

myuser.close();

}

### 3、查询数据

#### 初始化一批数据到HBase当中用于查询

@Test

public void insertBatchData() throws IOException {

//获取连接

Configuration configuration = HBaseConfiguration.create();

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

Connection connection = ConnectionFactory.createConnection(configuration);

//获取表

Table myuser = connection.getTable(TableName.valueOf("myuser"));

//创建put对象，并指定rowkey

Put put = new Put("0002".getBytes());

put.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(1));

put.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("曹操"));

put.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(30));

put.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));

put.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("沛国谯县"));

put.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("16888888888"));

put.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("helloworld"));

Put put2 = new Put("0003".getBytes());

put2.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(2));

put2.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("刘备"));

put2.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(32));

put2.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));

put2.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("幽州涿郡涿县"));

put2.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("17888888888"));

put2.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("talk is cheap , show me the code"));

Put put3 = new Put("0004".getBytes());

put3.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(3));

put3.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("孙权"));

put3.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(35));

put3.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));

put3.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("下邳"));

put3.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("12888888888"));

put3.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("what are you 弄啥嘞！"));

Put put4 = new Put("0005".getBytes());

put4.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(4));

put4.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("诸葛亮"));

put4.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(28));

put4.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));

put4.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("四川隆中"));

put4.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("14888888888"));

put4.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("出师表你背了嘛"));

Put put5 = new Put("0005".getBytes());

put5.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(5));

put5.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("司马懿"));

put5.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(27));

put5.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));

put5.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("哪里人有待考究"));

put5.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("15888888888"));

put5.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("跟诸葛亮死掐"));

Put put6 = new Put("0006".getBytes());

put6.addColumn("f1".getBytes(),"id".getBytes(),Bytes.toBytes(5));

put6.addColumn("f1".getBytes(),"name".getBytes(),Bytes.toBytes("xiaobubu—吕布"));

put6.addColumn("f1".getBytes(),"age".getBytes(),Bytes.toBytes(28));

put6.addColumn("f2".getBytes(),"sex".getBytes(),Bytes.toBytes("1"));

put6.addColumn("f2".getBytes(),"address".getBytes(),Bytes.toBytes("内蒙人"));

put6.addColumn("f2".getBytes(),"phone".getBytes(),Bytes.toBytes("15788888888"));

put6.addColumn("f2".getBytes(),"say".getBytes(),Bytes.toBytes("貂蝉去哪了"));

List<Put> listPut = new ArrayList<Put>();

listPut.add(put);

listPut.add(put2);

listPut.add(put3);

listPut.add(put4);

listPut.add(put5);

listPut.add(put6);

myuser.put(listPut);

myuser.close();

}

#### 按照rowkey进行查询获取所有列的所有值

查询主键rowkey为0003的人

/\*\*

\* 查询数据，按照主键id进行查询

\*/

@Test

public void searchData() throws IOException {

Configuration configuration = HBaseConfiguration.create();

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

Connection connection = ConnectionFactory.createConnection(configuration);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

Get get = new Get(Bytes.toBytes("0003"));

Result result = myuser.get(get);

Cell[] cells = result.rawCells();

//获取所有的列名称以及列的值

for (Cell cell : cells) {

//注意，如果列属性是int类型，那么这里就不会显示

System.out.println(Bytes.toString(cell.getQualifierArray(),cell.getQualifierOffset(),cell.getQualifierLength()));

System.out.println(Bytes.toString(cell.getValueArray(),cell.getValueOffset(),cell.getValueLength()));

}

myuser.close();

}

#### 按照rowkey查询指定列族下面的指定列的值

/\*\*

\* 通过rowkey查询指定列族下面的指定列的值

\*/

@Test

public void searchData2() 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);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

//通过rowKey进行查询

Get get = new Get("0003".getBytes());

get.addColumn("f1".getBytes(),"id".getBytes());

Result result = myuser.get(get);

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".getBytes())));

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "age".getBytes())));

System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "name".getBytes())));

myuser.close();

}

#### 通过startRowKey和endRowKey进行扫描

/\*\*

\* 通过startRowKey和endRowKey进行扫描查询

\*/

@Test

public void scanRowKey() 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);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

Scan scan = new Scan();

scan.setStartRow("0004".getBytes());

scan.setStopRow("0006".getBytes());

ResultScanner resultScanner = myuser.getScanner(scan);

for (Result result : resultScanner) {

//获取rowkey

System.out.println(Bytes.toString(result.getRow()));

//遍历获取得到所有的列族以及所有的列的名称

KeyValue[] raw = result.raw();

for (KeyValue keyValue : raw) {

//获取所属列族

System.out.println(Bytes.toString(keyValue.getFamilyArray(),keyValue.getFamilyOffset(),keyValue.getFamilyLength()));

System.out.println(Bytes.toString(keyValue.getQualifierArray(),keyValue.getQualifierOffset(),keyValue.getQualifierLength()));

}

//指定列族以及列打印列当中的数据出来

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".getBytes())));

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "age".getBytes())));

System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "name".getBytes())));

}

myuser.close();

}

#### 通过scan进行全表扫描

/\*\*

\* 全表扫描

\*/

@Test

public void scanAllData() 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);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

Scan scan = new Scan();

ResultScanner resultScanner = myuser.getScanner(scan);

for (Result result : resultScanner) {

//获取rowkey

System.out.println(Bytes.toString(result.getRow()));

//指定列族以及列打印列当中的数据出来

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".getBytes())));

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "age".getBytes())));

System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "name".getBytes())));

}

myuser.close();

}

### 4、过滤器查询

过滤器的类型很多，但是可以分为两大类——比较过滤器，专用过滤器

过滤器的作用是在服务端判断数据是否满足条件，然后只将满足条件的数据返回给客户端；

hbase过滤器的比较运算符：

|  |
| --- |
| LESS <  LESS\_OR\_EQUAL <=  EQUAL =  NOT\_EQUAL <>  GREATER\_OR\_EQUAL >=  GREATER >  NO\_OP 排除所有 |

Hbase过滤器的比较器（指定比较机制）：

|  |
| --- |
| BinaryComparator 按字节索引顺序比较指定字节数组，采用Bytes.compareTo(byte[])  BinaryPrefixComparator 跟前面相同，只是比较左端的数据是否相同  NullComparator 判断给定的是否为空  BitComparator 按位比较  RegexStringComparator 提供一个正则的比较器，仅支持 EQUAL 和非EQUAL  SubstringComparator 判断提供的子串是否出现在value中。 |

#### 1、比较过滤器

##### 1、rowKey过滤器RowFilter

通过RowFilter过滤比rowKey 0003小的所有值出来

/\*\*

\* hbase行键过滤器RowFilter

\*/

@Test

public void rowKeyFilter() 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);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

Scan scan = new Scan();

RowFilter rowFilter = new RowFilter(CompareFilter.CompareOp.LESS\_OR\_EQUAL, new BinaryComparator(Bytes.toBytes("0003")));

scan.setFilter(rowFilter);

ResultScanner resultScanner = myuser.getScanner(scan);

for (Result result : resultScanner) {

//获取rowkey

System.out.println(Bytes.toString(result.getRow()));

//指定列族以及列打印列当中的数据出来

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".getBytes())));

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "age".getBytes())));

System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "name".getBytes())));

}

myuser.close();

}

##### 2、列族过滤器FamilyFilter

查询比f2列族小的所有的列族内的数据

/\*\*

\* hbase列族过滤器FamilyFilter

\*/

@Test

public void familyFilter() 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);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

Scan scan = new Scan();

FamilyFilter familyFilter = new FamilyFilter(CompareFilter.CompareOp.LESS, new SubstringComparator("f2"));

scan.setFilter(familyFilter);

ResultScanner resultScanner = myuser.getScanner(scan);

for (Result result : resultScanner) {

//获取rowkey

System.out.println(Bytes.toString(result.getRow()));

//指定列族以及列打印列当中的数据出来

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".getBytes())));

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "age".getBytes())));

System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "name".getBytes())));

}

myuser.close();

}

##### 3、列过滤器QualifierFilter

只查询name列的值

/\*\*

\* hbase列过滤器

\*/

@Test

public void qualifierFilter() 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);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

Scan scan = new Scan();

QualifierFilter qualifierFilter = new QualifierFilter(CompareFilter.CompareOp.EQUAL, new SubstringComparator("name"));

scan.setFilter(qualifierFilter);

ResultScanner resultScanner = myuser.getScanner(scan);

for (Result result : resultScanner) {

//获取rowkey

System.out.println(Bytes.toString(result.getRow()));

//指定列族以及列打印列当中的数据出来

// System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".getBytes())));

System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "name".getBytes())));

}

myuser.close();

}

##### 4、列值过滤器ValueFilter

查询所有列当中包含8的数据

/\*\*

\* hbase值过滤器

\* 查询包含8的列值

\*/

@Test

public void valueFilter() 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);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

Scan scan = new Scan();

ValueFilter valueFilter = new ValueFilter(CompareFilter.CompareOp.EQUAL, new SubstringComparator("8"));

scan.setFilter(valueFilter);

ResultScanner resultScanner = myuser.getScanner(scan);

for (Result result : resultScanner) {

//获取rowkey

System.out.println(Bytes.toString(result.getRow()));

//指定列族以及列打印列当中的数据出来

// System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".getBytes())));

System.out.println(Bytes.toString(result.getValue("f2".getBytes(), "phone".getBytes())));

}

myuser.close();

}

#### 2、专用过滤器

##### 1、单列值过滤器 SingleColumnValueFilter

SingleColumnValueFilter会返回满足条件的整列值的所有字段

/\*\*

\* 单列值过滤器，返回满足条件的整行数据

\*/

@Test

public void singleColumnFilter() 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);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

Scan scan = new Scan();

SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilter("f1".getBytes(), "name".getBytes(), CompareFilter.CompareOp.EQUAL, "刘备".getBytes());

scan.setFilter(singleColumnValueFilter);

ResultScanner resultScanner = myuser.getScanner(scan);

for (Result result : resultScanner) {

//获取rowkey

System.out.println(Bytes.toString(result.getRow()));

//指定列族以及列打印列当中的数据出来

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".getBytes())));

System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "name".getBytes())));

System.out.println(Bytes.toString(result.getValue("f2".getBytes(), "phone".getBytes())));

}

myuser.close();

}

##### 2、列值排除过滤器SingleColumnValueExcludeFilter

与SingleColumnValueFilter相反，会排除掉指定的列，其他的列全部返回

##### 3、rowkey前缀过滤器PrefixFilter

查询以00开头的所有前缀的rowkey

/\*\*

\* 行键前缀过滤器

\*/

@Test

public void preFilter() 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);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

Scan scan = new Scan();

PrefixFilter prefixFilter = new PrefixFilter("00".getBytes());

scan.setFilter(prefixFilter);

ResultScanner resultScanner = myuser.getScanner(scan);

for (Result result : resultScanner) {

//获取rowkey

System.out.println(Bytes.toString(result.getRow()));

//指定列族以及列打印列当中的数据出来

System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".getBytes())));

System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "name".getBytes())));

System.out.println(Bytes.toString(result.getValue("f2".getBytes(), "phone".getBytes())));

}

myuser.close();

}

##### 4、分页过滤器PageFilter

通过pageFilter实现分页过滤器

@Test

public void pageFilter2() 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);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

int pageNum = 3;

int pageSize = 2;

Scan scan = new Scan();

if (pageNum == 1) {

PageFilter filter = new PageFilter(pageSize);

scan.setStartRow(Bytes.toBytes(""));

scan.setFilter(filter);

scan.setMaxResultSize(pageSize);

ResultScanner scanner = myuser.getScanner(scan);

for (Result result : scanner) {

//获取rowkey

System.out.println(Bytes.toString(result.getRow()));

//指定列族以及列打印列当中的数据出来

// System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".getBytes())));

System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "name".getBytes())));

//System.out.println(Bytes.toString(result.getValue("f2".getBytes(), "phone".getBytes())));

}

}else{

String startRowKey ="";

PageFilter filter = new PageFilter((pageNum - 1) \* pageSize + 1 );

scan.setStartRow(startRowKey.getBytes());

scan.setMaxResultSize((pageNum - 1) \* pageSize + 1);

scan.setFilter(filter);

ResultScanner scanner = myuser.getScanner(scan);

for (Result result : scanner) {

byte[] row = result.getRow();

startRowKey = new String(row);

}

Scan scan2 = new Scan();

scan2.setStartRow(startRowKey.getBytes());

scan2.setMaxResultSize(Long.valueOf(pageSize));

PageFilter filter2 = new PageFilter(pageSize);

scan2.setFilter(filter2);

ResultScanner scanner1 = myuser.getScanner(scan2);

for (Result result : scanner1) {

byte[] row = result.getRow();

System.out.println(new String(row));

}

}

myuser.close();

}

#### 3、多过滤器综合查询FilterList

需求：使用SingleColumnValueFilter查询f1列族，name为刘备的数据，并且同时满足rowkey的前缀以00开头的数据（PrefixFilter）

/\*\*

\* 多过滤器组合使用

\*/

@Test

public void manyFilter() 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);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

Scan scan = new Scan();

FilterList filterList = new FilterList();

SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilter("f1".getBytes(), "name".getBytes(), CompareFilter.CompareOp.EQUAL, "刘备".getBytes());

PrefixFilter prefixFilter = new PrefixFilter("00".getBytes());

filterList.addFilter(singleColumnValueFilter);

filterList.addFilter(prefixFilter);

scan.setFilter(filterList);

ResultScanner scanner = myuser.getScanner(scan);

for (Result result : scanner) {

//获取rowkey

System.out.println(Bytes.toString(result.getRow()));

//指定列族以及列打印列当中的数据出来

// System.out.println(Bytes.toInt(result.getValue("f1".getBytes(), "id".getBytes())));

System.out.println(Bytes.toString(result.getValue("f1".getBytes(), "name".getBytes())));

//System.out.println(Bytes.toString(result.getValue("f2".getBytes(), "phone".getBytes())));

}

myuser.close();

}

### 5、根据rowkey删除数据

/\*\*

\* 删除数据

\*/

@Test

public void deleteByRowKey() 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);

Table myuser = connection.getTable(TableName.valueOf("myuser"));

Delete delete = new Delete("0001".getBytes());

myuser.delete(delete);

myuser.close();

}

### 6、删除表操作

@Test

public void deleteTable() 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();

admin.disableTable(TableName.valueOf("myuser"));

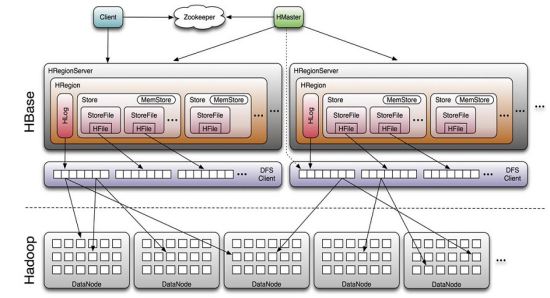
admin.deleteTable(TableName.valueOf("myuser"));

admin.close();

}

# 10、HBase底层原理

## 系统架构



Client

1 包含访问hbase的接口，client维护着一些cache来加快对hbase的访问，比如regione的位置信息。

Zookeeper

1 保证任何时候，集群中只有一个master

2 存贮所有Region的寻址入口

3 实时监控Region Server的状态，将Region server的上线和下线信息实时通知给Master

4 存储Hbase的schema,包括有哪些table，每个table有哪些column family

Master职责

1 为Region server分配region

2 负责region server的负载均衡

3 发现失效的region server并重新分配其上的region

4 HDFS上的垃圾文件回收

5 处理schema更新请求

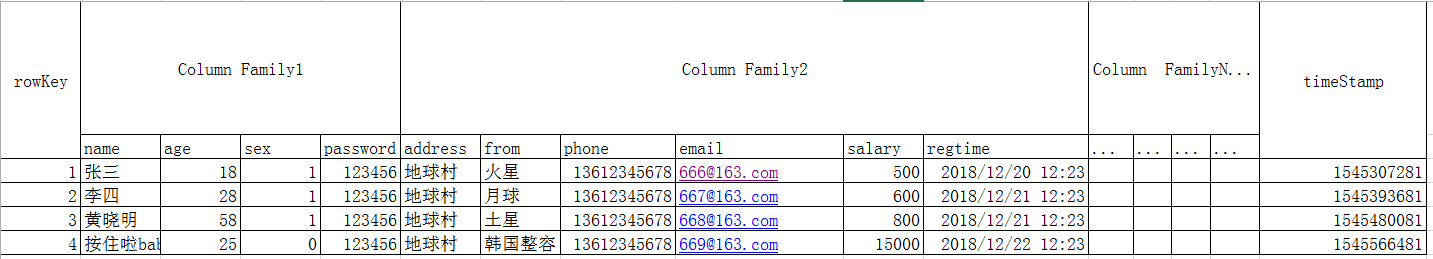
Region Server职责

1 Region server维护Master分配给它的region，处理对这些region的IO请求

2 Region server负责切分在运行过程中变得过大的region

可以看到，client访问hbase上数据的过程并不需要master参与（寻址访问zookeeper和region server，数据读写访问regione server），master仅仅维护者table和region的元数据信息，负载很低。

## HBase的表数据模型



### Row Key

与nosql数据库们一样,row key是用来检索记录的主键。访问hbase table中的行，只有三种方式：

1 通过单个row key访问

2 通过row key的range

3 全表扫描

Row key行键 (Row key)可以是任意字符串(最大长度是 64KB，实际应用中长度一般为 10-100bytes)，在hbase内部，row key保存为字节数组。

**Hbase会对表中的数据按照rowkey排序(字典顺序)**

存储时，数据按照Row key的字典序(byte order)排序存储。设计key时，要充分排序存储这个特性，将经常一起读取的行存储放到一起。(位置相关性)

注意：

字典序对int排序的结果是

1,10,100,11,12,13,14,15,16,17,18,19,2,20,21,…,9,91,92,93,94,95,96,97,98,99。要保持整形的自然序，行键必须用0作左填充。

行的一次读写是原子操作 (不论一次读写多少列)。这个设计决策能够使用户很容易的理解程序在对同一个行进行并发更新操作时的行为。

### 列族Column Family

hbase表中的每个列，都归属与某个列族。列族是表的schema的一部分(而列不是)，必须在使用表之前定义。

列名都以列族作为前缀。例如courses:history ， courses:math 都属于 courses 这个列族。

访问控制、磁盘和内存的使用统计都是在列族层面进行的。

列族越多，在取一行数据时所要参与IO、搜寻的文件就越多，所以，如果没有必要，不要设置太多的列族

### 列 Column

列族下面的具体列，属于某一个ColumnFamily,类似于我们mysql当中创建的具体的列

### 时间戳

HBase中通过row和columns确定的为一个存贮单元称为cell。每个 cell都保存着同一份数据的多个版本。版本通过时间戳来索引。时间戳的类型是 64位整型。时间戳可以由hbase(在数据写入时自动 )赋值，此时时间戳是精确到毫秒的当前系统时间。时间戳也可以由客户显式赋值。如果应用程序要避免数据版本冲突，就必须自己生成具有唯一性的时间戳。每个 cell中，不同版本的数据按照时间倒序排序，即最新的数据排在最前面。

为了避免数据存在过多版本造成的的管理 (包括存贮和索引)负担，hbase提供了两种数据版本回收方式：

* 保存数据的最后n个版本
* 保存最近一段时间内的版本（设置数据的生命周期TTL）。

用户可以针对每个列族进行设置。

### Cell

由{row key, column( =<family> + <label>), version} 唯一确定的单元。

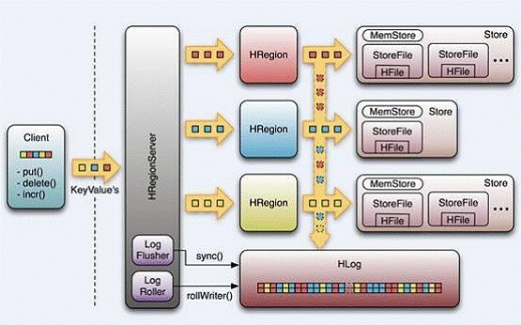
cell中的数据是没有类型的，全部是字节码形式存贮。

### VersionNum

数据的版本号，每条数据可以有多个版本号，默认值为系统时间戳，类型为Long

## 物理存储

### 1、整体结构



1 Table中的所有行都按照row key的字典序排列。

2 Table 在行的方向上分割为多个Hregion。

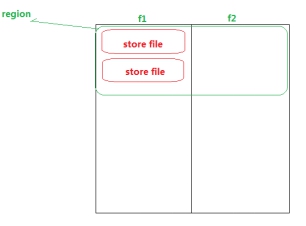
3 region按大小分割的(默认10G)，每个表一开始只有一个region，随着数据不断插入表，region不断增大，当增大到一个阀值的时候，Hregion就会等分会两个新的Hregion。当table中的行不断增多，就会有越来越多的Hregion。

4 Hregion是Hbase中分布式存储和负载均衡的最小单元。最小单元就表示不同的Hregion可以分布在不同的HRegion server上。但一个Hregion是不会拆分到多个server上的。

5 HRegion虽然是负载均衡的最小单元，但并不是物理存储的最小单元。

事实上，HRegion由一个或者多个Store组成，每个store保存一个column family。

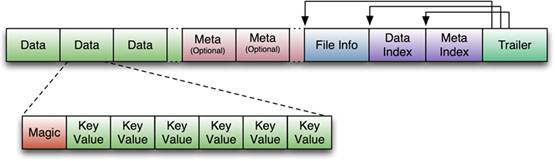
每个Strore又由一个memStore和0至多个StoreFile组成。如上图



### 2、STORE FILE & HFILE结构

StoreFile以HFile格式保存在HDFS上。

附：HFile的格式为：

[](http://www.searchtb.com/wp-content/uploads/2011/01/image0080.jpg)

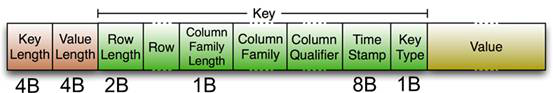
首先HFile文件是不定长的，长度固定的只有其中的两块：Trailer和FileInfo。正如图中所示的，Trailer中有指针指向其他数 据块的起始点。

File Info中记录了文件的一些Meta信息，例如：AVG\_KEY\_LEN, AVG\_VALUE\_LEN, LAST\_KEY, COMPARATOR, MAX\_SEQ\_ID\_KEY等。

Data Index和Meta Index块记录了每个Data块和Meta块的起始点。

Data Block是HBase I/O的基本单元，为了提高效率，HRegionServer中有基于LRU的Block Cache机制。每个Data块的大小可以在创建一个Table的时候通过参数指定，大号的Block有利于顺序Scan，小号Block利于随机查询。 每个Data块除了开头的Magic以外就是一个个KeyValue对拼接而成, Magic内容就是一些随机数字，目的是防止数据损坏。

HFile里面的每个KeyValue对就是一个简单的byte数组。但是这个byte数组里面包含了很多项，并且有固定的结构。我们来看看里面的具体结构：

[](http://www.searchtb.com/wp-content/uploads/2011/01/image0090.jpg)

开始是两个固定长度的数值，分别表示Key的长度和Value的长度。紧接着是Key，开始是固定长度的数值，表示RowKey的长度，紧接着是 RowKey，然后是固定长度的数值，表示Family的长度，然后是Family，接着是Qualifier，然后是两个固定长度的数值，表示Time Stamp和Key Type（Put/Delete）。Value部分没有这么复杂的结构，就是纯粹的二进制数据了。

HFile分为六个部分：

Data Block 段–保存表中的数据，这部分可以被压缩

Meta Block 段 (可选的)–保存用户自定义的kv对，可以被压缩。

File Info 段–Hfile的元信息，不被压缩，用户也可以在这一部分添加自己的元信息。

Data Block Index 段–Data Block的索引。每条索引的key是被索引的block的第一条记录的key。

Meta Block Index段 (可选的)–Meta Block的索引。

Trailer–这一段是定长的。保存了每一段的偏移量，读取一个HFile时，会首先 读取Trailer，Trailer保存了每个段的起始位置(段的Magic Number用来做安全check)，然后，DataBlock Index会被读取到内存中，这样，当检索某个key时，不需要扫描整个HFile，而只需从内存中找到key所在的block，通过一次磁盘io将整个 block读取到内存中，再找到需要的key。DataBlock Index采用LRU机制淘汰。

HFile的Data Block，Meta Block通常采用压缩方式存储，压缩之后可以大大减少网络IO和磁盘IO，随之而来的开销当然是需要花费cpu进行压缩和解压缩。

目标Hfile的压缩支持两种方式：Gzip，Lzo。

### 3、Memstore与storefile

一个region由多个store组成，每个store包含一个列族的所有数据

Store包括位于内存的memstore和位于硬盘的storefile

写操作先写入memstore,当memstore中的数据量达到某个阈值，Hregionserver启动flashcache进程写入storefile,每次写入形成单独一个storefile

当storefile大小超过一定阈值后，会把当前的region分割成两个，并由Hmaster分配给相应的region服务器，实现负载均衡

客户端检索数据时，先在memstore找，找不到再找storefile

### 4、HLog(WAL log)

WAL 意为Write ahead log(http://en.wikipedia.org/wiki/Write-ahead\_logging)，类似mysql中的binlog,用来 做灾难恢复时用，Hlog记录数据的所有变更,一旦数据修改，就可以从log中进行恢复。

每个Region Server维护一个Hlog,而不是每个Region一个。这样不同region(来自不同table)的日志会混在一起，这样做的目的是不断追加单个文件相对于同时写多个文件而言，可以减少磁盘寻址次数，因此可以提高对table的写性能。带来的麻烦是，如果一台region server下线，为了恢复其上的region，需要将region server上的log进行拆分，然后分发到其它region server上进行恢复。

HLog文件就是一个普通的Hadoop Sequence File：

* HLog Sequence File 的Key是HLogKey对象，HLogKey中记录了写入数据的归属信息，除了table和region名字外，同时还包括 sequence number和timestamp，timestamp是”写入时间”，sequence number的起始值为0，或者是最近一次存入文件系统中sequence number。
* HLog Sequece File的Value是HBase的KeyValue对象，即对应HFile中的KeyValue，可参见上文描述。

## 读写过程

### 1、读请求过程：

HRegionServer保存着meta表以及表数据，要访问表数据，首先Client先去访问zookeeper，从zookeeper里面获取meta表所在的位置信息，即找到这个meta表在哪个HRegionServer上保存着。

接着Client通过刚才获取到的HRegionServer的IP来访问Meta表所在的HRegionServer，从而读取到Meta，进而获取到Meta表中存放的元数据。

Client通过元数据中存储的信息，访问对应的HRegionServer，然后扫描所在HRegionServer的Memstore和Storefile来查询数据。

最后HRegionServer把查询到的数据响应给Client。

查看meta表信息

hbase(main):011:0> scan 'hbase:meta'

### 2、写请求过程：

Client也是先访问zookeeper，找到Meta表，并获取Meta表元数据。

确定当前将要写入的数据所对应的HRegion和HRegionServer服务器。

Client向该HRegionServer服务器发起写入数据请求，然后HRegionServer收到请求并响应。

Client先把数据写入到HLog，以防止数据丢失。

然后将数据写入到Memstore。

如果HLog和Memstore均写入成功，则这条数据写入成功

如果Memstore达到阈值，会把Memstore中的数据flush到Storefile中。

当Storefile越来越多，会触发Compact合并操作，把过多的Storefile合并成一个大的Storefile。

当Storefile越来越大，Region也会越来越大，达到阈值后，会触发Split操作，将Region一分为二。

细节描述：

hbase使用MemStore和StoreFile存储对表的更新。

数据在更新时首先写入Log(WAL log)和内存(MemStore)中，MemStore中的数据是排序的，当MemStore累计到一定阈值时，就会创建一个新的MemStore，并 且将老的MemStore添加到flush队列，由单独的线程flush到磁盘上，成为一个StoreFile。于此同时，系统会在zookeeper中记录一个redo point，表示这个时刻之前的变更已经持久化了。

当系统出现意外时，可能导致内存(MemStore)中的数据丢失，此时使用Log(WAL log)来恢复checkpoint之后的数据。

StoreFile是只读的，一旦创建后就不可以再修改。因此Hbase的更新其实是不断追加的操作。当一个Store中的StoreFile达到一定的阈值后，就会进行一次合并(minor\_compact, major\_compact),将对同一个key的修改合并到一起，形成一个大的StoreFile，当StoreFile的大小达到一定阈值后，又会对 StoreFile进行split，等分为两个StoreFile。

由于对表的更新是不断追加的，compact时，需要访问Store中全部的 StoreFile和MemStore，将他们按row key进行合并，由于StoreFile和MemStore都是经过排序的，并且StoreFile带有内存中索引，合并的过程还是比较快。

## Region管理

(1) region分配

任何时刻，一个region只能分配给一个region server。master记录了当前有哪些可用的region server。以及当前哪些region分配给了哪些region server，哪些region还没有分配。当需要分配的新的region，并且有一个region server上有可用空间时，master就给这个region server发送一个装载请求，把region分配给这个region server。region server得到请求后，就开始对此region提供服务。

(2) region server上线

master使用zookeeper来跟踪region server状态。当某个region server启动时，会首先在zookeeper上的server目录下建立代表自己的znode。由于master订阅了server目录上的变更消息，当server目录下的文件出现新增或删除操作时，master可以得到来自zookeeper的实时通知。因此一旦region server上线，master能马上得到消息。

(3) region server下线

当region server下线时，它和zookeeper的会话断开，zookeeper而自动释放代表这台server的文件上的独占锁。master就可以确定：

1 region server和zookeeper之间的网络断开了。

2 region server挂了。

无论哪种情况，region server都无法继续为它的region提供服务了，此时master会删除server目录下代表这台region server的znode数据，并将这台region server的region分配给其它还活着的同志。

## Master工作机制

* master上线

master启动进行以下步骤:

1 从zookeeper上获取唯一一个代表active master的锁，用来阻止其它master成为master。

2 扫描zookeeper上的server父节点，获得当前可用的region server列表。

3 和每个region server通信，获得当前已分配的region和region server的对应关系。

4 扫描.META.region的集合，计算得到当前还未分配的region，将他们放入待分配region列表。

* master下线

由于master只维护表和region的元数据，而不参与表数据IO的过程，master下线仅导致所有元数据的修改被冻结(无法创建删除表，无法修改表的schema，无法进行region的负载均衡，无法处理region 上下线，无法进行region的合并，唯一例外的是region的split可以正常进行，因为只有region server参与)，表的数据读写还可以正常进行。因此master下线短时间内对整个hbase集群没有影响。

从上线过程可以看到，master保存的信息全是可以冗余信息（都可以从系统其它地方收集到或者计算出来）

因此，一般hbase集群中总是有一个master在提供服务，还有一个以上的‘master’在等待时机抢占它的位置。

# 11、HBase三个重要机制

## 1、flush机制

1.（hbase.regionserver.global.memstore.size）默认;堆大小的40%

regionServer的全局memstore的大小，超过该大小会触发flush到磁盘的操作,默认是堆大小的40%,而且regionserver级别的flush会阻塞客户端读写

2.（hbase.hregion.memstore.flush.size）默认：128M

单个region里memstore的缓存大小，超过那么整个HRegion就会flush,

3.（hbase.regionserver.optionalcacheflushinterval）默认：1h

内存中的文件在自动刷新之前能够存活的最长时间

4.（hbase.regionserver.global.memstore.size.lower.limit）默认：堆大小 \* 0.4 \* 0.95

有时候集群的“写负载”非常高，写入量一直超过flush的量，这时，我们就希望memstore不要超过一定的安全设置。在这种情况下，写操作就要被阻塞一直到memstore恢复到一个“可管理”的大小, 这个大小就是默认值是堆大小 \* 0.4 \* 0.95，也就是当regionserver级别的flush操作发送后,会阻塞客户端写,一直阻塞到整个regionserver级别的memstore的大小为 堆大小 \* 0.4 \*0.95为止

5.（hbase.hregion.preclose.flush.size）默认为：5M

当一个 region 中的 memstore 的大小大于这个值的时候，我们又触发 了 close.会先运行“pre-flush”操作，清理这个需要关闭的memstore，然后 将这个 region 下线。当一个 region 下线了，我们无法再进行任何写操作。 如果一个 memstore 很大的时候，flush 操作会消耗很多时间。"pre-flush" 操作意味着在 region 下线之前，会先把 memstore 清空。这样在最终执行 close 操作的时候，flush 操作会很快。

6.（hbase.hstore.compactionThreshold）默认：超过3个

一个store里面允许存的hfile的个数，超过这个个数会被写到新的一个hfile里面 也即是每个region的每个列族对应的memstore在fulsh为hfile的时候，默认情况下当超过3个hfile的时候就会 对这些文件进行合并重写为一个新文件，设置个数越大可以减少触发合并的时间，但是每次合并的时间就会越长

## compact机制

[http://archive.cloudera.com/cdh5/cdh/5/hbase-1.2.0-cdh5.14.0/book.html#compaction](http://archive.cloudera.com/cdh5/cdh/5/hbase-1.2.0-cdh5.14.0/book.html" \l "compaction)

把小的storeFile文件合并成大的Storefile文件。

清理过期的数据，包括删除的数据

将数据的版本号保存为3个

## 3、split机制

当Region达到阈值，会把过大的Region一分为二。

默认一个HFile达到10Gb的时候就会进行切分

# 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

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

[http://archive.cloudera.com/cdh5/cdh/5/hbase-1.2.0-cdh5.14.0/book.html#mapreduce](http://archive.cloudera.com/cdh5/cdh/5/hbase-1.2.0-cdh5.14.0/book.html" \l "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

### 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

<repositories>

<repository>

<id>cloudera</id>

<url>https://repository.cloudera.com/artifactory/cloudera-repos/</url>

</repository>

</repositories>

<dependencies>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-client</artifactId>

<version>2.6.0-mr1-cdh5.14.0</version>

</dependency>

<dependency>

<groupId>org.apache.hbase</groupId>

<artifactId>hbase-client</artifactId>

<version>1.2.0-cdh5.14.0</version>

</dependency>

<dependency>

<groupId>org.apache.hbase</groupId>

<artifactId>hbase-server</artifactId>

<version>1.2.0-cdh5.14.0</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.testng</groupId>

<artifactId>testng</artifactId>

<version>6.14.3</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>maven-compiler-plugin</artifactId>

<version>3.0</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

<encoding>UTF-8</encoding>

<!-- <verbal>true</verbal>-->

</configuration>

</plugin>

<plugin>

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

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

<version>2.2</version>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>shade</goal>

</goals>

<configuration>

<filters>

<filter>

<artifact>\*:\*</artifact>

<excludes>

<exclude>META-INF/\*.SF</exclude>

<exclude>META-INF/\*.DSA</exclude>

<exclude>META-INF/\*/RSA</exclude>

</excludes>

</filter>

</filters>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

### 第三步：开发MR的程序

public class HBaseMR extends Configured implements Tool{

public static class HBaseMapper extends TableMapper<Text,Put>{

/\*\*

\*

\* @param key 我们的主键rowkey

\* @param value 我们一行数据所有列的值都封装在value里面了

\* @param context

\* @throws IOException

\* @throws InterruptedException

\*/

@Override

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

byte[] bytes = key.get();

String rowKey = Bytes.toString(bytes);

Put put = new Put(key.get());

Cell[] cells = value.rawCells();

for (Cell cell : cells) {

if("f1".equals(Bytes.toString(CellUtil.cloneFamily(cell)))){

if("name".equals(Bytes.toString(CellUtil.cloneQualifier(cell)))){

put.add(cell);

}

if("age".equals(Bytes.toString(CellUtil.cloneQualifier(cell)))){

put.add(cell);

}

}

}

if(!put.isEmpty()){

context.write(new Text(rowKey),put);

}

}

}

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

@Override

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

for (Put value : values) {

context.write(null,value);

}

}

}

@Override

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

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

job.setJarByClass(this.getClass());

Scan scan = new Scan();

scan.setCaching(500);

scan.setCacheBlocks(false);

//使用TableMapReduceUtil 工具类来初始化我们的mapper

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

//使用TableMapReduceUtil 工具类来初始化我们的reducer

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 {

//创建HBaseConfiguration配置

Configuration configuration = HBaseConfiguration.create();

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

System.exit(run);

}

}

### 第四步：打包运行

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

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

export HADOOP\_HOME=/export/servers/hadoop-2.6.0-cdh5.14.0/

export HBASE\_HOME=/export/servers/hbase-1.2.0-cdh5.14.0/

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内容如下

0007 zhangsan 18

0008 lisi 25

0009 wangwu 20

### 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

### 第二步：开发MR程序

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：作业，读取HBase的表数据，然后将数据写入到hdfs上面去

## 需求四：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资料导致效率低下，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

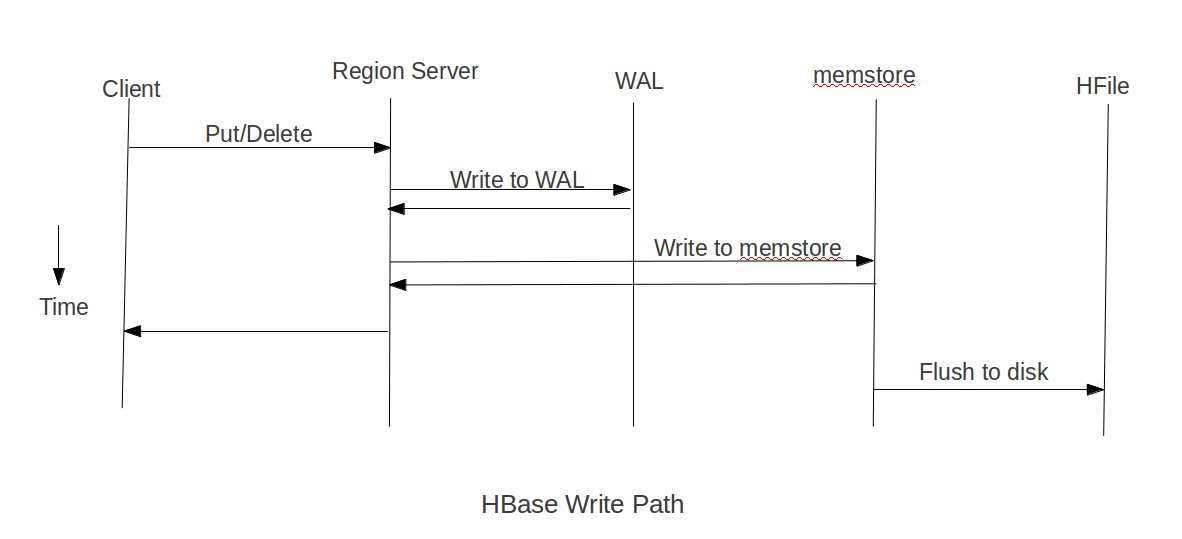
优点：

1.导入过程不占用Region资源

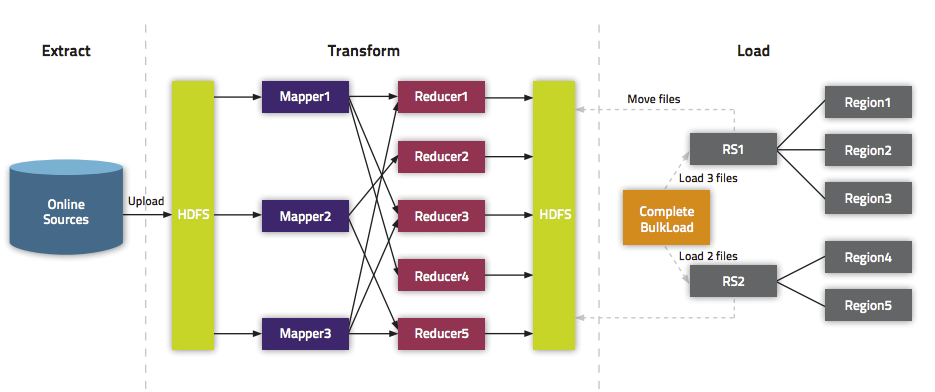
2.能快速导入海量的数据

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

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

@Override

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

String[] split = value.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);

}

}

### 第二步：开发我们的main程序入口类

public class HBaseLoad extends Configured implements Tool {

@Override

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

final String INPUT\_PATH= "hdfs://node01:8020/hbase/input";

final String OUTPUT\_PATH= "hdfs://node01:8020/hbase/output\_hfile";

Configuration conf = HBaseConfiguration.create();

Connection connection = ConnectionFactory.createConnection(conf);

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

Job job= Job.getInstance(conf);

job.setJarByClass(HBaseLoad.class);

job.setMapperClass(LoadMapper.class);

job.setMapOutputKeyClass(ImmutableBytesWritable.class);

job.setMapOutputValueClass(Put.class);

job.setOutputFormatClass(HFileOutputFormat2.class);

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

FileInputFormat.addInputPath(job,new Path(INPUT\_PATH));

FileOutputFormat.setOutputPath(job,new Path(OUTPUT\_PATH));

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 HBaseLoad(), 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-1.2.0-cdh5.14.0/

export HADOOP\_HOME=/export/servers/hadoop-2.6.0-cdh5.14.0/

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-1.2.0-cdh5.14.0/lib/hbase-server-1.2.0-cdh5.14.0.jar completebulkload /hbase/output\_hfile myuser2

# 13、HBase与hive的对比

## Hive

### 数据仓库

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。（不要钻不需要执行MapReduce代码的情况的牛角尖）

## HBase

### 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-1.2.0-cdh5.14.0/lib

我们需要拷贝五个jar包名字如下

hbase-client-1.2.0-cdh5.14.0.jar

hbase-hadoop2-compat-1.2.0-cdh5.14.0.jar

hbase-hadoop-compat-1.2.0-cdh5.14.0.jar

hbase-it-1.2.0-cdh5.14.0.jar

hbase-server-1.2.0-cdh5.14.0.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-1.2.0-cdh5.14.0/lib/hbase-client-1.2.0-cdh5.14.0.jar /export/servers/hive-1.1.0-cdh5.14.0/lib/hbase-client-1.2.0-cdh5.14.0.jar

ln -s /export/servers/hbase-1.2.0-cdh5.14.0/lib/hbase-hadoop2-compat-1.2.0-cdh5.14.0.jar /export/servers/hive-1.1.0-cdh5.14.0/lib/hbase-hadoop2-compat-1.2.0-cdh5.14.0.jar

ln -s /export/servers/hbase-1.2.0-cdh5.14.0/lib/hbase-hadoop-compat-1.2.0-cdh5.14.0.jar /export/servers/hive-1.1.0-cdh5.14.0/lib/hbase-hadoop-compat-1.2.0-cdh5.14.0.jar

ln -s /export/servers/hbase-1.2.0-cdh5.14.0/lib/hbase-it-1.2.0-cdh5.14.0.jar /export/servers/hive-1.1.0-cdh5.14.0/lib/hbase-it-1.2.0-cdh5.14.0.jar

ln -s /export/servers/hbase-1.2.0-cdh5.14.0/lib/hbase-server-1.2.0-cdh5.14.0.jar /export/servers/hive-1.1.0-cdh5.14.0/lib/hbase-server-1.2.0-cdh5.14.0.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/hive-1.1.0-cdh5.14.0/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/hive-1.1.0-cdh5.14.0/conf

vim hive-env.sh

export HADOOP\_HOME=/export/servers/hadoop-2.6.0-cdh5.14.0

export HBASE\_HOME=/export/servers/hbase-1.2.0-cdh5.14.0

export HIVE\_CONF\_DIR=/export/servers/hive-1.1.0-cdh5.14.0/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/hive-1.1.0-cdh5.14.0/

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

加载数据格式如下

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，直接通过

### 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、Sqoop整合HBase

sqoop是一个数据导入导出的工具，可以将关系型数据库当中的数据导入到大数据平台来，也可以将大数据平台当中的数据导入到关系型数据库当中去

我们也可以通过sqoop导入数据到hbase或者从hbase当中导出数据

## 需求一：将mysql表当中的数据导入到HBase当中来

### 第一步：修改sqoop配置文件

sqoop导入导出HBase的数据，需要修改sqoop的配置文件sqoop-env.sh

cd /export/servers/sqoop-1.4.6-cdh5.14.0/conf

vim sqoop-env.sh

#Set path to where bin/hadoop is available

export HADOOP\_COMMON\_HOME=/export/servers/hadoop-2.6.0-cdh5.14.0

#Set path to where hadoop-\*-core.jar is available

export HADOOP\_MAPRED\_HOME=/export/servers/hadoop-2.6.0-cdh5.14.0

#set the path to where bin/hbase is available

export HBASE\_HOME=/export/servers/hbase-1.2.0-cdh5.14.0

#Set the path to where bin/hive is available

export HIVE\_HOME=/export/servers/hive-1.1.0-cdh5.14.0

### 第二步：在mysql当中创建数据库以及数据库表并插入数据

#### 创建数据库表

CREATE DATABASE IF NOT EXISTS library;

USE library;

CREATE TABLE book(

id INT(4) PRIMARY KEY NOT NULL AUTO\_INCREMENT,

NAME VARCHAR(255) NOT NULL,

price VARCHAR(255) NOT NULL);

#### 插入数据

INSERT INTO book(NAME, price) VALUES('Lie Sporting', '30');

INSERT INTO book (NAME, price) VALUES('Pride & Prejudice', '70');

INSERT INTO book (NAME, price) VALUES('Fall of Giants', '50');

### 第三步：将mysql表当中的数据导入HBase表当中去

执行以下命令，将mysql表当中的数据导入到HBase当中去

bin/sqoop import \

--connect jdbc:mysql://192.168.1.5:3306/library \

--username root \

--password admin \

--table book \

--columns "id,name,price" \

--column-family "info" \

--hbase-create-table \

--hbase-row-key "id" \

--hbase-table "hbase\_book" \

--num-mappers 1 \

--split-by id

### 第四步：HBase当中查看表数据

进入hbase的shell客户端，通过scan查看数据

hbase(main):057:0> scan 'hbase\_book'

ROW COLUMN+CELL

1 column=info:name, timestamp=1550634017823, value=Lie Sporting

1 column=info:price, timestamp=1550634017823, value=30

2 column=info:name, timestamp=1550634017823, value=Pride & Prejudice

2 column=info:price, timestamp=1550634017823, value=70

3 column=info:name, timestamp=1550634017823, value=Fall of Giants

3 column=info:price, timestamp=1550634017823, value=50

## 需求二：将HBase当中的数据导出到mysql当中来

将hbase\_book这张表当中的数据导出到mysql当中来

注意：sqoop不支持我们直接将HBase当中的数据导出，所以我们可以通过以下的转换进行导出

Hbase→hive外部表→hive内部表→通过sqoop→mysql

### 第一步：创建hive外部表

进入hive客户端，创建hive外部表，映射hbase当中的hbase\_book表

CREATE EXTERNAL TABLE course.hbase2mysql (id int,name string,price int)

STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'

WITH SERDEPROPERTIES (

"hbase.columns.mapping" =

":key,info:name, info:price"

)

TBLPROPERTIES( "hbase.table.name" = "hbase\_book",

"hbase.mapred.output.outputtable" = "hbase2mysql");

### 第二步：创建hive内部表并将外部表数据插入到内部表当中来

进入hive客户端，执行以下命令，创建hive内部表，并将外部表的数据插入到hive的内部表当中来

CREATE TABLE course.hbase2mysqlin(id int,name string,price int);

### 第三步：外部表数据插入内部表

进入hive客户端执行以下命令，将hive外部表数据插入到hive内部表当中来

insert overwrite table course.hbase2mysqlin select \* from course.hbase2mysql;

### 第四步：清空mysql表数据

进入mysql客户端，执行以下命令，将mysql表数据清空

TRUNCATE TABLE book;

### 第五步：执行sqoop导出hive内部表数据到

sqoop export -connect jdbc:mysql://192.168.1.5:3306/library -username root -password admin -table book -export-dir /user/hive/warehouse/course.db/hbase2mysqlin --input-fields-terminated-by '\001' --input-null-string '\\N' --input-null-non-string '\\N';

# 16、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？

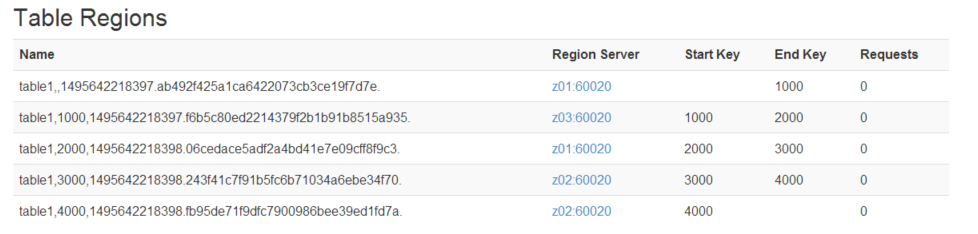
每一个region维护着startRow与endRowKey，如果加入的数据符合某个region维护的rowKey范围，则该数据交给这个region维护。

## 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']

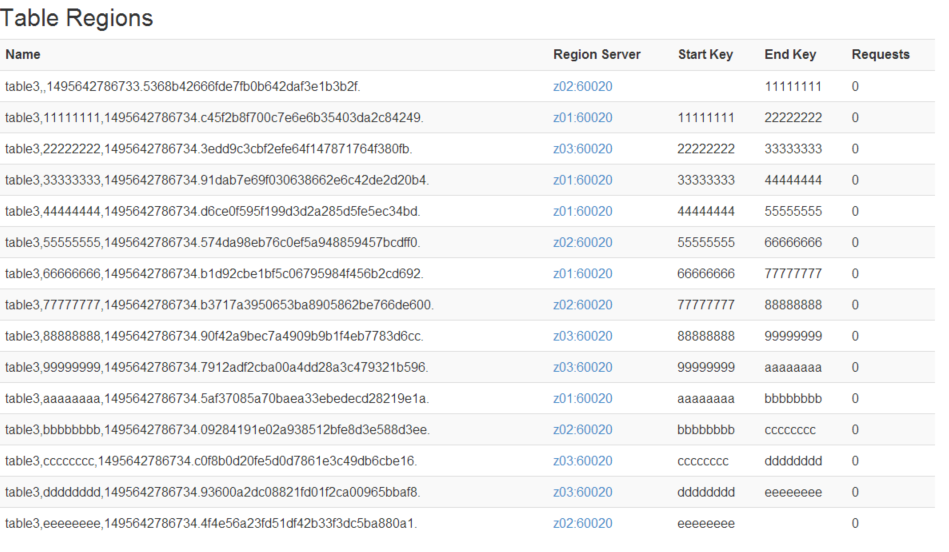
完成后如图：



### 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、分区规则创建于文件中

创建splits.txt文件内容如下：

cd /export/servers/

vim splits.txt

aaaa

bbbb

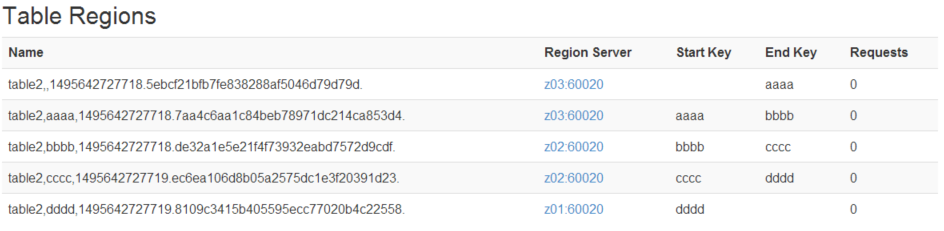
cccc

dddd

然后执行：

hbase(main):004:0> create 'staff3','partition2',SPLITS\_FILE => '/export/servers/splits.txt'

成功后如图：



### 4、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("stuff4"));

//添加列族

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

//添加列族

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

admin.createTable(hTableDescriptor,splitKeys);

admin.close();

}

# 17、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。

# 18、HBase的协处理器

[http://hbase.apache.org/book.html#cp](http://hbase.apache.org/book.html" \l "cp)

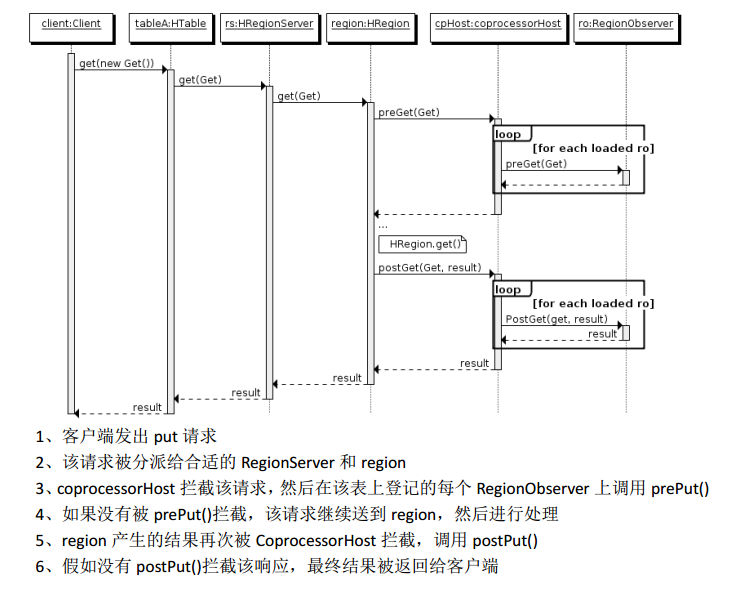
## 1、 起源 Hbase 作为列族数据库最经常被人诟病的特性包括：无法轻易建立“二级索引”，难以执 行求和、计数、排序等操作。比如，在旧版本的(<0.92)Hbase 中，统计数据表的总行数，需 要使用 Counter 方法，执行一次 MapReduce Job 才能得到。虽然 HBase 在数据存储层中集成 了 MapReduce，能够有效用于数据表的分布式计算。然而在很多情况下，做一些简单的相 加或者聚合计算的时候， 如果直接将计算过程放置在 server 端，能够减少通讯开销，从而获 得很好的性能提升。于是， HBase 在 0.92 之后引入了协处理器(coprocessors)，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

## 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

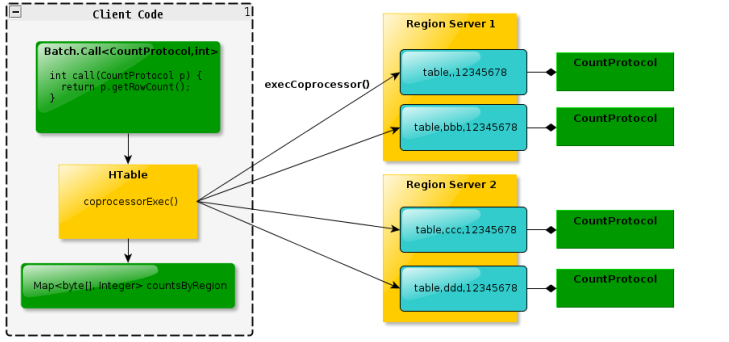
以 HBase0.92 版本为例，它提供了三种观察者接口：  
● RegionObserver：提供客户端的数据操纵事件钩子： Get、 Put、 Delete、 Scan 等。  
● WALObserver：提供 WAL 相关操作钩子。  
● MasterObserver：提供 DDL-类型的操作钩子。如创建、删除、修改数据表等。  
到 0.96 版本又新增一个 RegionServerObserver

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>

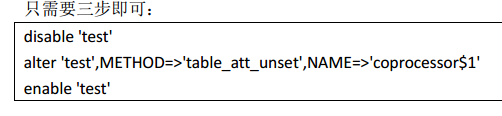
</property>

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

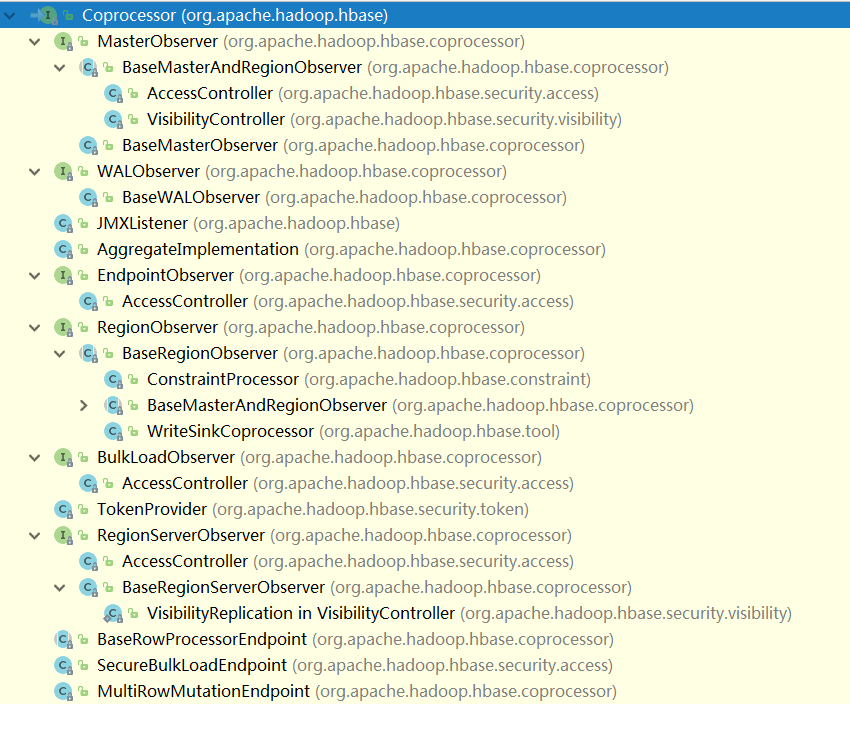
    2、动态加载

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



## 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-1.2.0-cdh5.14.0/

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user3，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user3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

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.Cell;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.TableName;

import org.apache.hadoop.hbase.client.\*;

import org.apache.hadoop.hbase.coprocessor.BaseRegionObserver;

import org.apache.hadoop.hbase.coprocessor.ObserverContext;

import org.apache.hadoop.hbase.coprocessor.RegionCoprocessorEnvironment;

import org.apache.hadoop.hbase.regionserver.wal.WALEdit;

import java.io.IOException;

public class MyProcessor extends BaseRegionObserver {

@Override

public void prePut(ObserverContext<RegionCoprocessorEnvironment> e, Put put, WALEdit edit, Durability durability) throws IOException {

//获取连接

Configuration configuration = HBaseConfiguration.create();

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

Connection connection = ConnectionFactory.createConnection(configuration);

Cell nameCell = put.get("info".getBytes(), "name".getBytes()).get(0);

Put put1 = new Put(put.getRow());

put1.add(nameCell);

Table reverseuser = connection.getTable(TableName.valueOf("proc2"));

reverseuser.put(put1);

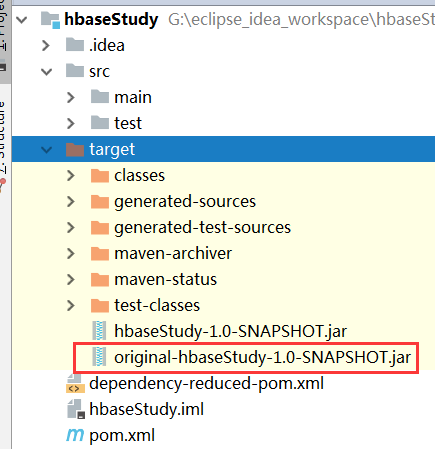
reverseuser.close();

}

}

### 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-hbaseStudy-1.0-SNAPSHOT.jar processor.jar

hdfs dfs -mkdir -p /processor

hdfs dfs -put processor.jar /processor

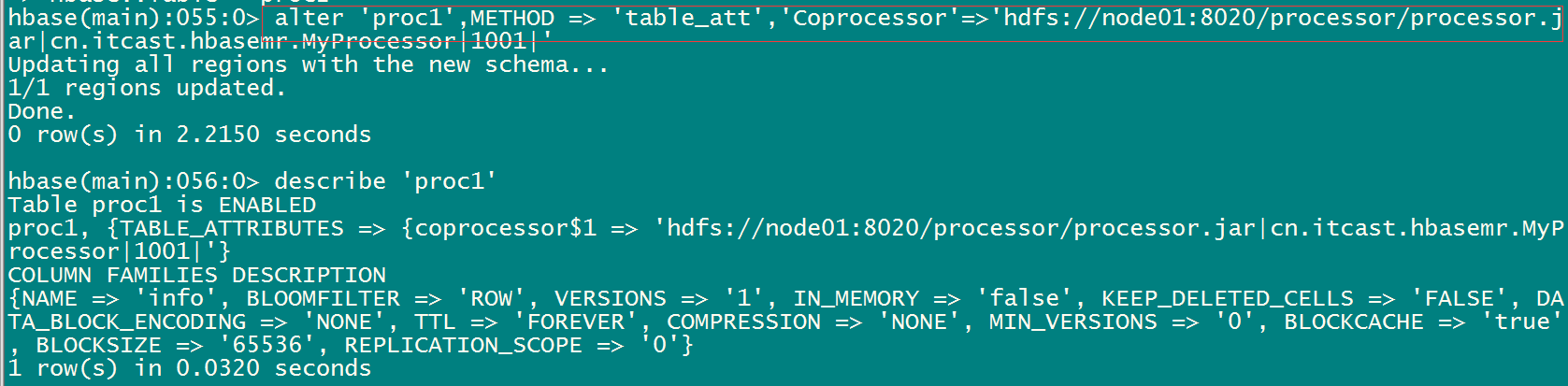
### 第五步：将打好的jar包挂载到proc1表当中去

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.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

/\*\*

\*

\*/

@Test

public void testPut() throws Exception{

//获取连接

Configuration configuration = HBaseConfiguration.create();

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

Connection connection = ConnectionFactory.createConnection(configuration);

Table user5 = connection.getTable(TableName.valueOf("proc1"));

Put put1 = new Put(Bytes.toBytes("hello\_world"));

put1.addColumn(Bytes.toBytes("info"),"name".getBytes(),"helloworld".getBytes());

put1.addColumn(Bytes.toBytes("info"),"gender".getBytes(),"abc".getBytes());

put1.addColumn(Bytes.toBytes("info"),"nationality".getBytes(),"test".getBytes());

user5.put(put1);

byte[] row = put1.getRow();

System.out.println(Bytes.toString(row));

user5.close();

}

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'

# 19、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

# 20、HBase整合hue

## 1、Hue的介绍

HUE=**Hadoop User Experience**

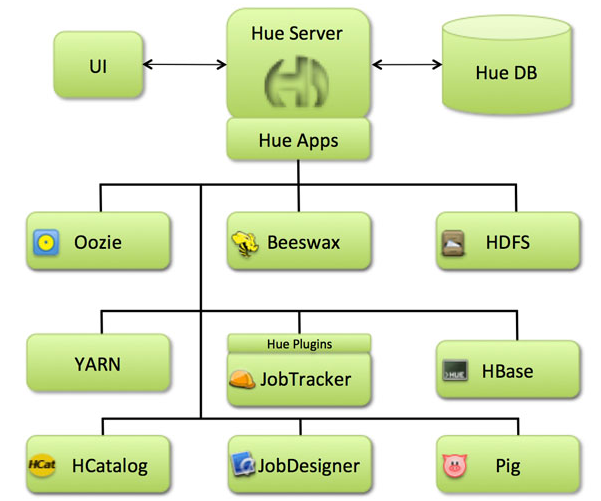
Hue是一个开源的Apache Hadoop UI系统，由Cloudera Desktop演化而来，最后Cloudera公司将其贡献给Apache基金会的Hadoop社区，它是基于Python Web框架Django实现的。

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/](http://gethue.com/" \t "_blank)
* Github: [https://github.com/cloudera/hue](https://github.com/cloudera/hue" \t "_blank)
* Reviews: [https://review.cloudera.org](https://review.cloudera.org/" \t "_blank)

### 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Hue的压缩包并上传到linux解压

Hue的压缩包的下载地址：

<http://archive.cloudera.com/cdh5/cdh/5/>

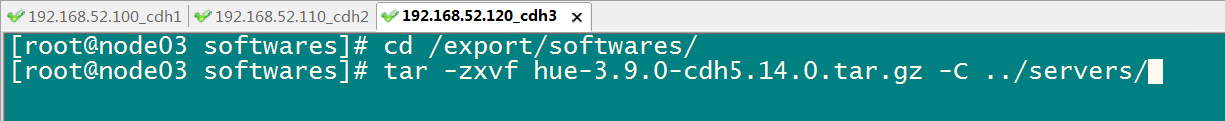
我们这里使用的是CDH5.14.0这个对应的版本，具体下载地址为

<http://archive.cloudera.com/cdh5/cdh/5/hue-3.9.0-cdh5.14.0.tar.gz>

下载然后上传到linux系统，然后进行解压

cd /export/softwares/

tar -zxvf hue-3.9.0-cdh5.14.0.tar.gz -C ../servers/



### 第二步：编译安装启动

#### 2.1、linux系统安装依赖包：

联网安装各种必须的依赖包

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

#### 2.2、开始配置Hue

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

is\_hue\_4=true

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587行左右

[[database]]

engine=mysql

host=node03.hadoop.com

port=3306

user=root

password=123456

name=hue

#### 2.3、创建mysql数据库

创建hue数据库

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

注意：实际工作中，还需要为hue这个数据库创建对应的用户，并分配权限，我这就不创建了，所以下面这一步不用执行了

grant all on hue.\* to 'hue'@'%' identified by 'hue';

#### 2.4、准备进行编译

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

make apps

#### 2.5、linux系统添加普通用户hue

useradd hue

passwd hue

#### 2.6、启动hue进程

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

build/env/bin/supervisor

#### 2.7、页面访问

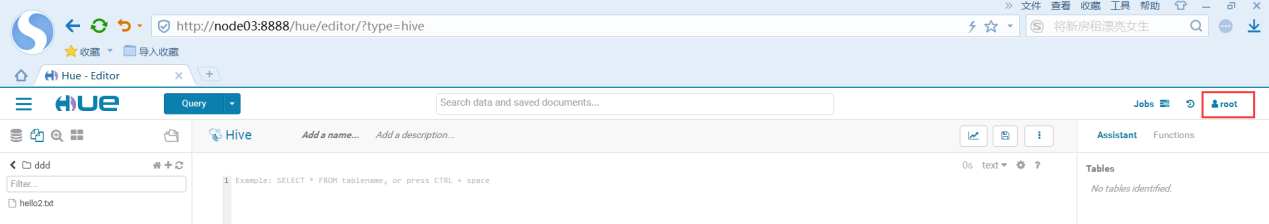
<http://node03:8888>

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-2.6.0-cdh5.14.0

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-2.6.0-cdh5.14.0

hadoop\_bin=/export/servers/hadoop-2.6.0-cdh5.14.0/bin

hadoop\_conf\_dir=/export/servers/hadoop-2.6.0-cdh5.14.0/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metastore服务以及hiveserver2服务（impala需要hive的metastore服务，hue需要hvie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/hive-1.1.0-cdh5.14.0/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metastore以及hiveserver2服务

cd /export/servers/hive-1.1.0-cdh5.14.0

nohup bin/hive --service metastore & 这个是用于与impala整合，如果不与imapla整合，不需要启动这个服务

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

### 3.3、配置hue与impala的集成

停止hue的服务进程

修改hue.ini配置文件

[impala]

server\_host=node03

server\_port=21050

impala\_conf\_dir=/etc/impala/conf

### 3.4、配置hue与mysql的集成

找到databases 这个选项，将这个选项下面的mysql注释给打开，然后配置mysql即可,大概在1547行

[[[mysql]]]

nice\_name="My SQL DB"

engine=mysql

host=node03.hadoop.com

port=3306

user=root

password=123456

重新启动hue的服务

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

build/env/bin/supervisor

### 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-1.2.0-cdh5.14.0/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-1.2.0-cdh5.14.0

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/>

# 21、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、储存新强、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：

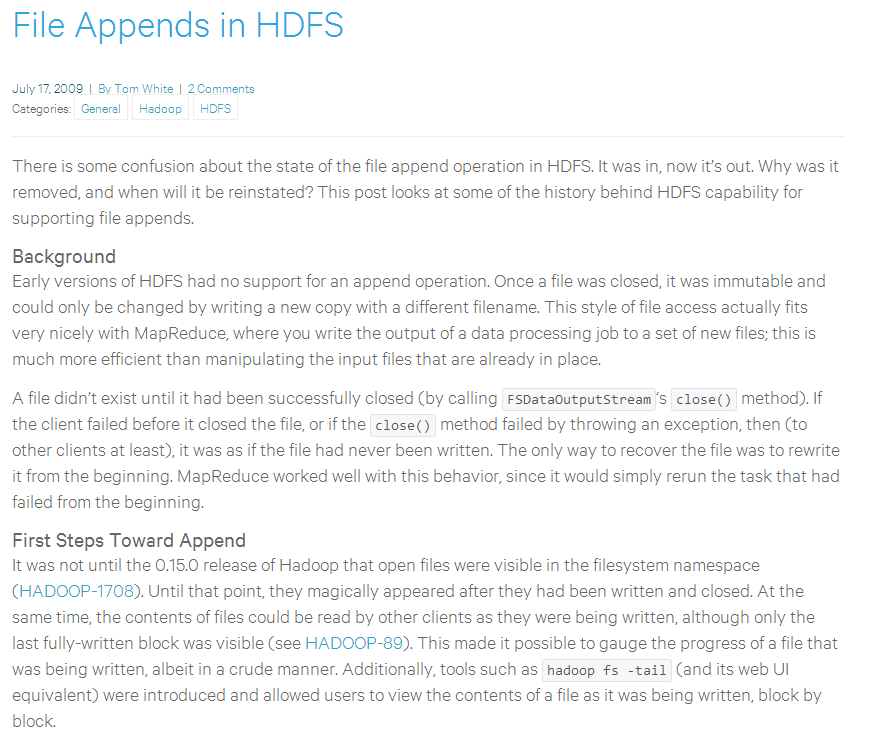
map = 2 + ⅔cpu\_core

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。

# 总结

hbase第一天

redis课程内容回顾

1、web系统变迁 通过redis可以实现session共享，实现快速存取的需求

2、nosql的产品

memcache：

redis：已经替代了memcache 所有的数据都是存放在内存里面的

mongodb

hbase：nosql数据库

3、redis的基本介绍以及使用场景

redis是nosql数据库，数据都是存放在内存里面的，决定了redis里面不可能存放海量数据

redis的适用场景

list：可以用于构建队列

redis的特点：存取速度比较快

4、单机版本redis的环境安装 ==》 gcc-c++ tcl

5、redis当中的数据类型

string：字符串 ，可以做计数器

set

get

incr

decr

incrby

decrby

expire

hash： key field:value

hset

hget

hkeys

hvals

list：类似于 ArrayList

lpush

rpush

lpop

rpop

rpoplpush

set：对数据进行去重操作

sadd

smembers

sdiff 求差集

sunion 求并集

zset： 类似于set集合，但是会对数据进行排序 可以用于求取topN的操作

6、redis的javaAPI操作

获取数据库连接池

7、redis当中持久化操作

RDB：默认开启 save seconds numsKeys 每隔多少秒有多少个key发生变化，就进行数据保存

将redis数据库当中的数据dump一份到磁盘文件里面保存

AOF：默认关闭 一般都会打开aof这种持久化方式 每秒钟都会保存客户端的操作命令

为了防止操作命令文件越来越大，定期的fork 一个子进程 来清理日志文件，防止文件过大

appendonly yes

fsync everySecond

缓存击穿：没有命中redis当中的缓存数据，请求获取数据直接去查询数据库

缓存雪崩：redis当中的数据一瞬间全部失效了

redis主从复制：可以实现一个主节点，多个从节点，实现读写分离

redis哨兵模式：解决了主节点单点故障宕机问题导致不能写入数据

redis内存横向扩展问题：redis集群操作

所有的数据库都是讲增删改查，只不过都是重点讲查询

1、hbase基本介绍 了解

2、hbase与hadoop关系 了解

3、rdbms与hbase对比 了解

4、hbase的简要特征

5、hbase的基础架构

6、hbase集群环境搭建

7、通过hbase shell命令来实现hbase的操作

8、hbase的高级shell管理命令

9、hbase的JavaAPI开发

10、hbase的底层实现原理

11、hbase的三个重要机制

1、hbase的基本介绍：

大数据领域里面一个非惯性数据库，与redis类似，也是一个nosql数据库

hbase不支持sql语句的查询

hbase是建立在hdfs之上的一个非关系型数据库，hbase的数据最终都是存储在hdfs上面的

仅仅支持通过行键 rowkey来进行检索数据 ，类似于redis当中key ，类似于mysql当中数据主键

主要用于存储结构化和半结构化松散数据

hbase的基本特点

表比较大：可以存储上十亿条数据，可以有上百万列（字段可以有上百万个）

面向列：使用的是列式存储的方式

稀疏表：如果某条数据没有某个字段的值，这个字段不会占用磁盘空间

rowkey name age

1 zhangsan ==》 age这个字段在第一条数据里面不会占用磁盘空间

2 28 ==>name这个字段在第二条数据里面不会占用磁盘空间

hbase是建立在hdfs上面的，hbase的数据都是存储在hdfs上面了

hbase数据库：支持实时的增删改查，所有的数都是保存在hdfs上面的

hdfs：分布式文件存储系统，擅长一次写入，多次读取

hbase适合做实时的增删改查，数据都是保存在hdfs上面的，hdfs又不适合做实时的增删改查的操作，

他们之间是如何协作的？？？？？

hbase依赖于hdfs，与hadoop的关系：依赖关系 紧耦合关系

使用hbase的话，需要启动hadoop hbase还需要依赖于zk，一定要保证hadoop以及zk正常启动

关系型数据库：最大的特点就是支持事务操作

hbase：非关系型数据库

hbase不支持事务：不支持连续操作多条数据的事务性，hbase也支持单条数据操作的事务性

操作hbase当中一条数据，要么成功，要么失败，所以hbase支持的是行级别事务

hbase不支持多张表join操作，不支持sql语句的查询

4、hbase的简要特征：

1、海量数据存储，hbase是大数据领域里面一个非关系型数据库，适合存储海量数据

2、列式存储：数据都是面向列进行存储的

3、易于扩展 如果hbase数据存不下了，加datanode节点即可

4、hbase支持客户端的高并发请求

5、稀疏表

rowkey name age

1 zhangsan ==》 age这个字段在第一条数据里面不会占用磁盘空间

2 28 ==>name这个字段在第二条数据里面不会占用磁盘空间

hbase主要的架构：

主从架构

HMaster：主节点 管理从节点HRegionServer 分配region

HRegionServer：从节点 管理region

一个HRegionServer可以分为两个大的部分

一个HLog：预写日志

多个region：就是用于存储数据

一个region：多个store模块组成

一个store模块：一个memoryStore和多个StoreFile

HLog：预写日志

memoryStore：内存空间128M

storeFile：文件，memoryStore溢写的文件

HFile：最终storeFile都会合并到HFile里面去

region：数据存储单位

store模块：一个store模块对应hbase的一个列族

hbase的表模型：

rowkey：行键，类似于redis当中key，每条不同的数据，行键都是不一样的，类似于mysql表的主键

列族：columnFamily 每一个列族下面对应可以有很多个列 每一个列族都是对应一个store模块

列：column 每个列都对应唯一的一个列族，一个列族下面可以动态的扩展很多列。

列名是我们在插入数据的时候动态指定的

一张hbas的表当中，列族不需要有很多个，一般3-5个最多了

timestamp：时间戳，每个数据都会有时间戳的概念

versionnum：版本号，每条数据都会有版本号的概念

创建hbase表的时候，最小需要指定两个条件，表名和列族名

其他的东西例如rowkey，列名，以及时间戳，以及版本号，都是在我们插入数据的时候，动态生成的

定位一个列值：rowkey + 列族 + 列名 cell值

如果某些数据没有其他的字段，不会占据存储空间，就叫做稀疏表

向hbas当中添加数据

put 'user','rk0001','info:age',20

hbase当中的数据查询操作：

查询只有三种方式：

第一种通过rowkey直接进行get get rowkey

第二种方式通过scan来进行扫描 scan startrowkey stoprowkey 加上rowkey范围值进行范围扫描

hbase当中的rowkey都是按照字典顺序进行排序的

第三种方式：scan table\_name 全表扫描 没人用

使用过滤器进行查询

列值过滤器 ValueFilter：为zhangsan

get 'user', 'rk0001', {FILTER => "ValueFilter(=, 'binary:zhangsan')"}

列名过滤器：QualifierFilter 查询哪些列名包含 a

get 'user', 'rk0001', {FILTER => "(QualifierFilter(=,'substring:a'))"}

hbase当中所有的数据都是字节数组类型的

使用get直接命令rowkye查询效率最高 就算hbase当中有上十亿条数据，使用get查询，hbase可以做到秒级返回

rowkey前缀过滤器PrefixFilter

scan 'user',{FILTER=>"PrefixFilter('rk')"}

hbase当中数据的更新操作

更新的操作与插入操作一模一样，如果rowkey存在就更新，如果rowkey不存在，就插入数据

truncate 'user' 清空表数据：

第一步：禁用表

第二步：删除表

第三步：创建表

hbase当中的过滤器的查询：

比较过滤器：主要是比较列名，列值等等

专用过滤器：专门用于过滤一些数据

比较过滤器：

RowFilter：rowkey的过滤器

FamilyFilter：列族过滤器

QualifierFilter：列名过滤器

ValueFilter：列值过滤器

专用过滤器：

SingleColumnValueFilter：单列值过滤器 与valueFilter不同，会返回符合条件数据的所有字段的值

prefixFilter：前缀过滤器

pageFilter：分页过滤器

FilterList

hbase架构：

主从架构：

HMaster：主节点

HRegionServer：从节点

一个HLog + 多个Region

一个region ： 多个Store模块 一个store模块对应一个列族

一个store模块：一个memoryStore和多个StoreFile

rowkey是一个字符串，最大长度64KB，实际工作当中，rowkey一般为10-100个字节

数据的物理存储：

HLog：预写日志，存储预写日志数据

memoryStore：内存空间128M 可以存储数据

storeFile：将memoryStore flush的数据都是存储在storeFile里面了

一个storeFile就是一个小的HFile这种存储格式

HFile：最终storeFile都会合并到HFile里面去

HLog预写日志可以关闭掉，可以提高写入的速度，但是同时也有丢失数据的风险

region，store：逻辑划分

一张表：一个HLog + 多个Region

hbase：meta表：一个Hlog + 一个region

regionServer 专门管理region

hbase当中数据写入：

第一步：客户端连接zk，获取一张特殊表 hbase:meta表位置

第二步：连接hbase:meta表所在的regionServer，读取HBase:meta表数据

这个表里面存放了其他表的元数据信息

第三步：获取对应表所在region对应的regionServer的位置

第四步：连接对应的regionServer

第五步：将数据写入到Hlog，然后再写入到MemoryStore，这两个地方写入成功，就算写入成功了

数据写入到memoryStore里面去了，写满了flush到StoreFile里面去

很多storeFile进行合并 compact过程 ==》 清理过期的数据，删除无效的数据 合并成为一个大的Hfile

HFile达到阈值10GB，就会自动的进行region切分，一个region切分成为两个，HFile也会一分为二

hbase数据的读取过程：

第一步：先连接zk，获取hbase:meta表位置

第二步：连接HregionServer读取hbase:meta表数据

第三步：与对应的regionServer进行通信

第四步：进行数据查询：先查询memoryStore，如果memoryStore里面没有了，再去查询StoreFIle，如果StoreFile里面还是没有，就去查询HFile 使用的是布隆过滤器

bloomfilter

hbase支持高并发请求操作

region的管理：

region分配：HMaster负责分配region

regionServer上线：感知 都是通过zk来实现的

regionServer下线：感知 都是通过zk来实现的

HMaster必须得要知道有哪些可用的regionServer

zookeeper在很多系统当中都有用到，zookeeper至关重要软件

一旦zk宕机，很多其他的框架都没法正常运行了，

如何监控zk：taoKepper 监测zk正常运行的框架 阿里开源的

master工作机制：

master首先上线，先去zk当中进行注册，抢一把锁 成为活跃的节点，其他的主节点启动

都是standBy状态节点

hbase三个重要机制：

flush：数据从memoryStore到storeFile

compact：数据从storeFile到Hfile minor major

主要是用于清理过期的数据，去掉删除的数据等等

split：将大的HFile一分为二，region也会一分为二

HFile会越来越大，一旦HFile超过10GB，就会进行拆分，一分为二

一个region会分裂成为两个region，一个HFile会拆分成为两个HFile

课程总结：除了第四章以前的，后面的全部搞定

hbase第二天

课程内容回顾：

1、hbase的基本介绍 大数据领域里面一个nosql数据库

2、hbase与hadoop关系 紧耦合关系，hbase需要依赖于hadoop

3、RDBMS与hbase之间的对比

RDBMS：关系型数据库，主要支持事务处理

hbase：非关系型数据库，不支持多行数据的事务处理，只支持行级别事务

4、HBase的基本架构：

HMaster：主节点

负责管理整个集群

region的分配

HRegionServer：从节点

管理region

HRegionServer = 1个HLog + 多个region

1个region = 多个store模块

1个store模块 = 1个memoryStore 128M内存空间 + 多个storeFile

memoryStore写满了之后，将数据flush到storeFile里面去

很多store会进行compact机制，将数据合并到HFile里面去

HFile存放在hdfs里面，为了避免hfile越来越大，达到10GB的时候，HFile就会进行split机制

5、HBase的集群环境搭建

6、hbase的基本shell命令以及高级shell管理命令

7、HBase的JavaAPI开发 数据库增删改查操作

8、hbase的过滤器 比较过滤器 专用过滤器

9、hbase的底层实现原理

hbase的读写流程：

写入流程：

第一步：客户端发起写入数据请求，连接zk

第二步：获取zk当中一张特殊表 hbase:meta表的位置信息

第三步：连接hbase：meta表所在的regionServer服务器，与之进行通信，读取hbase:meta表数据

这张表里面都是存储了元数据信息

第四步：与对应的表的regionServer进行通信，准备写入数据

第五步：数据先写入到HLog，然后将数据写入到memoryStore里面去，这两个地方数据写入成功，就算数据写入成功

memoryStore写满了之后，就会冲洗开辟一个新的memoryStore，老的memoryStore会加入到flush队列里面去，等待flush

storeFile也会进行compact机制 ==》 将数据进行合并，删除掉无用的数据，清理过期的数据

split机制：region分裂 将HFile一分为二，而且region也会一分为二，region的分裂，不会分到其他的机器上面去

读取流程：

第一步：客户端发起请求，读取数据，连接zk

第二步：获取zk当中一张特殊表hbase:meta表数据

第三步：与对应表所在的regionServer进行通信，进行数据查询

先查询memoryStore，如果memoryStore里面没有数据，继续查询storeFile，继续查询HFile

读取或者写入数据：全程没有HMaster的参与

10、hbase当中三个重要机制

flush机制：数据从memoryStore到storeFile

compact机制：数据从storeFile到HFile

split机制：将HFile一分为二

今日课程大纲

1、hbase与mr的集成

2、hbase与hive的整合

3、hbase与sqoop的集成

4、hbase的预分区操作

5、hbase的rowkey设计技巧

6、hbase的协处理器

7、hbase的二级索引介绍

8、hbase整合hue

9、hbase的一些基本调优

1、hbase与mr的集成

可以通过mr直接去读取hbase当中的数据，进行处理，并且我们可以通过mr将处理完成之后的数据写入到hbase里面去

需求：

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

读取hbase一张表数据，写入到hbase另外一张表里面去

如果需要读取hbase里面的数据，mapper需要继承 TableMapper

如果需要写入数据到hbase里面去，reduce需要继承TableReducer

需求二：读取hdfs数据，写入到hbase里面去

0007 zhangsan 18

0008 lisi 25

0009 wangwu 20

作业：读取hbase里面的数据，写入到hdfs里面去

mapper extends TableMapper

reducer extends Reducer

通过bulk load的方式，快速的加载海量的数据到hbase里面去，并且不会影响客户端的操作

核心的思想：就是将文件转换成为HFile这种格式，然后将HFile的数据加载到hbase里面去

现在有海量的数据，如何快速的加载到hbase里面去 ==》 bulk load 的方式进行加载

需求：将 /hbase/input/user.txt 这个文件，转换成为HFile这种格式，加载到myuser2这个表里面去

读取数据，然后将数据存储格式转换成为HFile，不需要reducer类就可以实现

classpath : 资源的类路径

加载hbase的jar包到hadoop的classpath里面去，便于我们执行mr的时候，能够找得到hbase的jar包

export HBASE\_HOME=/export/servers/hbase-1.2.0-cdh5.14.0/

export HADOOP\_HOME=/export/servers/hadoop-2.6.0-cdh5.14.0/

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

hive：数据仓库工具 sql on hadoop的框架 impala 也是一个sql on hadoop的框架

hive不会存储数据，数据都是在hdfs上面

hive也不会进行数据计算，数据计算都是mapreduce来做的

HBase：数据库工具 nosql的非关系型数据库

hbase可以集成mapreduce

hbase数据存储在hdfs上面

hive的用途：主要用于做数据分析

hbase的用途：实时的数据的增删改查

hive整合hbase：

第一个方向，数据在hbase里面了，我们可以通过hive映射hbase的表，实现hive分析hbase表当中的数据

第二个方向，数据都是在hive里面了，我们可以将hive分析的结果，保存到hbase里面去，供我们报表查询使用

需求一：将hive分析结果的数据，保存到HBase当中去

创建hive映射hbase的表

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");

stored as 'textfile'

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

指定字段之间的分割

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

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

需求二：数据已经保存在了hbase里面了，需要通过hive直接分析hbase里面的数据

在hive当中创建一个表，映射hbase的表即可

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");

CREATE external TABLE course.hbase2hive(id int, name string, score int)

WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key 将rowkey映射到id字段,cf:name,cf:score")

作业：

读取hbase里面的数据，保存到hdfs上面去

如何通过hive整合hbase来实现功能

sqoop整合hbase

导入导出工具

需求：将mysql的数据导入到hbase里面来

使用sqoop命令将mysql数据导入到hbase里面去

bin/sqoop import \

--connect jdbc:mysql://192.168.83.53:3306/library \

--username root \

--password 123456 \

--table book \

--columns "id,name,price" \

--column-family "info" \

--hbase-create-table \

--hbase-row-key "id" \

--hbase-table "hbase\_book" \

--num-mappers 1 \

--split-by id

导出需求：将hbase当中表的数据导出到mysql里面去

将hbase的数据导出到mysql里面来

dataSource ==》hbase

dataSink ==》 mysql

bin/sqoop export --help 没有参数

注意，不能够使用sqoop直接导出hbase里面的数据

hbase表数据 ===》 创建hive外部表，映射hbase表 ==》 创建hive内部表 ==》 insert overwrite 内部表 select from 外部表 ==》 数据存储在hive里面了，也就是直接放到了hdfs上面了 ==》 导出hdfs上面的数据到mysql里面去即可

创建外部表映射hbase数据

CREATE EXTERNAL TABLE course.hbase2mysql (id int,name string,price int)

STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'

WITH SERDEPROPERTIES (

"hbase.columns.mapping" =

":key,info:name, info:price"

)

TBLPROPERTIES( "hbase.table.name" = "hbase\_book",

"hbase.mapred.output.outputtable" = "hbase2mysql");

insert overwrite table course.hbase2mysqlin select \* from course.hbase2mysql;

使用sqoop导出数据

bin/sqoop export --connect jdbc:mysql://192.168.83.53:3306/library --username root --password 123456 --table book --export-dir /user/hive/warehouse/course.db/hbase2mysqlin --input-fields-terminated-by '\001' --input-null-string '\\N' --input-null-non-string '\\N';

将hbase表数据，写入到hdfs上面去

增量导入

全量导入

减量导入：

读取mysql表数据

--connect jdbc:mysql://192.168.83.53:3306/libray --password 123456 --username root --table book

-m 1 --hbase-table myhbase\_table

导入hbase表：

--column-family <family> Sets the target column family for the

import

--hbase-bulkload Enables HBase bulk loading

--hbase-create-table If specified, create missing HBase tables

--hbase-row-key <col> Specifies which input column to use as the

row key

--hbase-table <table> Import to <table> in HBase

hbase当中的预分区：

创建一张表，默认就会有一个region，startKey空值 endKey空值

现在这个表当中只有一个region，所有的数据都会 写入到这一个region里面去

可不可以在创建表的时候，，多创建几个region出来，每台机器上面都有一些region

预分区：在我们创建hbase表的时候，提前定义好region个数，以及每个region管理的startkey和endkey的范围值，就叫做预分区

预分区方式：

第一种方式，通过手动指定

create 'staff','info','partition1',SPLITS => ['1000','2000','3000','4000']

第二种方式：通过16进制算法，进行预分区

create 'staff2','info','partition2',{NUMREGIONS => 15, SPLITALGO => 'HexStringSplit'}

所有的rowkey都会按照字典顺序进行排序，这个rowkey符合哪一个region管理的范围值，就放到哪一个reigon里面去

第三种方式：将预分区的规则定义到文件里面去

create 'staff3','partition2',SPLITS\_FILE => '/export/servers/splits.txt'

第四种方式：通过JavaAPI实现预分区创建

通过JavaAPI实现预分区

hbase当中rowkey的设计：

1、rowkey的长度原则：

rowkey最大的长度64KB，实际工作当中，一般10-100字节，不宜过长，在可能的前提下，越短越好

2、rowkey的散列原则：

尽量设计rowkey实现将数据均匀的分布到不同的region里面去，不要出现所有的数据都去往了某一个region

rowkey都是按照字典顺序进行排列的，hbase数据究竟存储到哪一个region里面去的时候，会比较rowkey高几位数据

随机字符串\_15897845612

随机字符串\_15894561235

实现将数据均匀保存到不同的region里面去

3、rowkey的唯一原则：rowkey不能重复，如果重复，就会出现数据覆盖情况

尽量将统一类型的数据，存放到同一个region里面去，便于我们查找数据

如何解决rowkey的热点问题：也叫数据倾斜问题

1、加盐 ==》 给rowkey加上随机的前缀的字符串

2、取hashCode码值，保证同一行数据，永远使用相同的前缀加盐

3、rowkey的反转

15894651235 ==》 反转 53215649851

15874563214 ==》 反转 41236547851

4、时间戳反转

每条数据都会有一个时间戳 时间戳的前缀都一样，可以将时间戳进行反转

话费查询：一个人一个月充一次话费，一年 1亿用户 \* 12个月 \* 1 次 = 12亿条数据

mysql存不下

oracle太贵存不起

只能够查询最近三个月的话费详单

上网详情：流量详情

使用hbase来实现最近一年的数据的查询：

你该如何设计rowkey？？？

手机号：

业务类型：

起始时间：

结束时间：

手机号反转：不同的人的手机号反转，高位都不一样，同一个人手机号反转，

反转之后还是一样（可以将同一个人的数据放到同一个region里面去），查询会更快一点

手机号反转:详单类型:时间戳

new Scan ();

scan.setStartRow("手机号反转:业务类型:起始时间")

scan.setStopRow("手机号反转:业务类型:结束时间")

hbase当中rowkey的设计，一定是为了方便我们快速的查询

new Scan ();

scan.setStartRow("手机号反转:业务类型:起始时间")

scan.setStopRow("手机号反转:业务类型:结束时间")

实际工作当中一般rowkey都是使用复合键：多个条件值组合到一起去，使用冒号或者下划线进行分割

hbase当中的协处理器：

hbase当中没法使用sql语句来查询数据

hbase没法轻易的建立二级索引，也没法记性求和，统计，排序等功能

0.92版本之后引入协处理器的功能：可以用于数据的拦截，以及求和，统计，排序等等

observer：触发器 ==》 产生特定的事件的时候，就会去运行的程序

endpoint： 数值统计的功能 ==》 求和，求最大值，求最小值，求平均值等等

需求：将插入到proc1表里面的数据，进行拦截，先插入到proc2表里面去，然后再插入到proc1表里面去

动态的挂载协处理器

alter 'proc1',METHOD => 'table\_att','Coprocessor'=>'hdfs://node01:8020/processor/processor.jar|cn.itcast.hbase.processor.MyProcessor|1001|'

可以使用协处理器observer来构建二级索引的操作

hbase当中的索引问题：

一级索引：使用rowkey来进行查询就是一级索引 ，如果查询条件自由组合的话，rowkey基本上就没法设计进行查询了

所以引入二级索引的概念：以空间换时间的概念

get rowkey

scan startRowkey endRowKey

scan 全表扫描 没人用

前端页面查询条件：name，age ，address，salary，dept

二级索引：核心概念：以空间换时间的概念，将数据冗余存储，实现数据的快速查询

主要解决多条件组合查询的问题

二级索引：

<property>

<name>dfs.namenode.name.dir</name>

<value>file:///export/servers/hadoop-2.6.0-cdh5.14.0/hadoopDatas/namenodeDatas,file:///export/disk2</value>

</property>

100个GB数据 1.5个本分 ==》 150GB

mapper个数增加：将文件全部打散成为小文件

mapper个数减少：将小文件进行合并

课程总结：

1、hbase与mr的整合 搞定

2、hbase与hive整合 搞定

3、hbase与sqoop整合 知道

4、hbase的预分区 好好理解一下，几种预分区方式 知道

5、hbaserowkey设计技巧 登录你们的营业厅查询一下你们的话费详单

6、hbase协处理器 知道 observer endpoint

7、二级索引介绍 二级索引为了解决什么问题而提出来的

8、hbase与hue整合

9、hbase调优 了解

HBase第三天

课程内容回顾：

1、hbase与mapreduce集成

通过mr读取hbase里面的数据 mapper extends TableMapper

通过mr写入数据到hbase里面去 reducer extends TableReducer

如果需要将数据写入到hbase里面去 构建 key，value对的类型 ImmutableBytesWritable Put

第一个案例：读取hbase数据，写入到另外一张表里面去

第二个案例：读取hdfs的数据，写入到hbase里面去

第三个：读取hbase数据，写入到hdfs里面去

第四个：bulk load 批量数据加载 ==》 将数据转换成为 HFile这种格式，然后直接加载到hbase里面去

2、hbaae与hive整合

hive：数据仓库工具，主要是面向数据分析

hbase：数据库的工具，主要是面向数据的实时的增删改查

hive整合hbase实现通过hive来分析hbase当中的数据，或者将hive分析的结果，保存到hbase里面去

第一个方向：数据存储在hive表里面 可以将分析的结果，保存到hbase里面去

第二个方向：数据存储在hbase里面了，可以通过hive来分析hbase里面的数据

3、sqoop与hbase的整合

sqoop就是一个导入导出的工具

通过sqoop可以将mysql数据导入到hbase里面去

不能直接通过sqoop导出hbase的数据 hbase数据 ==》 hive映射表 ==》 hive内部表 ==》 数据在hdfs上面了 ==》导出hdfs的数据

4、hbase的预分区

预分区：主要是为了解决数据倾斜的问题。

在创建表的时候，可以提前指定预分区的规则，创建多个分区出来，实现数据的自动的负载均衡

四种方式：

1、手动指定方式

2、通过16进制算法，生成预分区的规则

3、通过文件指定 一行一个分区规则

4、通过JavaAPI，使用二维数组来实现的

5、rowkey的设计技巧

1、长度原则 最大rowkey 64KB ,一般都是10-100个字节 ，越短越好

2、散列原则 rowkey需要均匀的负载到各个不同的region里面去，避免数据的热点问题

3、唯一原则 rowkey不能重复

rowkey的热点问题：

1、加盐 ==》 给rowkey的高位加随机数

2、hash ==》 给rowkey取hashCode码值

3、反转 ==》 手机号反转

4、时间戳反转 ==》 给rowkey添加时间戳反转

尽量避免rowkey的高几位出现大量的相同的数据

6、hbase的协处理器

observer：类似于一个触发器，出现某些特定的事件的时候，就会发生。还有点类似于监听器

endpoint：主要用于求最大值，最小值，平均值 数值统计的功能

7、二级索引介绍

二级索引为了解决什么问题：为了解决hbase的多条件查询的问题

二级索引核心思想：以空间换时间 将需要查询的字段，存储一份到其他的地方去，进行冗余存储，实现以空间换取时间查询的效率

8、hbase整合hue

9、hbase的调优 了解

今日课程内容：

使用hbase实现存储微博内容

发送微博：

关注某些人：

被某些人关注：

取消关注：

拉取关注人的微博内容：

1、hbase当中的命名空间namespace

namespace：类似于mysql当中创建的一个库，默认有两个namespace default 和 hbase

mysql当中，可以对数据库进行授权的管理：可以给mysql当中某个指定的用户授予某个指定的数据库的权限

namespace类似于mysql当中的数据库的概念，也可以做到安全认证授权等等一些操作

hbase当中数据的TTL的概念：Time To Live 就是设置hbase当中数据的过期时间

在hbase当中可以针对某一个列族设置数据的过期时间，也可以针对某一条具体的数据，进行设置过期时间

rowkey name

1 zhangsan

1 lisi

1 wangwu

1 zhaoliu

1 tianqi

1 laoli

可以设置版本的确界

设置数据版本 下界为3 上界为 5

一个cell当中，最少保留3个版本，最多保留五个版本

早期的hbase版本当中，每个单元格，默认保留三个版本，上界和下界都是3

新版本当中，hbase的每个单元格，默认保留1个版本，上界和下界都是1

注意：如果针对列族设置了过期的时间，以及针对列族设置了上界以及下界，

那么就会优先满足下界的条件，然后再进行过期

第一张表：用户微博内容发送表，主要用于存储微博发送的内容weibo:content

用户id为1的这个人，关注了用户id为2,3，M的这三个人

用户id为1的这个人，被用户id为4,5，N这三个人关注了

2,3，M有一个粉丝是1

1 关注了2,3，M这三个人

1有三个粉丝是4,5,N

4,5,N这三个人关注了1 这个用户

rowkey：使用 uid\_时间戳

列族：info

列名：content，pic，title

第二张表：weibo:relation 表 主要存储用户之间的关系

谁关注了谁，谁粉丝了谁

定义两个列族：

attends：关注了哪些人

fans：有哪些粉丝，被哪些人关注了

1：关注了2,3,M

2:fans 1

3:fans 1

M:fans 1

rowkey:使用uid

列族：attends列族 fans列族

列名：直接使用uid列值作为列名就行了

第一次 1 用户关注了一批人 2,3,M 列名设计 直接使用uid

列名直接设置成为列值

第三张表：微博收件箱表weibo:receive\_content\_email

主要用于保存每个用户拉取收件信息

rowkey：设计成为uid

列族：info列族

列名：关注人的uid

列值：使用多版本特性，存储某个用户历史的所有的版本

第一张表：weibo:content表

主要用于存储用户发送的微博内容

rowkey：uid\_时间戳

列族：info 列族 用于存储发送的微博内容

列名：随便起

第二张表：存储用户关系表weibo:relation

主要用于存储用户的关系表

用户关注了哪些人，用户有哪些粉丝

rowkey：uid

列族：attends列族 关注了哪些人 fans列族 有哪些粉丝

列名：使用uid来作为列名

第三张表：weibo:receive\_content\_email

主要用于存储用户关注的人的微博的rowkey

rowkey：uid

列族：info

列名：关注用户的uid

课程总结：

1、命名空间 hbase当中命名空间概念 namespace

类似于myslq当中数据库的概念

2、hbase当中版本确界以及TTL过期时间

过期时间：可以针对列族设置，也可以针对某一条具体的数据设置

版本确界：定义版本上界以及下界

使用hbase实现微博数据存储：

创建了三张表

content表：存放微博发送内容

relation表：存储用户之间的关系

receive\_content\_email：用户的收件箱表，拉取所有关注的用户发送的微博的rowkey

发送微博内容：

A用户发送一个微博，首先将微博内容存储到weibo:content表里面去

获取A用户有哪些fans

第三步：给这些粉丝的收件箱表里面添加一条数据

添加关注：添加关注用户，A用户可以添加一批关注用户

A用户在attends列族里面添加关注用户

被关注的用户添加一个fans是A

收件箱表里面，A用户还需要收取关注人的微博内容

取消关注：与添加关注反着来

拉取微博内容：直接在收件箱表里面获取微博内容rowkey即可