# HBASE数据库

# 1. Hbase基础

## 1.1 hbase数据库介绍

**1、简介**

hbase是基于Google BigTable模型开发的，典型的key/value系统。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。它是Apache Hadoop生态系统中的重要一员，主要用于海量结构化和半结构化数据存储。

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)。

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）

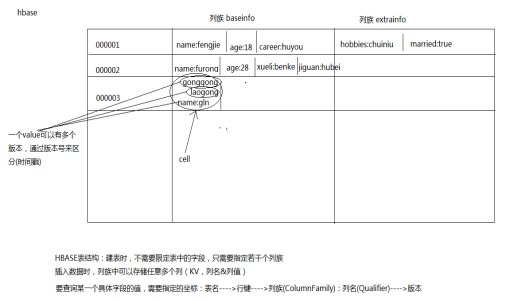
与hadoop一样，Hbase目标主要依靠横向扩展，通过不断增加廉价的商用服务器，来增加计算和存储能力。

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
* 无模式：每行都有一个可排序的主键和任意多的列，列可以根据需要动态的增加，同一张表中不同的行可以有截然不同的列；
* 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。
* 数据多版本：每个单元中的数据可以有多个版本，默认情况下版本号自动分配，是单元格插入时的时间戳
* 数据类型单一：Hbase中的数据都是字节数组 byte[]。

**2、表结构逻辑视图**

HBase以表的形式存储数据。表有行和列组成。列划分为若干个列族(column family)



**3、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。

**4、列族**

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

**5、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）。

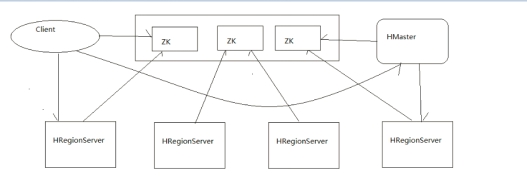
用户可以针对每个列族进行设置。

**6、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。

## 1.2 hbase集群结构



Hbase基本组件说明：

|  |
| --- |
| **Client：**  包含访问Hbase的接口，并维护cache来加快对Hbase的访问，比如region的位置信息。  **HMaster：**  是hbase集群的主节点，可以配置多个，用来实现HA  为RegionServer分配region  负责RegionServer的负载均衡  发现失效的RegionServer并重新分配其上的region  **RegionServer：**  Regionserver维护region，处理对这些region的IO请求  Regionserver负责切分在运行过程中变得过大的region  **Region:**  分布式存储的最小单元。  **Zookeeper作用:**  通过选举，保证任何时候，集群中只有一个HMaster，HMaster与RegionServers 启动时会向ZooKeeper注册  存贮所有Region的寻址入口  实时监控Region server的上线和下线信息。并实时通知给HMaster  存储HBase的schema和table元数据  Zookeeper的引入使得HMaster不再是单点故障 |

## 1.3 hbase集群搭建

----先部署一个zookeeper集群

（1）上传hbase安装包

（2）解压

（3）配置hbase集群，要修改3个文件

注意：要把hadoop的hdfs-site.xml和core-site.xml 放到hbase/conf下

（3.1）修改hbase-env.sh

|  |
| --- |
| export JAVA\_HOME=/var/local/jdk1.7.0\_67  //告诉hbase使用外部的zk  export HBASE\_MANAGES\_ZK=false |

（3.2）修改 hbase-site.xml

|  |
| --- |
| <configuration>  <!-- 指定hbase在HDFS上存储的路径 -->  <property>  <name>hbase.rootdir</name>  <value>hdfs://itcast01:9000/hbase</value>  </property>  <!-- 指定hbase是分布式的 -->  <property>  <name>hbase.cluster.distributed</name>  <value>true</value>  </property>  <!-- 指定zk的地址，多个用“,”分割 -->  <property>  <name>hbase.zookeeper.quorum</name>  <value>itcast01:2181,itcast02:2181,itcast03:2181</value>  </property>  </configuration> |

（3.3）修改 regionservers

|  |
| --- |
| itcast01  itcast02  itcast03 |

(3.3) 修改 backup-masters来指定备用的主节点

|  |
| --- |
| [root@itcast01 conf]# vi backup-masters  itcast02 |

(3.4) 拷贝hbase到其他节点

scp -r /var/local/hbase itcast02:/var/local

scp -r /var/local/hbase itcast03:/var/local

(4) 将配置好的HBase拷贝到每一个节点并同步时间。

(5) 启动所有的hbase进程

首先启动zk集群

./zkServer.sh start

启动hdfs集群

start-dfs.sh

启动hbase，在主节点itcast01上运行：

start-hbase.sh

(6) 通过浏览器访问hbase管理页面

itcast01:16010

itcast02:16010

(7) 为保证集群的可靠性，要启动多个HMaster

hbase-daemon.sh start master

## 1.4 命令行演示

### 1.4.1 基本shell命令

|  |
| --- |
| 进入hbase命令行  ./hbase shell  显示hbase中的表  list  创建user表，包含info、data两个列族  create 'user', 'info1', 'data1'  create 'user', {NAME => 'info', VERSIONS => '3'}  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  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  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  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  put 'user', 'rk0001', 'data:pic', 'picture'  获取user表中row key为rk0001的所有信息  get 'user', 'rk0001'  获取user表中row key为rk0001，info列族的所有信息  get 'user', 'rk0001', 'info'  获取user表中row key为rk0001，info列族的name、age列标示符的信息  get 'user', 'rk0001', 'info:name', 'info:age'  获取user表中row key为rk0001，info、data列族的信息  get 'user', 'rk0001', 'info', 'data'  get 'user', 'rk0001', {COLUMN => ['info', 'data']}  get 'user', 'rk0001', {COLUMN => ['info:name', 'data:pic']}  获取user表中row key为rk0001，列族为info，版本号最新5个的信息  get 'user', 'rk0001', {COLUMN => 'info', VERSIONS => 2}  get 'user', 'rk0001', {COLUMN => 'info:name', VERSIONS => 5}  get 'user', 'rk0001', {COLUMN => 'info:name', VERSIONS => 5, TIMERANGE => [1392368783980, 1392380169184]}  获取user表中row key为rk0001，cell的值为zhangsan的信息  get 'people', 'rk0001', {FILTER => "ValueFilter(=, 'binary:图片')"}  获取user表中row key为rk0001，列标示符中含有a的信息  get 'people', 'rk0001', {FILTER => "(QualifierFilter(=,'substring:a'))"}  put 'user', 'rk0002', 'info:name', 'fanbingbing'  put 'user', 'rk0002', 'info:gender', 'female'  put 'user', 'rk0002', 'info:nationality', '中国'  get 'user', 'rk0002', {FILTER => "ValueFilter(=, 'binary:中国')"}  查询user表中的所有信息  scan 'user'  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 'persion', {COLUMNS => 'info', RAW => true, VERSIONS => 3}  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']}  查询user表中列族为info、列标示符为name的信息  scan 'user', {COLUMNS => 'info:name'}  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}  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'))"}  查询user表中列族为info，rk范围是[rk0001, rk0003)的数据  scan 'people', {COLUMNS => 'info', STARTROW => 'rk0001', ENDROW => 'rk0003'}  查询user表中row key以rk字符开头的  scan 'user',{FILTER=>"PrefixFilter('rk')"}  查询user表中指定范围的数据  scan 'user', {TIMERANGE => [1392368783980, 1392380169184]}  删除数据  删除user表row key为rk0001，列标示符为info:name的数据  delete 'people', 'rk0001', 'info:name'  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  清空user表中的数据  truncate 'people'  修改表结构  首先停用user表（新版本不用）  disable 'user'  添加两个列族f1和f2  alter 'people', NAME => 'f1'  alter 'user', NAME => 'f2'  启用表  enable 'user'  ###disable 'user'(新版本不用)  删除一个列族：  alter 'user', NAME => 'f1', METHOD => 'delete' 或 alter 'user', 'delete' => 'f1'  添加列族f1同时删除列族f2  alter 'user', {NAME => 'f1'}, {NAME => 'f2', METHOD => 'delete'}  将user表的f1列族版本号改为5  alter 'people', NAME => 'info', VERSIONS => 5  启用表  enable 'user'  删除表  disable 'user'  drop 'user'  get 'person', 'rk0001', {FILTER => "ValueFilter(=, 'binary:中国')"}  get 'person', 'rk0001', {FILTER => "(QualifierFilter(=,'substring:a'))"}  scan 'person', {COLUMNS => 'info:name'}  scan 'person', {COLUMNS => ['info', 'data'], FILTER => "(QualifierFilter(=,'substring:a'))"}  scan 'person', {COLUMNS => 'info', STARTROW => 'rk0001', ENDROW => 'rk0003'}  scan 'person', {COLUMNS => 'info', STARTROW => '20140201', ENDROW => '20140301'}  scan 'person', {COLUMNS => 'info:name', TIMERANGE => [1395978233636, 1395987769587]}  delete 'person', 'rk0001', 'info:name'  alter 'person', NAME => 'ffff'  alter 'person', NAME => 'info', VERSIONS => 10  get 'user', 'rk0002', {COLUMN => ['info:name', 'data:pic']} |

## 1.5 hbase代码开发（基本，过滤器查询）

### 1.5.1 基本增删改查java实现

|  |
| --- |
| public class HbaseDemo {  private Configuration conf = null;    @Before  public void init(){  conf = HBaseConfiguration.create();  conf.set("hbase.zookeeper.quorum", "itcast01:2181,itcast02:2181,itcast03:2181");  }    @Test  public void testDrop() throws Exception{  HBaseAdmin admin = new HBaseAdmin(conf);  admin.disableTable("account");  admin.deleteTable("account");  admin.close();  }    @Test  public void testPut() throws Exception{  HTable table = new HTable(conf, "person\_info");  Put p = new Put(Bytes.toBytes("person\_rk\_bj\_zhang\_000002"));  p.add("base\_info".getBytes(), "name".getBytes(), "zhangwuji".getBytes());  table.put(p);  table.close();  }    @Test  public void testDel() throws Exception{  HTable table = new HTable(conf, "user");  Delete del = new Delete(Bytes.toBytes("rk0001"));  del.deleteColumn(Bytes.toBytes("data"), Bytes.toBytes("pic"));  table.delete(del);  table.close();  }  @Test  public void testGet() throws Exception{  HTable table = new HTable(conf, "person\_info");  Get get = new Get(Bytes.toBytes("person\_rk\_bj\_zhang\_000001"));  get.setMaxVersions(5);  Result result = table.get(get);    List<Cell> cells = result.listCells();    for(Cell c:cells){  }    //result.getValue(family, qualifier); 可以从result中直接取出一个特定的value    //遍历出result中所有的键值对  List<KeyValue> kvs = result.list();  //kv ---> f1:title:superise.... f1:author:zhangsan f1:content:asdfasldgkjsldg  for(KeyValue kv : kvs){  String family = new String(kv.getFamily());  System.out.println(family);  String qualifier = new String(kv.getQualifier());  System.out.println(qualifier);  System.out.println(new String(kv.getValue()));    }  table.close();  } |

### 1.5.2 过滤器查询

引言：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。 |

Hbase的过滤器分类

* 比较过滤器

1.1 行键过滤器RowFilter

|  |
| --- |
| Filter filter1 = new RowFilter(CompareOp.LESS\_OR\_EQUAL, new BinaryComparator(Bytes.toBytes("row-22")));  scan.setFilter(filter1); |

1.2 列族过滤器FamilyFilter

|  |
| --- |
| Filter filter1 = new FamilyFilter(CompareFilter.CompareOp.LESS, new BinaryComparator(Bytes.toBytes("colfam3")));  scan.setFilter(filter1); |

1.3 列过滤器QualifierFilter

|  |
| --- |
| filter = new QualifierFilter(CompareFilter.CompareOp.LESS\_OR\_EQUAL, new BinaryComparator(Bytes.toBytes("col-2")));  scan.setFilter(filter1); |

1.4 值过滤器 ValueFilter

|  |
| --- |
| Filter filter = new ValueFilter(CompareFilter.CompareOp.EQUAL, new SubstringComparator(".4") );  scan.setFilter(filter1); |

* 专用过滤器

2.1 单列值过滤器 SingleColumnValueFilter ----会返回满足条件的整行

|  |
| --- |
| SingleColumnValueFilter filter = new SingleColumnValueFilter(  Bytes.toBytes("colfam1"),  Bytes.toBytes("col-5"),  CompareFilter.CompareOp.NOT\_EQUAL,  new SubstringComparator("val-5"));  filter.setFilterIfMissing(true); //如果不设置为true，则那些不包含指定column的行也会返回  scan.setFilter(filter1); |

2.2 SingleColumnValueExcludeFilter

与上相反

2.3 前缀过滤器 PrefixFilter----针对行键

|  |
| --- |
| Filter filter = new PrefixFilter(Bytes.toBytes("row1"));  scan.setFilter(filter1); |

2.4 列前缀过滤器 ColumnPrefixFilter

|  |
| --- |
| Filter filter = new ColumnPrefixFilter(Bytes.toBytes("qual2"));  scan.setFilter(filter1); |

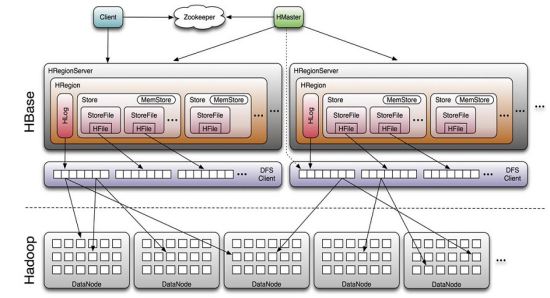
2.4分页过滤器 PageFilter

|  |
| --- |
| public static void main(String[] args) throws Exception {  Configuration conf = HBaseConfiguration.create();  conf.set("hbase.zookeeper.quorum", "spark01:2181,spark02:2181,spark03:2181");    String tableName = "testfilter";  String cfName = "f1";  final byte[] POSTFIX = new byte[] { 0x00 };  HTable table = new HTable(conf, tableName);  Filter filter = new PageFilter(3);  byte[] lastRow = null;  int totalRows = 0;  while (true) {  Scan scan = new Scan();  scan.setFilter(filter);  if(lastRow != null){  //注意这里添加了POSTFIX操作，用来重置扫描边界  byte[] startRow = Bytes.add(lastRow,POSTFIX);  scan.setStartRow(startRow);  }  ResultScanner scanner = table.getScanner(scan);  int localRows = 0;  Result result;  while((result = scanner.next()) != null){  System.out.println(localRows++ + ":" + result);  totalRows ++;  lastRow = result.getRow();  }  scanner.close();  if(localRows == 0) break;  }  System.out.println("total rows:" + totalRows);  } |

|  |
| --- |
| /\*\*  \* 多种过滤条件的使用方法  \* @throws Exception  \*/  @Test  public void testScan() throws Exception{  HTable table = new HTable(conf, "person\_info".getBytes());  Scan scan = new Scan(Bytes.toBytes("person\_rk\_bj\_zhang\_000001"), Bytes.toBytes("person\_rk\_bj\_zhang\_000002"));    //前缀过滤器----针对行键  Filter filter = new PrefixFilter(Bytes.toBytes("rk"));    //行过滤器 ---针对行键  ByteArrayComparable rowComparator = new BinaryComparator(Bytes.toBytes("person\_rk\_bj\_zhang\_000001"));  RowFilter rf = new RowFilter(CompareOp.LESS\_OR\_EQUAL, rowComparator);    /\*\*  \* 假设rowkey格式为：创建日期\_发布日期\_ID\_TITLE  \* 目标：查找 发布日期 为 2014-12-21 的数据  \* sc.textFile("path").flatMap(line=>line.split("\t")).map(x=>(x,1)).reduceByKey(\_+\_).map((\_(2),\_(1))).sortByKey().map((\_(2),\_(1))).saveAsTextFile("")  \*  \*  \*/  rf = new RowFilter(CompareOp.EQUAL , new SubstringComparator("\_2014-12-21\_"));      //单值过滤器1完整匹配字节数组  new SingleColumnValueFilter("base\_info".getBytes(), "name".getBytes(), CompareOp.EQUAL, "zhangsan".getBytes());  //单值过滤器2 匹配正则表达式  ByteArrayComparable comparator = new RegexStringComparator("zhang.");  new SingleColumnValueFilter("info".getBytes(), "NAME".getBytes(), CompareOp.EQUAL, comparator);  //单值过滤器3匹配是否包含子串,大小写不敏感  comparator = new SubstringComparator("wu");  new SingleColumnValueFilter("info".getBytes(), "NAME".getBytes(), CompareOp.EQUAL, comparator);  //键值对元数据过滤-----family过滤----字节数组完整匹配  FamilyFilter ff = new FamilyFilter(  CompareOp.EQUAL ,  new BinaryComparator(Bytes.toBytes("base\_info")) //表中不存在inf列族，过滤结果为空  );  //键值对元数据过滤-----family过滤----字节数组前缀匹配  ff = new FamilyFilter(  CompareOp.EQUAL ,  new BinaryPrefixComparator(Bytes.toBytes("inf")) //表中存在以inf打头的列族info，过滤结果为该列族所有行  );    //键值对元数据过滤-----qualifier过滤----字节数组完整匹配    filter = new QualifierFilter(  CompareOp.EQUAL ,  new BinaryComparator(Bytes.toBytes("na")) //表中不存在na列，过滤结果为空  );  filter = new QualifierFilter(  CompareOp.EQUAL ,  new BinaryPrefixComparator(Bytes.toBytes("na")) //表中存在以na打头的列name，过滤结果为所有行的该列数据  );    //基于列名(即Qualifier)前缀过滤数据的ColumnPrefixFilter  filter = new ColumnPrefixFilter("na".getBytes());    //基于列名(即Qualifier)多个前缀过滤数据的MultipleColumnPrefixFilter  byte[][] prefixes = new byte[][] {Bytes.toBytes("na"), Bytes.toBytes("me")};  filter = new MultipleColumnPrefixFilter(prefixes);    //为查询设置过滤条件  scan.setFilter(filter);        scan.addFamily(Bytes.toBytes("base\_info"));  //一行  // Result result = table.get(get);  //多行的数据  ResultScanner scanner = table.getScanner(scan);  for(Result r : scanner){  /\*\*  for(KeyValue kv : r.list()){  String family = new String(kv.getFamily());  System.out.println(family);  String qualifier = new String(kv.getQualifier());  System.out.println(qualifier);  System.out.println(new String(kv.getValue()));  }  \*/  //直接从result中取到某个特定的value  byte[] value = r.getValue(Bytes.toBytes("base\_info"), Bytes.toBytes("name"));  System.out.println(new String(value));  }  table.close();  } |

## 1.6 hbase内部原理

### 1.6.1 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region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----root表在哪台服务器上。

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

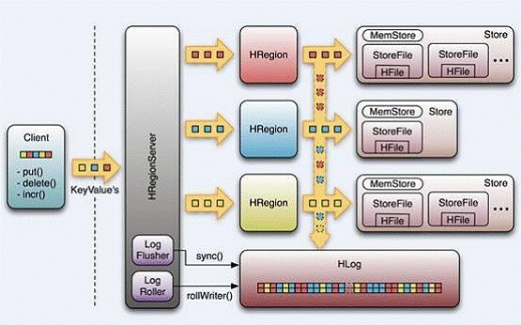
1 Region server维护Master分配给它的region，处理对这些region的IO请求

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。

### 1.6.2 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。

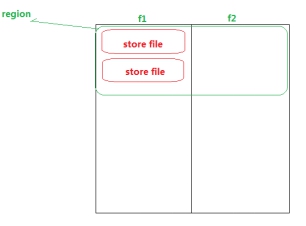
3 region按大小分割的(默认10G)，每个表一开始只有一个region，随着数据不断插入表，region不断增大，当增大到一个阀值的时候，Hregion就会等分会两个新的Hregion。当table中的行不断增多，就会有越来越多的Hregion。

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regionserver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。

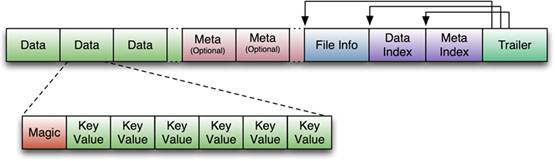
每个Strore又由一个memStore和0至多个StoreFile组成。如上图



#### 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)

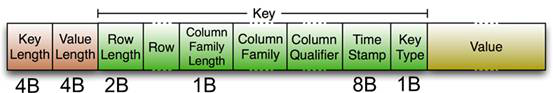
首先HFile文件是不定长的，长度固定的只有其中的两块：Trailer和FileInfo。正如图中所示的，Trailer中有指针指向其他数 据块的起始点。

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)，该机制用于数据的容错和恢复，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。

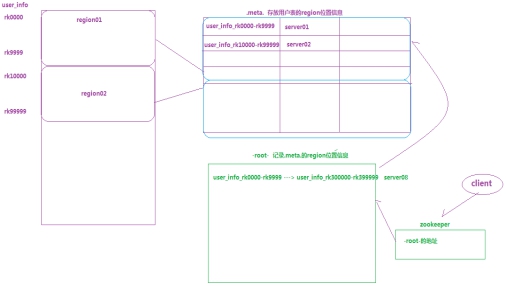
每个HRegionServer中都有一个HLog对象，HLog是一个实现Write Ahead Log的类，在每次用户操作写入MemStore的同时，也会写一份数据到HLog文件中（HLog文件格式见后续），HLog文件定期会滚动出新的，并删除旧的文件（已持久化到StoreFile中的数据）。当HRegionServer意外终止后，HMaster会通过Zookeeper感知到，HMaster首先会处理遗留的 HLog文件，将其中不同Region的Log数据进行拆分，分别放到相应region的目录下，然后再将失效的region重新分配，领取到这些region的HRegionServer在Load Region的过程中，会发现有历史HLog需要处理，因此会Replay HLog中的数据到MemStore中，然后flush到StoreFiles，完成数据恢复。

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。

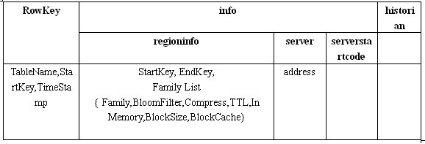
### 1.6.3 寻址机制

#### 1、寻址示意图

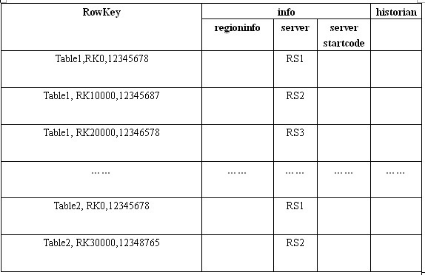


#### -ROOT-和.META.表结构

-ROOT-表结构



**.META.行记录结构**



#### 3、寻址流程

现在假设我们要从Table2里面查询一条RowKey是RK10000的数据。那么我们应该遵循以下步骤：

1. 从.META.表里面查询哪个Region包含这条数据。

2. 获取管理这个Region的RegionServer地址。

3. 连接这个RegionServer, 查到这条数据。

系统如何找到某个row key (或者某个 row key range)所在的region

bigtable 使用三层类似B+树的结构来保存region位置。

第一层： 保存zookeeper里面的文件，它持有root region的位置。

第二层：root region是.META.表的第一个region其中保存了.META.表其它region的位置。通过root region，我们就可以访问.META.表的数据。

第三层： .META.表它是一个特殊的表，保存了hbase中所有数据表的region 位置信息。

说明：

(1) root region永远不会被split，保证了最需要三次跳转，就能定位到任意region 。

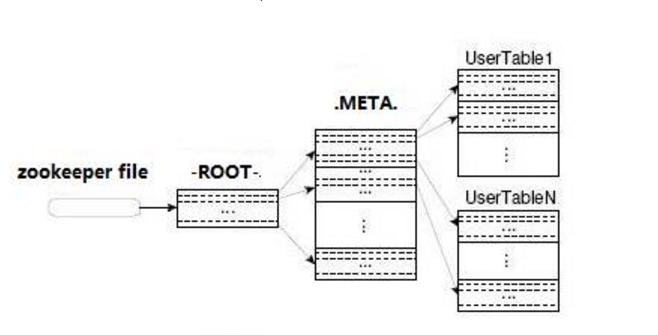
(2).META.表每行保存一个region的位置信息，row key 采用表名+表的最后一行编码而成。

(3) 为了加快访问，.META.表的全部region都保存在内存中。

(4) client会将查询过的位置信息保存缓存起来，缓存不会主动失效，因此如果client上的缓存全部失效，则需要进行最多6次网络来回，才能定位到正确的region(其中三次用来发现缓存失效，另外三次用来获取位置信息)。

#### 4、总结

Region定位流程：



1. **寻找RegionServer**

ZooKeeper--> -ROOT-(单Region)--> .META.--> 用户表

1. **-ROOT-表**

表包含.META.表所在的region列表，该表只会有一个Region；

Zookeeper中记录了-ROOT-表的location。

1. **.META.表**

表包含所有的用户空间region列表，以及RegionServer的服务器地址

### 读写过程

#### 1、读请求过程：

1 客户端通过zookeeper以及root表和meta表找到目标数据所在的regionserver

2 联系regionserver查询目标数据

3 regionserver定位到目标数据所在的region，发出查询请求

4 region先在memstore中查找，命中则返回

5 如果在memstore中找不到，则在storefile中扫描（可能会扫描到很多的storefile----bloomfilter）

补充：布隆过滤器参数类型有2种：

Row、row+col

#### 2、写请求过程：

1 client向region server提交写请求

2 region server找到目标region

3 region检查数据是否与schema一致

4 如果客户端没有指定版本，则获取当前系统时间作为数据版本

5 将更新写入WAL log

6 将更新写入Memstore

7 判断Memstore的是否需要flush为StoreFile文件。

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。

### 1.6.5 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。

### 1.6.6 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。

### 1.6.7 HBase容错性

**Master容错**：Zookeeper重新选择一个新的Master  
 无Master过程中，数据读取仍照常进行；  
 无Master过程中，region切分、负载均衡等无法进行；  
**RegionServer容错**：定时向Zookeeper汇报心跳，如果一旦时间内未出现心跳，Master将该RegionServer上的Region重新分配到其他RegionServer上，失效服务器上“预写”日志由主服务器进行分割并派送给新的RegionServer  
**Zookeeper容错**：Zookeeper是一个可靠地服务，一般配置3或5个Zookeeper实例

# 2. Hbase高级应用

## 2.1建表高级属性

下面几个shell 命令在hbase操作中可以起到很到的作用，且主要体现在建表的过程中，看下面几个create 属性

1、BLOOMFILTER 默认是NONE 是否使用布隆过虑及使用何种方式

布隆过滤可以每列族单独启用。

使用 HColumnDescriptor.setBloomFilterType(NONE | ROW | ROWCOL) 对列族单独启用布隆。

* Default = ROW 对行进行布隆过滤。
* 对 ROW，行键的哈希在每次插入行时将被添加到布隆。
* 对 ROWCOL，行键 + 列族 + 列族修饰的哈希将在每次插入行时添加到布隆

使用方法: create 'table',{BLOOMFILTER =>'ROW'}

启用布隆过滤可以节省读磁盘过程，可以有助于降低读取延迟

2、VERSIONS 默认是1 这个参数的意思是数据保留1个 版本，如果我们认为我们的数据没有这么大的必要保留这么多，随时都在更新，而老版本的数据对我们毫无价值，那将此参数设为1 能节约2/3的空间

使用方法: create 'table',{VERSIONS=>'2'}

附：MIN\_VERSIONS => '0'是说在compact操作执行之后，至少要保留的版本

3、COMPRESSION 默认值是NONE 即不使用压缩

这个参数意思是该列族是否采用压缩，采用什么压缩算法

使用方法: create 'table',{NAME=>'info',COMPRESSION=>'SNAPPY'}

建议采用SNAPPY压缩算法

HBase中，在Snappy发布之前（Google 2011年对外发布Snappy），采用的LZO算法，目标是达到尽可能快的压缩和解压速度，同时减少对CPU的消耗；

在Snappy发布之后，建议采用Snappy算法（参考《HBase: The Definitive Guide》），具体可以根据实际情况对LZO和Snappy做过更详细的对比测试后再做选择。

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **% remaining** | **Encoding** | **Decoding** |
| GZIP | 13.4% | 21 MB/s | 118 MB/s |
| LZO | 20.5% | 135 MB/s | 410 MB/s |
| Zippy/Snappy | 22.2% | 172 MB/s | 409 MB/s |

如果建表之初没有压缩，后来想要加入压缩算法，可以通过alter修改schema

4、alter

使用方法：

如 修改压缩算法

disable 'table'

alter 'table',{NAME=>'info',COMPRESSION=>'snappy'}

enable 'table'

但是需要执行major\_compact 'table' 命令之后 才会做实际的操作。

5、TTL

默认是 2147483647 即:Integer.MAX\_VALUE 值大概是68年

这个参数是说明该列族数据的存活时间，单位是s

这个参数可以根据具体的需求对数据设定存活时间，超过存过时间的数据将在表中不在显示，待下次major compact的时候再彻底删除数据

注意的是TTL设定之后 MIN\_VERSIONS=>'0' 这样设置之后，TTL时间戳过期后，将全部彻底删除该family下所有的数据，如果MIN\_VERSIONS 不等于0那将保留最新的MIN\_VERSIONS个版本的数据，其它的全部删除，比如MIN\_VERSIONS=>'1' 届时将保留一个最新版本的数据，其它版本的数据将不再保存。

6、describe 'table' 这个命令查看了create table 的各项参数或者是默认值。

7、disable\_all 'toplist.\*' disable\_all 支持正则表达式，并列出当前匹配的表的如下：

toplist\_a\_total\_1001

toplist\_a\_total\_1002

toplist\_a\_total\_1008

toplist\_a\_total\_1009

toplist\_a\_total\_1019

toplist\_a\_total\_1035

...

Disable the above 25 tables (y/n)? 并给出确认提示

8、drop\_all 这个命令和disable\_all的使用方式是一样的

9、hbase 表预分区----手动分区

默认情况下，在创建HBase表的时候会自动创建一个region分区，当导入数据的时候，所有的HBase客户端都向这一个region写数据，直到这个region足够大了才进行切分。一种可以加快批量写入速度的方法是通过预先创建一些空的regions，这样当数据写入HBase时，会按照region分区情况，在集群内做数据的负载均衡。

命令方式:

create 't1', 'f1', {NUMREGIONS => 15, SPLITALGO => 'HexStringSplit'}

也可以使用api的方式:

bin/hbase org.apache.hadoop.hbase.util.RegionSplitter test\_table HexStringSplit -c 10 -f info

参数：

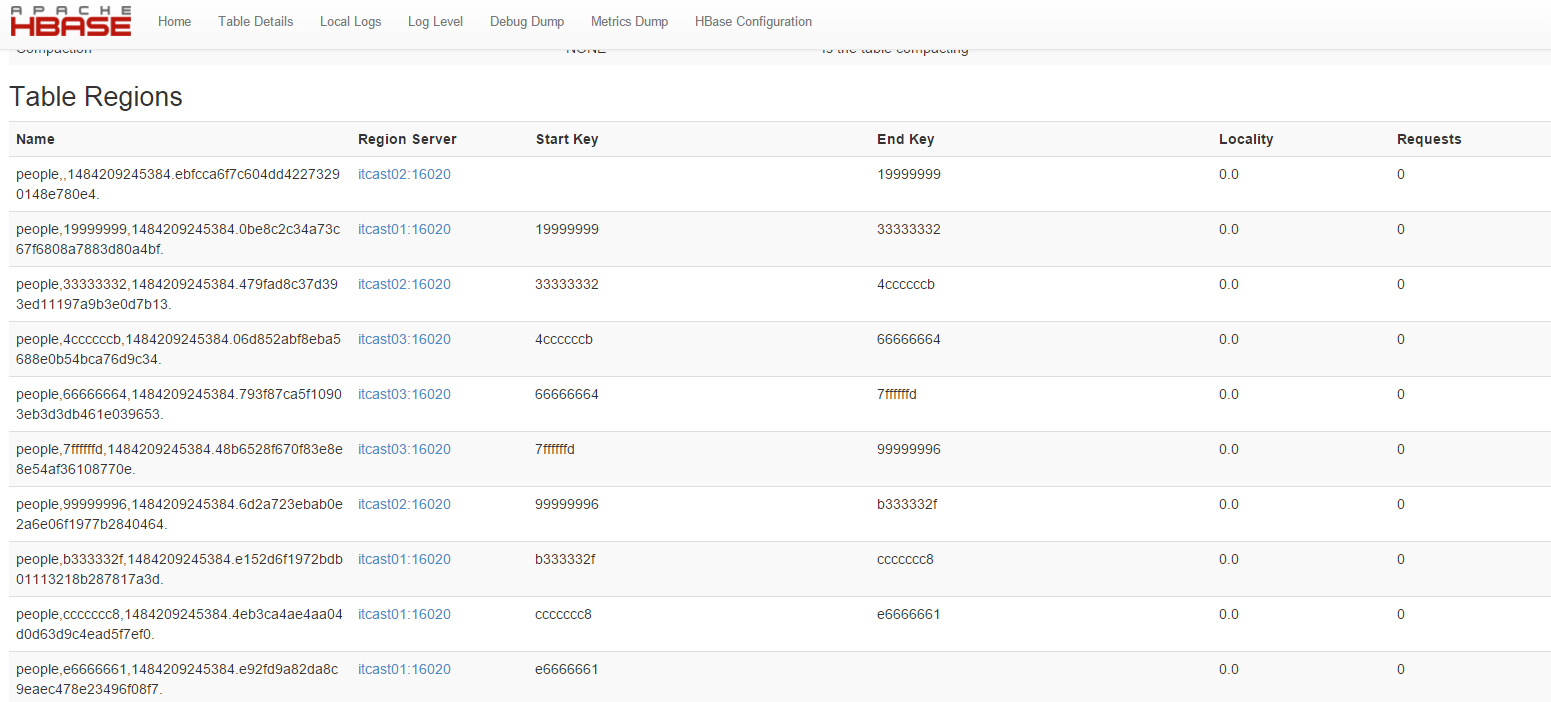
test\_table是表名

HexStringSplit 是split 方式

-c 是分10个region

-f 是family

可在UI上查看结果，如图：



这样就可以将表预先分为15个区，减少数据达到storefile 大小的时候自动分区的时间消耗，并且还有以一个优势，就是合理设计rowkey 能让各个region 的并发请求平均分配(趋于均匀) 使IO 效率达到最高，但是预分区需要将filesize 设置一个较大的值，设置哪个参数呢 hbase.hregion.max.filesize 这个值默认是10G 也就是说单个region 默认大小是10G

这个参数的默认值在0.90 到0.92到0.94.3各版本的变化：256M--1G--10G

但是如果MapReduce Input类型为TableInputFormat 使用hbase作为输入的时候，就要注意了，每个region一个map，如果数据小于10G 那只会启用一个map 造成很大的资源浪费，这时候可以考虑适当调小该参数的值，或者采用预分配region的方式，并将检测如果达到这个值，再手动分配region。

## 2.2 hbase应用案例看行键设计

表结构设计

1、列族数量的设定

以用户信息为例，可以将必须的基本信息存放在一个列族，而一些附加的额外信息可以放在另一列族；

2、行键的设计

语音详单：

13877889988-20150625

13877889988-20150625

13877889988-20150626

13877889988-20150626

13877889989

13877889989

13877889989

----将需要批量查询的数据尽可能连续存放

CMS系统----多条件查询

尽可能将查询条件关键词拼装到rowkey中，查询频率最高的条件尽量往前靠

20150230-zhangsan-category…

20150230-lisi-category…

Category+20150230 20150230-zhangsan-category

(每一个条件的值长度不同，可以通过做定长映射来提高效率)

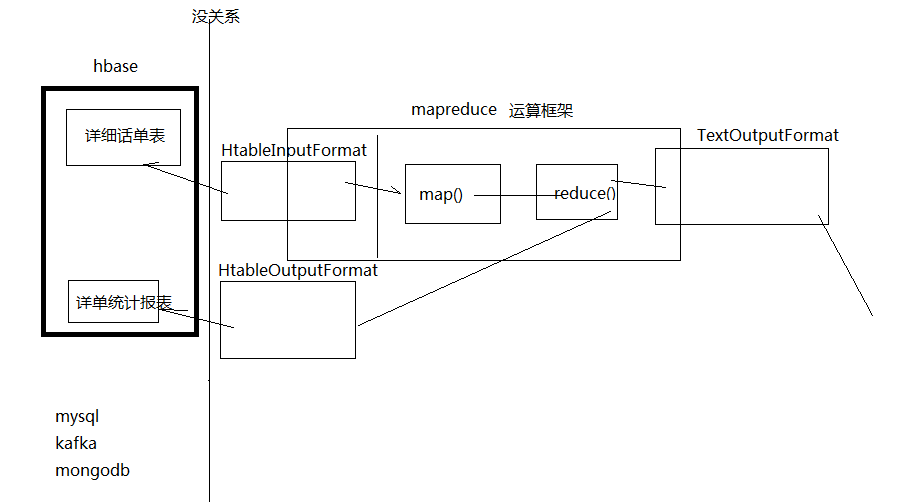
参考：《hbase 实战》----详细讲述了facebook /GIS等系统的表结构设计

## 2.3 Hbase和mapreduce结合

为什么需要用mapreduce去访问hbase的数据？

——加快分析速度和扩展分析能力

Mapreduce访问hbase数据作分析一定是在离线分析的场景下应用



### 2.3.1 从Hbase中读取数据、分析，写入hdfs

|  |
| --- |
| /\*\*  public abstract class TableMapper<KEYOUT, VALUEOUT>  extends Mapper<ImmutableBytesWritable, Result, KEYOUT, VALUEOUT> {  }  \* @author duanhaitao@itcast.cn  \*  \*/  public class HbaseReader {  public static String flow\_fields\_import = "flow\_fields\_import";  static class HdfsSinkMapper extends TableMapper<Text, NullWritable>{  @Override  protected void map(ImmutableBytesWritable key, Result value, Context context) throws IOException, InterruptedException {  byte[] bytes = key.copyBytes();  String phone = new String(bytes);  byte[] urlbytes = value.getValue("f1".getBytes(), "url".getBytes());  String url = new String(urlbytes);  context.write(new Text(phone + "\t" + url), NullWritable.get());    }    }    static class HdfsSinkReducer extends Reducer<Text, NullWritable, Text, NullWritable>{    @Override  protected void reduce(Text key, Iterable<NullWritable> values, Context context) throws IOException, InterruptedException {    context.write(key, NullWritable.get());  }  }    public static void main(String[] args) throws Exception {  Configuration conf = HBaseConfiguration.create();  conf.set("hbase.zookeeper.quorum", "spark01");    Job job = Job.getInstance(conf);    job.setJarByClass(HbaseReader.class);    // job.setMapperClass(HdfsSinkMapper.class);  Scan scan = new Scan();  TableMapReduceUtil.initTableMapperJob(flow\_fields\_import, scan, HdfsSinkMapper.class, Text.class, NullWritable.class, job);  job.setReducerClass(HdfsSinkReducer.class);    FileOutputFormat.setOutputPath(job, new Path("c:/hbasetest/output"));    job.setOutputKeyClass(Text.class);  job.setOutputValueClass(NullWritable.class);    job.waitForCompletion(true);  }    } |

### 2.3.2 从hdfs中读取数据写入Hbase

|  |
| --- |
| package cn.itcast.bigdata.hbase.mr;  import java.io.IOException;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.hbase.HBaseConfiguration;  import org.apache.hadoop.hbase.client.Mutation;  import org.apache.hadoop.hbase.client.Put;  import org.apache.hadoop.hbase.io.ImmutableBytesWritable;  import org.apache.hadoop.hbase.mapreduce.TableMapReduceUtil;  import org.apache.hadoop.hbase.mapreduce.TableOutputFormat;  import org.apache.hadoop.hbase.mapreduce.TableReducer;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.Mapper;  import org.apache.hadoop.mapreduce.Reducer;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;  public class HbaseSinker {  public static String flow\_fields\_import = "flow\_fields\_import";  static class HbaseSinkMrMapper extends Mapper<LongWritable, Text, Text, NullWritable> {  @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  String line = value.toString();  String[] fields = line.split("\t");  String phone = fields[0];  String url = fields[1];  Text text = new Text();  text.set(phone+"\t"+url);  context.write(text, NullWritable.get());  }  }  static class HbaseSinkMrReducer extends TableReducer<Text, NullWritable, ImmutableBytesWritable> {  protected void reduce(Text key, Iterable<NullWritable> values, Context context) throws IOException, InterruptedException {  String[] fields = key.toString().split("\t");  byte[] rowkey=fields[0].getBytes();  Put put = new Put(rowkey);  put.add("info".getBytes(), "url".getBytes(), fields[1].getBytes());  context.write(new ImmutableBytesWritable(rowkey), put);  }  }  public static void main(String[] args) throws Exception {  Configuration conf = HBaseConfiguration.create();  conf.set("hbase.zookeeper.quorum", "itcast01:2181,itcast02:2181,itcast03:2181");  Job job = Job.getInstance(conf);  job.setJarByClass(HbaseSinker.class);  job.setMapperClass(HbaseSinkMrMapper.class);  TableMapReduceUtil.initTableReducerJob(flow\_fields\_import, HbaseSinkMrReducer.class, job);  FileInputFormat.setInputPaths(job, new Path("d://hbasetest/data2"));    job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(NullWritable.class);    job.setOutputKeyClass(ImmutableBytesWritable.class);  job.setOutputValueClass(Mutation.class);  job.waitForCompletion(true);  }  } |

## 2.3 hbase高级编程

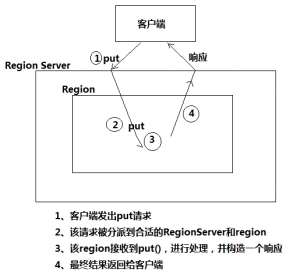
### 2.3.1 协处理器---- Coprocessor

协处理器有两种：observer和endpoint

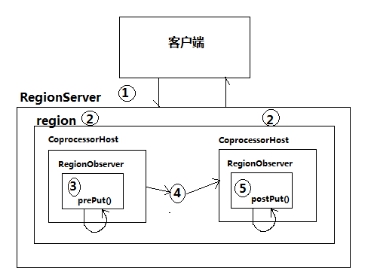
Observer允许集群在正常的客户端操作过程中可以有不同的行为表现

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协处理器
* 正常put请求的流程：



* 加入Observer协处理后的put流程：



1 客户端发出put请求

2 该请求被分派给合适的RegionServer和region

3 coprocessorHost拦截该请求，然后在该表上登记的每个RegionObserver上调用prePut()

4 如果没有被prePut()拦截，该请求继续送到region，然后进行处理

5 region产生的结果再次被CoprocessorHost拦截，调用postPut()

6 假如没有postPut()拦截该响应，最终结果被返回给客户端

* Observer的类型

1. RegionObs——这种Observer钩在数据访问和操作阶段，所有标准的数据操作命令都可以被pre-hooks和post-hooks拦截
2. WALObserver——WAL所支持的Observer；可用的钩子是pre-WAL和post-WAL
3. MasterObserver——钩住DDL事件，如表创建或模式修改

* Observer应用场景示例

见下节；

* Endpoint—参考《Hbase 权威指南》

### 2.3.2 二级索引

row key在HBase中是以B+ tree结构化有序存储的，所以scan起来会比较效率。单表以row key存储索引，column value存储id值或其他数据 ，这就是Hbase索引表的结构。

由于HBase本身没有二级索引（Secondary Index）机制，基于索引检索数据只能单纯地依靠RowKey，为了能支持多条件查询，开发者需要将所有可能作为查询条件的字段一一拼接到RowKey中，这是HBase开发中极为常见的做法

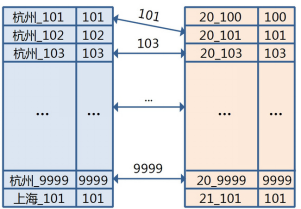
比如，现在有一张1亿的用户信息表，建有出生地和年龄两个索引，我想得到一个条件是在杭州出生，年龄为20岁的按用户id正序排列前10个的用户列表。

有一种方案是，系统先扫描出生地为杭州的索引，得到一个用户id结果集，这个集合的规模假设是10万。然后扫描年龄，规模是5万，最后merge这些用户id，去重，排序得到结果。

这明显有问题，如何改良？

保证出生地和年龄的结果是排过序的，可以减少merge的数据量？但Hbase是按row key排序，value是不能排序的。

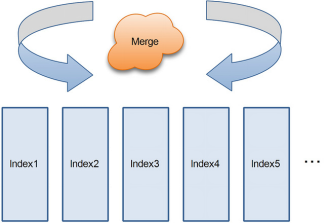
变通一下——将用户id冗余到row key里？OK，这是一种解决方案了，这个方案的图示如下：



merge时提取交集就是所需要的列表，顺序是靠索引增加了\_id，以字典序保证的。

2， 按索引查询种类建立组合索引。

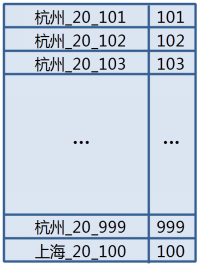
在方案1的场景中，想象一下，如果单索引数量多达10个会怎么样？10个索引，就要merge 10次，性能可想而知。



解决这个问题需要参考RDBMS的组合索引实现。

比如出生地和年龄需要同时查询，此时如果建立一个出生地和年龄的组合索引，查询时效率会高出merge很多。

当然，这个索引也需要冗余用户id，目的是让结果自然有序。结构图示如下：



这个方案的优点是查询速度非常快，根据查询条件，只需要到一张表中检索即可得到结果list。缺点是如果有多个索引，就要建立多个与查询条件一一对应的组合索引

而索引表的维护如果交给应用客户端，则无疑增加了应用端开发的负担

通过协处理器可以将索引表维护的工作从应用端剥离

* 利用Observer自动维护索引表示例

在社交类应用中，经常需要快速检索各用户的关注列表t\_guanzhu，同时，又需要反向检索各户的粉丝列表t\_fensi，为了实现这个需求，最佳实践是建立两张互为反向的表：

* + 一个表为正向索引关注表 “t\_guanzhu”：

Rowkey: A-B

f1:From

f1:To

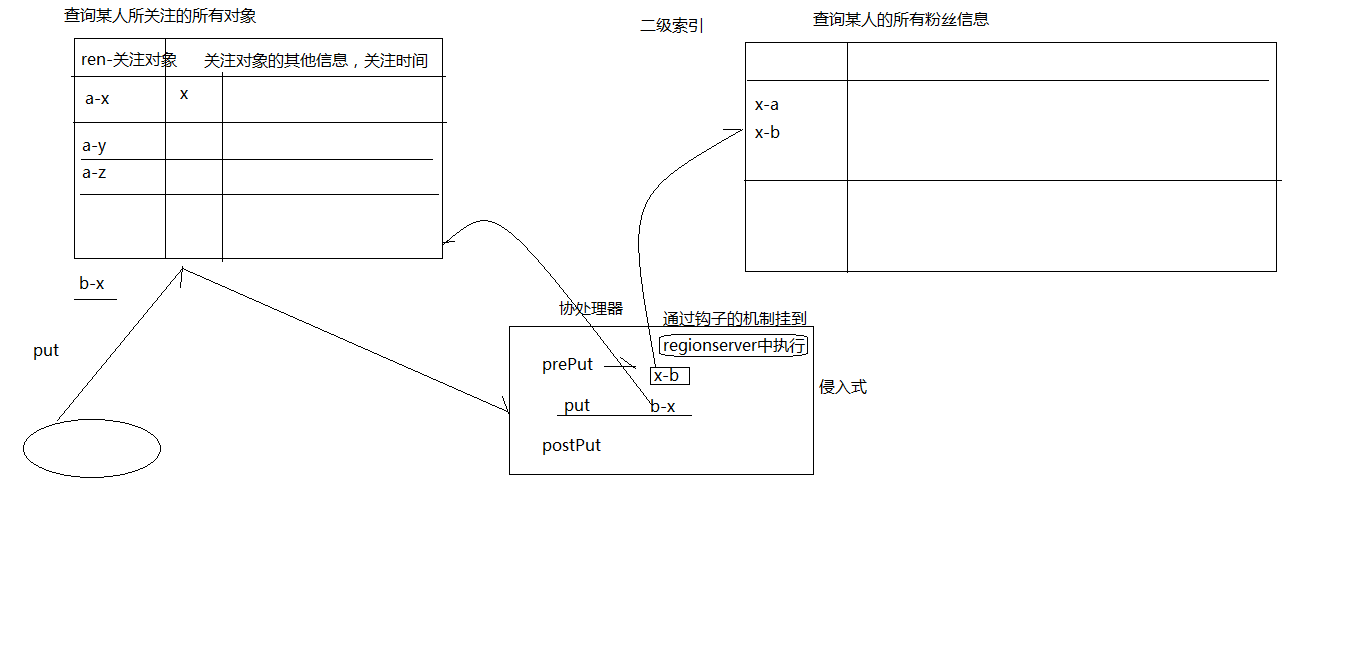
* + 另一个表为反向索引粉丝表：“t\_fensi”：

Rowkey: B—A

f1:From

f1:To

插入一条关注信息时，为了减轻应用端维护反向索引表的负担，可用Observer协处理器实现：



1、编写自定义RegionServer

|  |
| --- |
| package cn.itcast.bigdata.hbase.demo;  import java.io.IOException;  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.Connection;  import org.apache.hadoop.hbase.client.ConnectionFactory;  import org.apache.hadoop.hbase.client.Durability;  import org.apache.hadoop.hbase.client.HTable;  import org.apache.hadoop.hbase.client.Put;  import org.apache.hadoop.hbase.client.Table;  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;  public class InverIndexCoprocessor extends BaseRegionObserver {    @Override  public void prePut(ObserverContext<RegionCoprocessorEnvironment> e, Put put, WALEdit edit, Durability durability) throws IOException {  Configuration conf = HBaseConfiguration.create();  Connection conn = ConnectionFactory.createConnection(conf);    //二级索引表  Table table = conn.getTable(TableName.valueOf("ffindex"));  //拦截到原始put对象中的rowkey和列的Value    byte[] row = put.getRow();  String rowkey = new String(row);  Cell addressCell = put.get("f1".getBytes(), "address".getBytes()).get(0);  byte[] valueArray = addressCell.getValueArray();  String address = new String(valueArray,addressCell.getValueOffset(),addressCell.getValueLength());    String[] user\_fensi = rowkey.split("-");  Put putIndex = new Put((user\_fensi[1]+"-"+user\_fensi[0]).getBytes());    putIndex.addColumn("f1".getBytes(), "address".getBytes(),address.getBytes());  table.put(putIndex);  table.close();  }  }  }  } |

2、打成jar包“coprocess.jar”上传hdfs

hadoop fs -put coprocess.jar /

1. 修改user\_guanzhu的schema，注册协处理器

|  |
| --- |
| hbase(main):017:0> alter ' user\_guanzhu ',METHOD => 'table\_att','coprocessor'=>'hdfs://itcast01:9000/coprocess.jar|cn.itcast.bigdat  a.hbase.demo.InverIndexCoprocessor|1001|'  Updating all regions with the new schema...  0/1 regions updated.  1/1 regions updated.  Done. |

4、检查是否注册成功

|  |
| --- |
| hbase(main):018:0> describe 'user\_guanzhu'  DESCRIPTION ENABLED  'ff', {TABLE\_ATTRIBUTES => {coprocessor$1 => 'hdfs://itcast01:9000/coprocess.jar|cn.itcast.bigdat  a.hbase.demo.InverIndexCoprocessor|1001|'}, {NAME => 'f1', DATA\_BLOCK\_ENCODING => 'NONE', BLOOMF  ILTER => 'ROW', REPLICATION\_SCOPE => '0', VERSIONS => '1', COMPRESSION => 'NONE', MIN\_VERSIONS => '0  ', TTL => '2147483647', KEEP\_DELETED\_CELLS => 'false', BLOCKSIZE => '65536', IN\_MEMORY => 'false', B  LOCKCACHE => 'true'}, {NAME => 'f2', DATA\_BLOCK\_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICATIO  N\_SCOPE => '0', VERSIONS => '1', COMPRESSION => 'NONE', MIN\_VERSIONS => '0', TTL => '2147483647', KE  EP\_DELETED\_CELLS => 'false', BLOCKSIZE => '65536', IN\_MEMORY => 'false', BLOCKCACHE => 'true'}  1 row(s) in 0.0250 seconds |

1. 向正向索引表中插入数据进行验证
2. 先向user\_guanzhu 插入数据
3. 查看user\_guanzhu表的数据
4. 查看索引表ffindex中是否有数据
5. 删除协处理器

|  |
| --- |
| 先disable表  disable 'user\_guanzhu'  删除  alter 'user\_guanzhu', METHOD => 'table\_att\_unset', NAME => 'coprocessor$1'  再enable表  enable 'user\_guanzhu' |