HBase

第一部分 初识 HBase

第 1 节 HBase 简介

1.1 HBase是什么

HBase 基于 Google的BigTable论文而来，是一个分布式海量列式非关系型数据库系统，**可以提供超大规模数据集的实时随机读写**。

接下来，通过一个场景初步认识HBase列存储

如下MySQL存储机制，空值字段浪费存储空间

id	NAME	AGE	SALARY	JOB
1	小明	23		学生
2	小红		10w	律师

如果是列存储的话，可以这么玩……

rowkey：1 name：小明

rowkey：1 age：23

rowkey：1 job：学生

rowkey：2 name ： 小红

rowkey：2 salary：10w

rowkey：2 job：律师

....

列存储的优点：

1）减少存储空间占用。

2）支持好多列

1.2 HBase的特点

- **海量存储：** 底层基于HDFS存储海量数据
- **列式存储：**HBase表的数据是基于列族进行存储的，一个列族包含若干列
- **极易扩展：**底层依赖HDFS，当磁盘空间不足的时候，只需要动态增加DataNode服务节点就可以
- **高并发：**支持高并发的读写请求
- **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。
- **数据的多版本：**HBase表中的数据可以有多个版本值，默认情况下是根据版本号去区分，版本号就是插入数据的时间戳
- **数据类型单一：**所有的数据在HBase中是以字节数组进行存储

1.3 HBase的应用

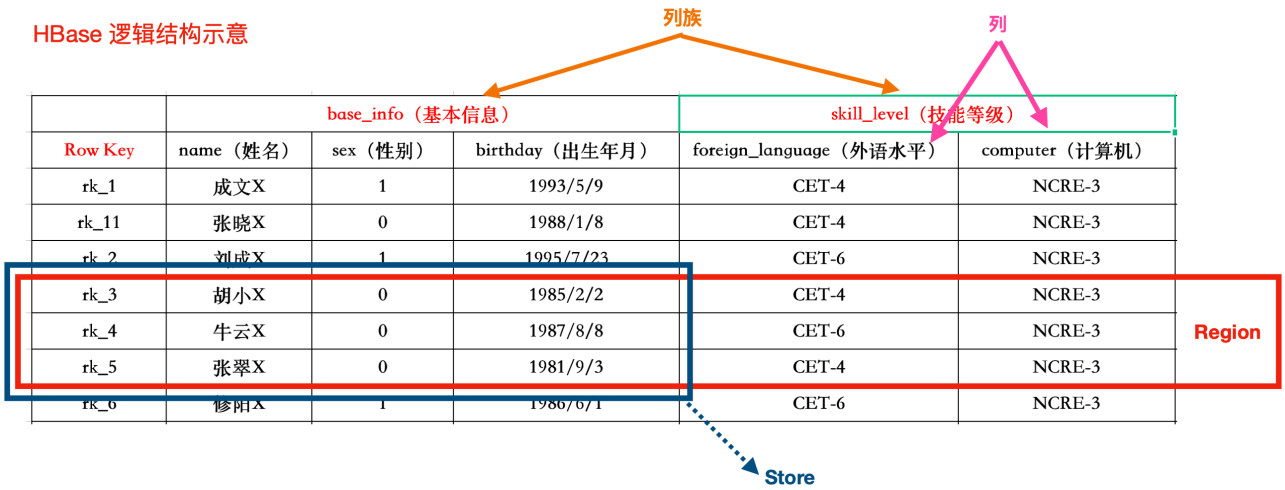
- 交通方面：船舶GPS信息，每天有上千万左右的数据存储。
- 金融方面：消费信息、贷款信息、信用卡还款信息等
- 电商方面：电商网站的交易信息、物流信息、游览信息等
- 电信方面：通话信息

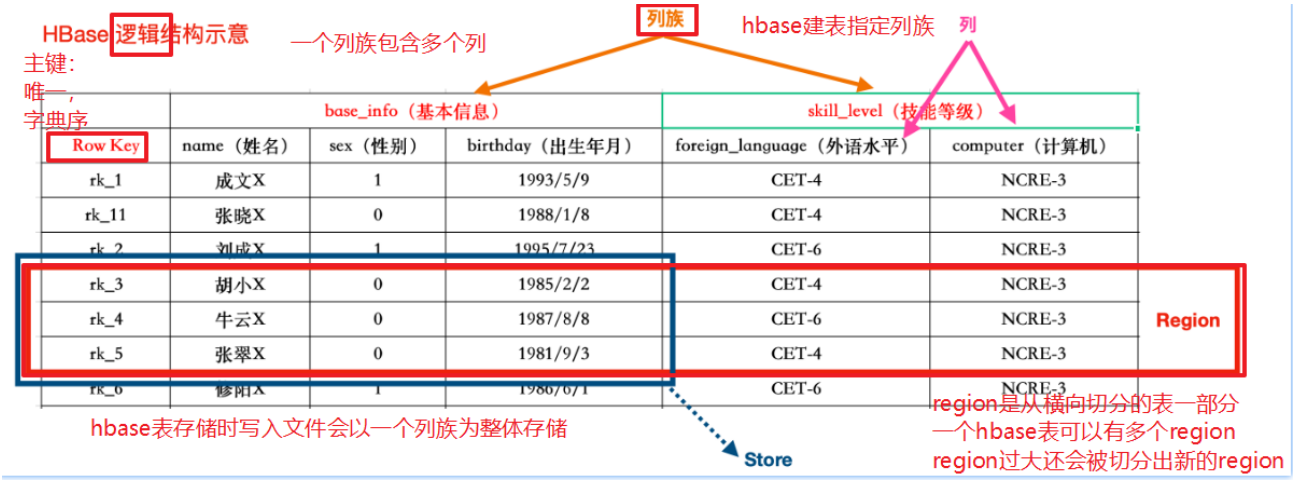
  总结：HBase适合海量明细数据的存储，并且后期需要有很好的查询性能（单表超千万、上亿，且并发要求高）

第 2 节 HBase数据模型

HBase的数据也是以表（有行有列）的形式存储

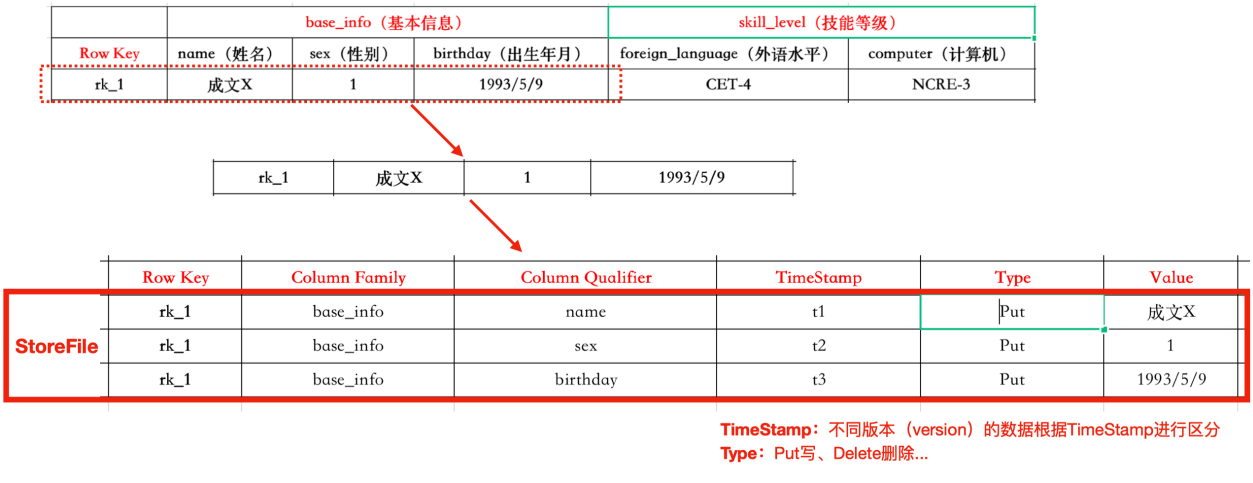
HBase逻辑架构





HBase物理存储

HBase 物理存储示意



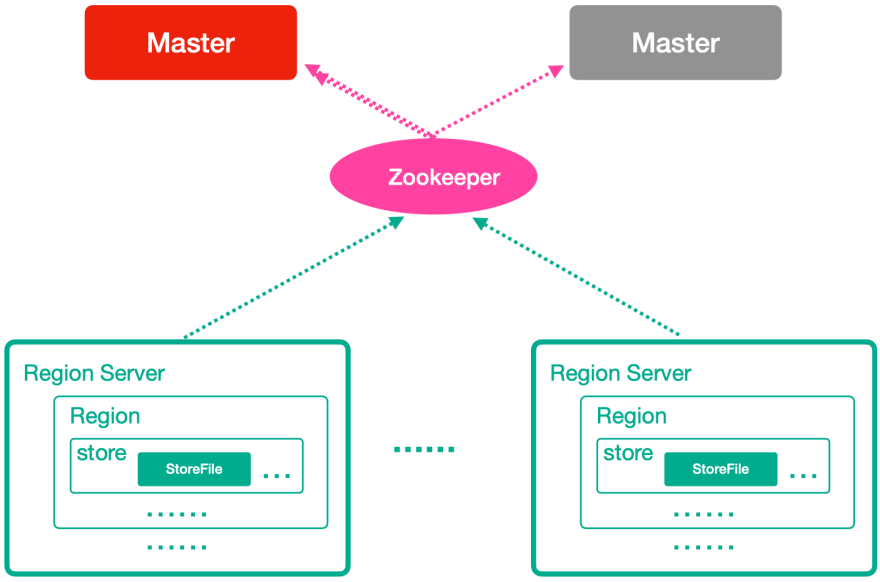
概念	描述
	命名空间，类似于关系型数据库的database概念，每个命名空间下有多个表。HBase两个自带的命名空间，分别是hbase和default，hbase中存放的是HBase内置的表，default表是用户默认使用的命名空间。一个表可以自由选择是否有命名空间，如果创建表的时候加上了命名空间后，这个表名字以:作为区分！

Table	类似于关系型数据库的表概念。不同的是，HBase定义表时只需要声明列族即可，数据属性，比如超时时间（TTL），压缩算法（COMPRESSION）等，都在列族的定义中定义，不需要声明具体的列。
Row（一行逻辑数据）	HBase表中的每行数据都由一个RowKey和多个Column（列）组成。一个行包含了多个列，这些列通过列族来分类,行中的数据所属列族只能从该表所定义的列族中选取,不能定义这个表中不存在的列族，否则报错NoSuchColumnFamilyException。
RowKey（每行数据主键）	Rowkey由用户指定的一串不重复的字符串定义，是一行的**唯一**标识！数据是按照RowKey的**字典顺序**存储的，并且查询数据时只能根据RowKey进行检索，所以RowKey的设计十分重要。如果使用了之前已经定义的RowKey，那么会将之前的数据更新掉！
Column Family（列族）	列族是多个列的集合。一个列族可以动态地灵活定义多个列。表的相关属性大部分都定义在列族上，同一个表里的不同列族可以有完全不同的属性配置，但是同一个列族内的所有列都会有相同的属性。列族存在的意义是HBase会把相同列族的列尽量放在同一台机器上，所以说，如果想让某几个列被放到一起，你就给他们定义相同的列族。
Column Qualifier（列）	Hbase中的列是可以随意定义的，一个行中的列不限名字、不限数量，只限定列族。因此列必须依赖于列族存在！列的名称前必须带着其所属的列族！例如info：name，info：age
TimeStamp（时间戳--》版本）	用于标识数据的不同版本（version）。时间戳默认由系统指定，也可以由用户显式指定。在读取单元格的数据时，版本号可以省略，如果不指定，Hbase默认会获取最后一个版本的数据返回！
Cell	一个列中可以存储多个版本的数据。而每个版本就称为一个单元格（Cell）。
Region（表的分区）	Region由一个表的若干行组成！在Region中行的排序按照行键（rowkey）字典排序。Region不能跨RegionSever，且当数据量大的时候，HBase会拆分Region。

第 3 节 HBase整体架构

HBase 架构示意

Master作用———>
Table：create，delete，alter
RegionServer：分配Regions到每个RegionServer，监控每个RegionServer的状态



RegionServer作用———>
Data：get，put，delete
Region：splitRegion，compactRegion

Zookeeper

- 实现了HMaster的高可用
  - 保存了HBase的元数据信息，是所有HBase表的寻址入口
- 对HMaster和HRegionServer实现了监控

HMaster（Master）

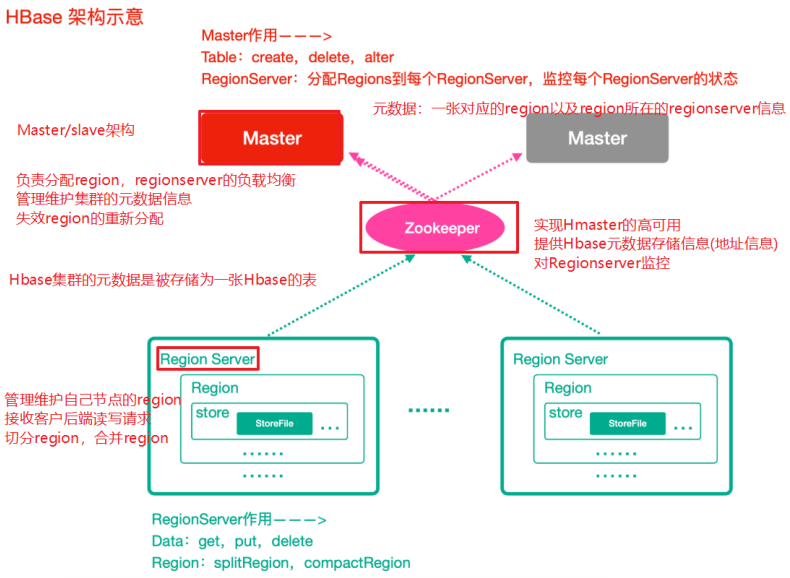
- 为HRegionServer分配Region
  - 维护整个集群的负载均衡
- 维护集群的元数据信息
- 发现失效的Region，并将失效的Region分配到正常的HRegionServer上

HRegionServer（RegionServer）

- 负责管理Region
- 接受客户端的读写数据请求
- 切分在运行过程中变大的Region

Region

- 每个HRegion由多个Store构成，
- 每个Store保存一个列族（Columns Family），表有几个列族，则有几个Store，
- 每个Store由一个MemStore和多个StoreFile组成，MemStore是Store在内存中的内容，写到文件后就是StoreFile。StoreFile底层是以HFile的格式保存。



第 4 节 HBase集群安装部署

（1）下载安装包

- http://archive.apache.org/dist/hbase/1.3.1/
- hbase-1.3.1-bin.tar.gz

（2）规划安装目录

```
/opt/lagou/servers/
```

（3）上传安装包到服务器

（4）解压安装包到指定的规划目录

```
tar -zxvf hbase-1.3.1-bin.tar.gz -C /opt/lagou/servers
```

（5）修改配置文件

- 需要把hadoop中的配置core-site.xml 、hdfs-site.xml拷贝到hbase安装目录下的conf文件夹中

```
ln -s /opt/lagou/servers/hadoop-2.9.2/etc/hadoop/core-site.xml /opt/lagou/servers/hbase-1.3.1/conf/core-site.xml
ln -s /opt/lagou/servers/hadoop-2.9.2/etc/hadoop/hdfs-site.xml /opt/lagou/servers/hbase-1.3.1/conf/hdfs-site.xml
```

- 修改conf目录下配置文件
  - 修改 hbase-env.sh

```
#添加java环境变量
export JAVA_HOME=/opt/module/jdk1.8.0_231
#指定使用外部的zk集群
export HBASE_MANAGES_ZK=FALSE
```

  - 修改 hbase-site.xml

```
<configuration>
          <!-- 指定hbase在HDFS上存储的路径 -->
        <property>
                <name>hbase.rootdir</name>
                <value>hdfs://linux121:9000/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>linux121:2181,linux122:2181,linux123:2181</value>
        </property>
</configuration>
```

  - 修改regionservers文件

```
#指定regionserver节点
linux121
linux122
linux123
```

- hbase的conf目录下创建文件backup-masters (Standby Master)

```
linux122
```

（6）配置hbase的环境变量

```
export HBASE_HOME=/opt/lagou/servers/hbase-1.3.1
export PATH=$PATH:$HBASE_HOME/bin
```

（7）分发hbase目录和环境变量到其他节点

```
rsync-script hbase-1.3.1
```

（8）让所有节点的hbase环境变量生效

- 在所有节点执行 source /etc/profile

HBase集群的启动和停止

- 前提条件：先启动hadoop和zk集群
- 启动HBase：start-hbase.sh
- 停止HBase：stop-hbase.sh

HBase集群的web管理界面

启动好HBase集群之后，可以访问地址：HMaster的主机名:16010

第 5 节 HBase shell 基本操作

1、进入Hbase客户端命令操作界面

```
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):006:0> list
```

4、创建一张lagou表， 包含base_info、extra_info两个列族

```
hbase(main):001:0> create 'lagou', 'base_info', 'extra_info'

或者(Hbase建表必须指定列族信息)

create 'lagou', {NAME => 'base_info', VERSIONS => '3'},{NAME => 'extra_info',VERSIONS => '3'}

    VERSIONS 是指此单元格内的数据可以保留最近的 3 个版本
```

5、添加数据操作

- 向lagou表中插入信息，row key为 rk1，列族base_info中添加name列标示符，值为wang

```
hbase(main):001:0> put 'lagou', 'rk1', 'base_info:name', 'wang'
```

- 向lagou表中插入信息，row key为rk1，列族base_info中添加age列标示符，值为30

```
hbase(main):001:0>  put 'lagou', 'rk1', 'base_info:age', 30
```

- 向lagou表中插入信息，row key为rk1，列族extra_info中添加address列标示符，值为shanghai

```
hbase(main):001:0> put 'lagou', 'rk1', 'extra_info:address', 'shanghai'
```

6、查询数据

6.1 通过rowkey进行查询

- 获取表中row key为rk1的所有信息

```
hbase(main):001:0> get 'lagou', 'rk1'
```

6.2 查看rowkey下面的某个列族的信息

- 获取lagou表中row key为rk1，base_info列族的所有信息

```
hbase(main):001:0> get 'lagou', 'rk1', 'base_info'
```

6.3 查看rowkey指定列族指定字段的值

- 获取表中row key为rk1，base_info列族的name、age列标示符的信息

```
hbase(main):008:0> get 'lagou', 'rk1', 'base_info:name', 'base_info:age'
```

6.4 查看rowkey指定多个列族的信息

- 获取lagou表中row key为rk1，base_info、extra_info列族的信息

```
hbase(main):010:0> get 'lagou', 'rk1', 'base_info', 'extra_info'

或者

hbase(main):011:0> get 'lagou', 'rk1', {COLUMN => ['base_info', 'extra_info']}

或者
hbase(main):012:0> get 'lagou', 'rk1', {COLUMN => ['base_info:name', 'extra_info:address']}
```

6.5 指定rowkey与列值查询

- 获取表中row key为rk1，cell的值为wang的信息

```
hbase(main):001:0> get 'lagou', 'rk1', {FILTER => "ValueFilter(=, 'binary:wang')"}
```

6.6 指定rowkey与列值模糊查询

- 获取表中row key为rk1，列标示符中含有a的信息

```
hbase(main):001:0> get 'lagou', 'rk1', {FILTER => "(QualifierFilter(=,'substring:a'))"}
```

6.7 查询所有数据

- 查询lagou表中的所有信息

```
hbase(main):000:0> scan 'lagou'
```

6.8 列族查询

- 查询表中列族为 base_info 的信息

```
hbase(main):001:0> scan 'lagou', {COLUMNS => 'base_info'}

hbase(main):002:0> scan 'lagou', {COLUMNS => 'base_info', RAW => true, VERSIONS => 3}

## Scan时可以设置是否开启Raw模式,开启Raw模式会返回包括已添加删除标记但是未实际删除的数据
## VERSIONS指定查询的最大版本数
```

6.9 指定多个列族与按照数据值模糊查询

- 查询lagou表中列族为 **base_info** 和 **extra_info**且列标示符中含有a字符的信息

```
hbase(main):001:0> scan 'lagou', {COLUMNS => ['base_info', 'extra_info'], FILTER => "
(QualifierFilter(=,'substring:a'))"}
```

6.10 rowkey的范围值查询（非常重要）

- 查询lagou表中列族为base_info，rk范围是[rk1, rk3)的数据（rowkey底层存储是字典序）
- 按rowkey顺序存储。

```
hbase(main):001:0> scan 'lagou', {COLUMNS => 'base_info', STARTROW => 'rk1', ENDROW => 'rk3'}
```

6.11 指定rowkey模糊查询

- 查询lagou表中row key以rk字符开头的

```
hbase(main):001:0> scan 'lagou',{FILTER=>"PrefixFilter('rk')"}
```

7、更新数据

- 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

7.1 更新数据值

- 把lagou表中rowkey为rk1的base_info列族下的列name修改为liang

```
hbase(main):030:0> put 'lagou', 'rk1', 'base_info:name', 'liang'
```

8、删除数据和表

8.1 指定rowkey以及列名进行删除

- 删除lagou表row key为rk1，列标示符为 base_info:name 的数据

```
hbase(main):002:0>  delete 'lagou', 'rk1', 'base_info:name'
```

8.2 指定rowkey，列名以及字段值进行删除

- 删除lagou表row key为rk1，列标示符为base_info:name的数据

```
hbase(main):033:0> delete 'lagou', 'rk1', 'base_info:age'
```

8.3 删除列族

- 删除 base_info 列族

```
hbase(main):001:0>

hbase(main):035:0> alter 'lagou', 'delete' => 'base_info'
```

8.4 清空表数据

- 删除lagou表数据

```
hbase(main):001:0> truncate 'lagou'
```

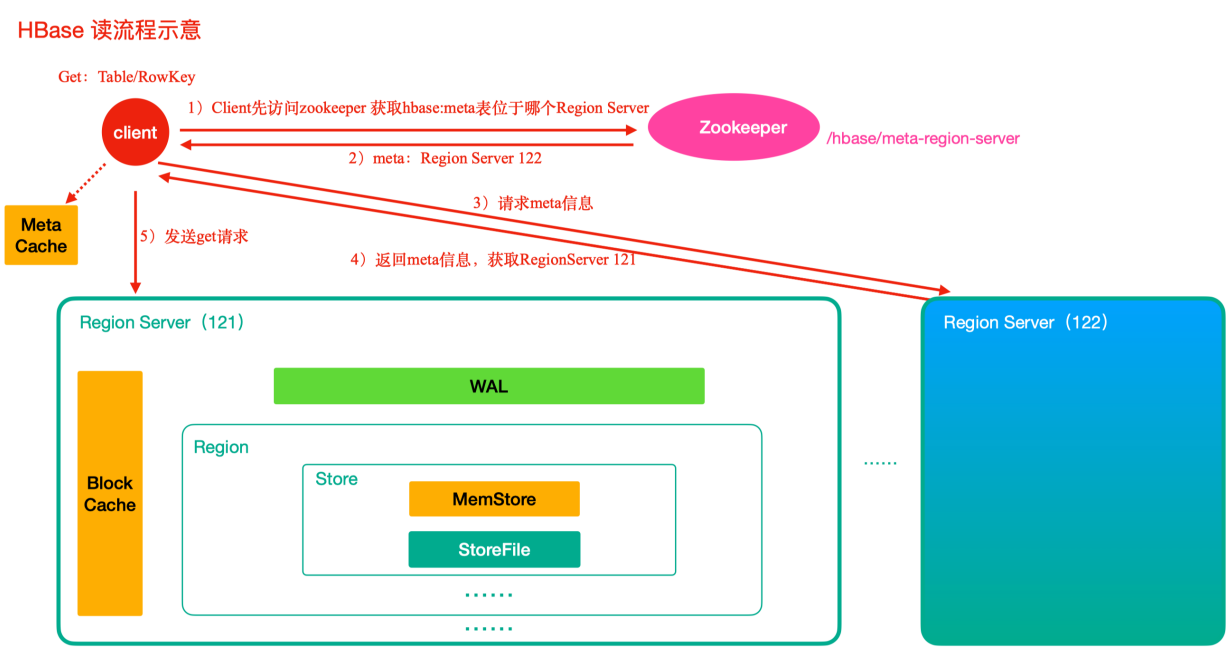
8.5 删除表

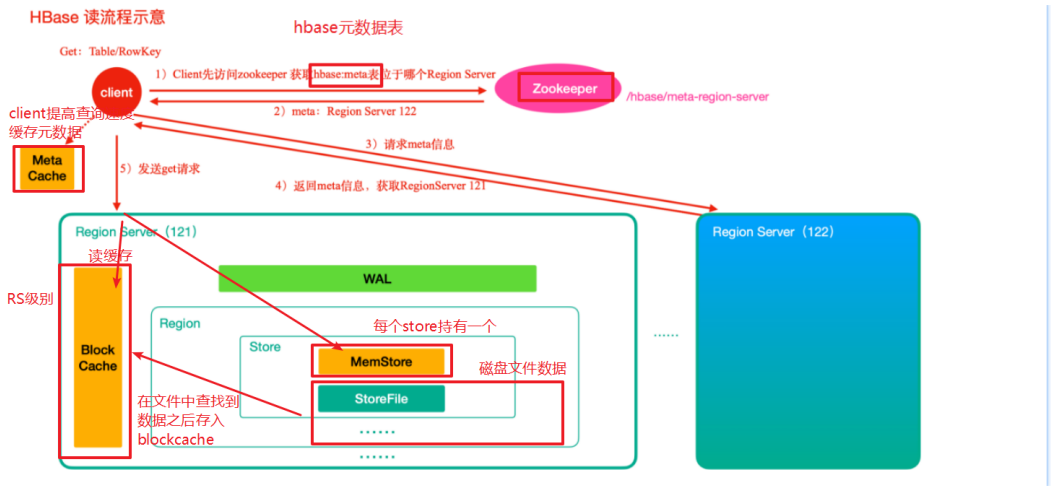
- 删除lagou表

```
#先disable  再drop

hbase(main):036:0> disable 'lagou'
hbase(main):037:0> drop 'lagou'

#如果不进行disable，直接drop会报错
ERROR: Table user is enabled. Disable it first.
```

第二部分 HBase原理深入

第 1 节 HBase读数据流程





```
=> ["lagou"]
hbase(main):031:0> scan 'hbase:meta'
ROW                                   COLUMN+CELL
 hbase:namespace,,159522 column=info:regioninfo, timestamp=1595229626056, value={ENCODED => 1
 9625085.16af75b35e3bc03 6af75b35e3bc03592935e4102998c4e, NAME => 'hbase:namespace,,159522962
 592935e4102998c4e.      5085.16af75b35e3bc03592935e4102998c4e.', STARTKEY => '', ENDKEY => '
                         '}
 hbase:namespace,,159522 column=info:seqnumDuringOpen, timestamp=1595229626056, value=\x00\x0
 9625085.16af75b35e3bc03 0\x00\x00\x00\x00\x00\x02
 592935e4102998c4e.
 hbase:namespace,,159522 column=info:server, timestamp=1595229626056, value=linux121:16020
 9625085.16af75b35e3bc03
 592935e4102998c4e.
 hbase:namespace,,159522 column=info:serverstartcode, timestamp=1595229626056, value=15952296
 9625085.16af75b35e3bc03 13268
 592935e4102998c4e.
 lagou,,1587372070330.24 column=info:regioninfo, timestamp=1595234412743, value={ENCODED => 2
 aaa6a3e856a6f98d53cfc60 4aaa6a3e856a6f98d53cfc6047f091d, NAME => 'lagou,,1587372070330.24aaa
 47f091d.                6a3e856a6f98d53cfc6047f091d.', STARTKEY => '', ENDKEY => ''}
 lagou,,1587372070330.24 column=info:seqnumDuringOpen, timestamp=1595234412743, value=\x00\x0
 aaa6a3e856a6f98d53cfc60 0\x00\x00\x00\x00\x00\x02
 47f091d.
 lagou,,1587372070330.24 column=info:server, timestamp=1595234412743, value=linux122:16020
 aaa6a3e856a6f98d53cfc60
 47f091d.
 lagou,,1587372070330.24 column=info:serverstartcode, timestamp=1595234412743, value=15952296
 aaa6a3e856a6f98d53cfc60 11536
 47f091d.
2 row(s) in 0.0530 seconds
```

hbase表中rowkey按照字典序排序，切分region就是按照rowkey来划分
一张表3个region
0-1000：region1
1001-2000:region2
2001-5000:region3

- **HBase读操作**

  1）首先从zk找到meta表的region位置，然后读取meta表中的数据，meta表中存储了用户表的region信息

  2）根据要查询的namespace、表名和rowkey信息。找到写入数据对应的region信息

  3）找到这个region对应的regionServer，然后发送请求

  4）查找对应的region

  5）先从memstore查找数据，如果没有，再从BlockCache上读取

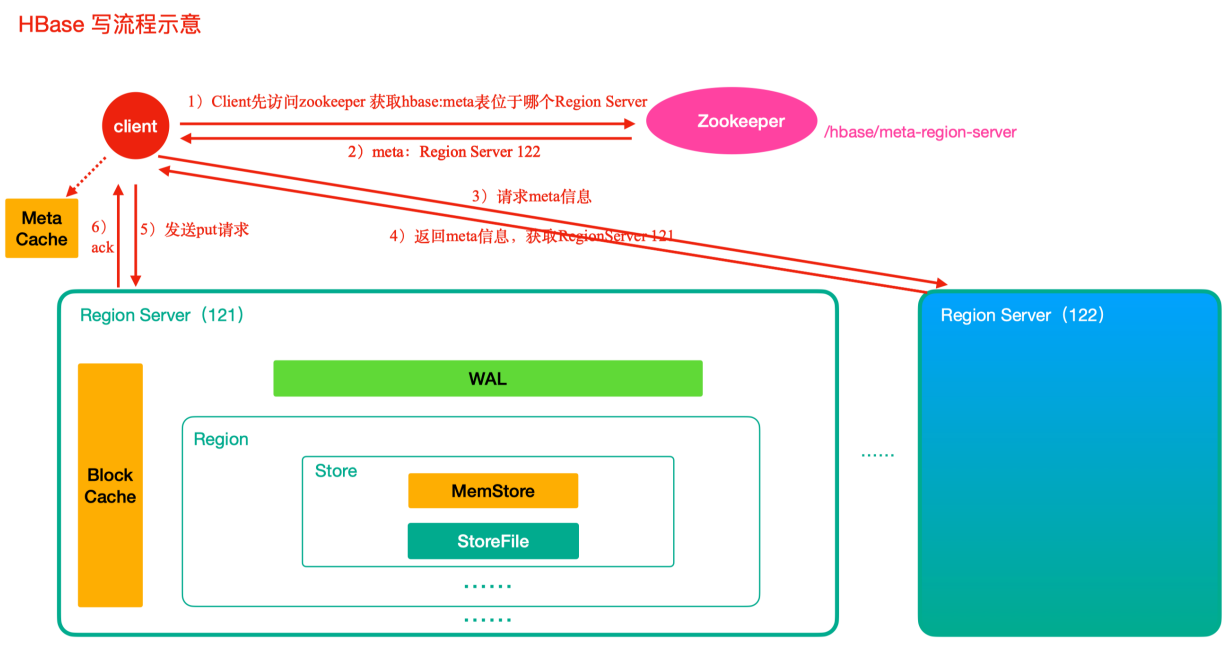
  HBase上Regionserver的内存分为两个部分

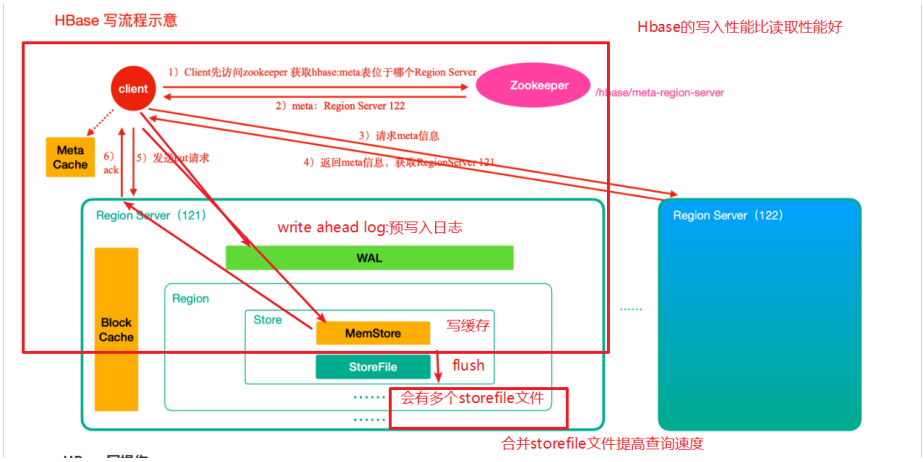
  - 一部分作为Memstore，主要用来写；
  - 另外一部分作为BlockCache，主要用于读数据；

  6）如果BlockCache中也没有找到，再到StoreFile上进行读取

  从storeFile中读取到数据之后，不是直接把结果数据返回给客户端， 而是把数据先写入到BlockCache中，目的是为了加快后续的查询；然后在返回结果给客户端。

第 2 节 HBase写数据流程





- **HBase写操作**

  1）首先从zk找到meta表的region位置，然后读取meta表中的数据，meta表中存储了用户表的region信息

  2）根据namespace、表名和rowkey信息。找到写入数据对应的region信息

  3）找到这个region对应的regionServer，然后发送请求

  4）把数据分别写到HLog（write ahead log）和memstore各一份

  5）memstore达到阈值后把数据刷到磁盘，生成storeFile文件

  6）删除HLog中的历史数据

第 3 节 HBase的flush(刷写)及compact(合并)机制

Flush机制

- （1）当memstore的大小超过这个值的时候，会flush到磁盘,默认为128M

```
<property>
  <name>hbase.hregion.memstore.flush.size</name>
  <value>134217728</value>
</property>
```

- （2）当memstore中的数据时间超过1小时，会flush到磁盘

```
<property>
  <name>hbase.regionserver.optionalcacheflushinterval</name>
  <value>3600000</value>
</property>
```

- （3）HregionServer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%

```
<property>
  <name>hbase.regionserver.global.memstore.size</name>
  <value>0.4</value>
</property>
```

- （4）手动flush

```
flush tableName
```

阻塞机制

以上介绍的是Store中memstore数据刷写磁盘的标准，但是Hbase中是周期性的检查是否满足以上标准满足则进行刷写，但是如果在下次检查到来之前，数据疯狂写入Memstore中，会出现什么问题呢?

会触发阻塞机制，此时无法写入数据到Memstore，数据无法写入Hbase集群。

- memstore中数据达到512MB

  计算公式：hbase.hregion.memstore.flush.size*hbase.hregion.memstore..block.multiplier

  hbase.hregion.memstore.flush.size刷写的阀值，默认是 134217728，即128MB。

  hbase.hregion.memstore.block.multiplier是一个倍数，默认 是4。

- RegionServer全部memstore达到规定值

  hbase.regionserver.global.memstore.size.lower.limit是0.95,

  hbase.regionserver.global.memstore.size是0.4,

  堆内存总共是 16G,

  触发刷写的阈值是：6.08GB 触发阻塞的阈值是：6.4GB

Compact合并机制

- 在hbase中主要存在两种类型的compac合并
  - **minor compact 小合并**
    - 在将Store中多个HFile(StoreFile)合并为一个HFile

      这个过程中，删除和更新的数据仅仅只是做了标记，并没有物理移除，这种合并的触发频率很高。

    - minor compact文件选择标准由以下几个参数共同决定：

```
<!--待合并文件数据必须大于等于下面这个值-->
<property>
  <name>hbase.hstore.compaction.min</name>
  <value>3</value>
</property>

<!--待合并文件数据必须小于等于下面这个值-->
<property>
  <name>hbase.hstore.compaction.max</name>
  <value>10</value>
```

```
</property>

<!--默认值为128m,
表示文件大小小于该值的store file 一定会加入到minor compaction的store file中
-->
<property>
  <name>hbase.hstore.compaction.min.size</name>
  <value>134217728</value>
</property>


<!--默认值为LONG.MAX_VALUE,
表示文件大小大于该值的store file 一定会被minor compaction排除-->
<property>
  <name>hbase.hstore.compaction.max.size</name>
  <value>9223372036854775807</value>
</property>
```

触发条件

- **memstore flush**

  在进行memstore flush前后都会进行判断是否触发compact

- **定期检查线程**

  周期性检查是否需要进行compaction操作,由参数：hbase.server.thread.wakefrequency决定，默认值是10000 millseconds

- **major compact 大合并**
  - 合并Store中所有的HFile为一个HFile

    这个过程有删除标记的数据会被真正移除，同时超过单元格maxVersion的版本记录也会被删除。合并频率比较低，默认7天执行一次，并且性能消耗非常大，建议生产关闭(设置为0)，在应用空闲时间手动触发。一般可以是手动控制进行合并，防止出现在业务高峰期。

    - major compaction触发时间条件

```
<!--默认值为7天进行一次大合并，-->
<property>
  <name>hbase.hregion.majorcompaction</name>
  <value>604800000</value>
</property>
```

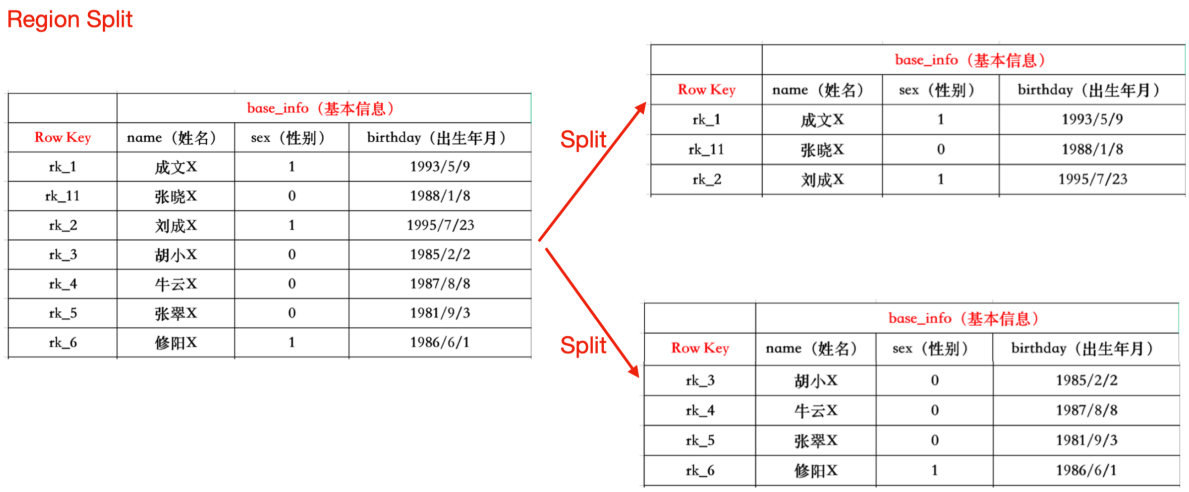
    - 手动触发

```
##使用major_compact命令
major_compact tableName
```

第 4 节 Region 拆分机制

Region中存储的是大量的rowkey数据 ,当Region中的数据条数过多的时候,直接影响查询效率.当Region过大的时候.HBase会拆分Region , 这也是Hbase的一个优点 .

NameSpace（数据库）



4.1 拆分策略

HBase的Region Split策略一共有以下几种：

1）ConstantSizeRegionSplitPolicy

0.94版本前默认切分策略

当region大小大于某个阈值(hbase.hregion.max.filesize=10G)之后就会触发切分，一个region等分为2个region。

但是在生产线上这种切分策略却有相当大的弊端：切分策略对于大表和小表没有明显的区分。阈值(hbase.hregion.max.filesize)设置较大对大表比较友好，但是小表就有可能不会触发分裂，极端情况下可能就1个，这对业务来说并不是什么好事。如果设置较小则对小表友好，但一个大表就会在整个集群产生大量的region，这对于集群的管理、资源使用、failover来说都不是一件好事。

2）IncreasingToUpperBoundRegionSplitPolicy

0.94版本~2.0版本默认切分策略

切分策略稍微有点复杂，总体看和ConstantSizeRegionSplitPolicy思路相同，一个region大小大于设置阈值就会触发切分。但是这个阈值并不像ConstantSizeRegionSplitPolicy是一个固定的值，而是会在一定条件下不断调整，调整规则和region所属表在当前regionserver上的region个数有关系.

region split的计算公式是：
regioncount^3 * 128M * 2，当region达到该size的时候进行split
例如：
第一次split：1^3 * 256 = 256MB
第二次split：2^3 * 256 = 2048MB
第三次split：3^3 * 256 = 6912MB
第四次split：4^3 * 256 = 16384MB > 10GB，因此取较小的值10GB
后面每次split的size都是10GB了

3）SteppingSplitPolicy

2.0版本默认切分策略

这种切分策略的切分阈值又发生了变化，相比 IncreasingToUpperBoundRegionSplitPolicy 简单了一些，依然和待分裂region所属表在当前regionserver上的region个数有关系，如果region个数等于1，
切分阈值为flush size * 2，否则为MaxRegionFileSize。这种切分策略对于大集群中的大表、小表会比IncreasingToUpperBoundRegionSplitPolicy 更加友好，小表不会再产生大量的小region，而是适可而止。

4）KeyPrefixRegionSplitPolicy

根据rowKey的前缀对数据进行分组，这里是指定rowKey的前多少位作为前缀，比如rowKey都是16位的，指定前5位是前缀，那么前5位相同的rowKey在进行region split的时候会分到相同的region中。

5）DelimitedKeyPrefixRegionSplitPolicy

保证相同前缀的数据在同一个region中，例如rowKey的格式为：userid_eventtype_eventid，指定的delimiter为 _ ，则split的的时候会确保userid相同的数据在同一个region中。

6）DisabledRegionSplitPolicy

不启用自动拆分, 需要指定手动拆分

4.2 RegionSplitPolicy的应用

Region拆分策略可以全局统一配置，也可以为单独的表指定拆分策略。

1）通过hbase-site.xml全局统一配置(对hbase所有表生效)

```
<property>

  <name>hbase.regionserver.region.split.policy</name>

  <value>org.apache.hadoop.hbase.regionserver.IncreasingToUpperBoundRegionSplitPolicy</value>

</property>
```

2）通过Java API为单独的表指定Region拆分策略

```
HTableDescriptor tableDesc = new HTableDescriptor("test1");

tableDesc.setValue(HTableDescriptor.SPLIT_POLICY, IncreasingToUpperBoundRegionSplitPolicy.class.getName());

tableDesc.addFamily(new HColumnDescriptor(Bytes.toBytes("cf1")));

admin.createTable(tableDesc);
```

3）通过HBase Shell为单个表指定Region拆分策略

```
hbase> create 'test2', {METADATA => {'SPLIT_POLICY' =>
'org.apache.hadoop.hbase.regionserver.IncreasingToUpperBoundRegionSplitPolicy'}},{NAME => 'cf1'}
```

第 5 节 HBase表的预分区(region)

5.1 为何要预分区?

当一个table刚被创建的时候，Hbase默认的分配一个region给table。也就是说这个时候，所有的读写请求都会访问到同一个regionServer的同一个region中，这个时候就达不到负载均衡的效果了，集群中的其他regionServer就可能会处于比较空闲的状态。解决这个问题可以用pre-splitting,在创建table的时候就配置好，生成多个region。

- 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

  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

5.2 手动指定预分区

```
create 'person','info1','info2',SPLITS => ['1000','2000','3000']
```

也可以把分区规则创建于文件中

```
vim split.txt
```

文件内容

```
aaa
bbb
ccc
ddd
```

执行

```
create 'student','info',SPLITS_FILE => '/root/hbase/split.txt'
```

第 6 节 Region 合并

6.1 Region合并说明

Region的合并不是为了性能，而是出于维护的目的。

6.2 如何进行Region合并

通过Merge类冷合并Region

- 需要先关闭hbase集群
- 需求：需要把student表中的2个region数据进行合并：student,,1593244870695.10c2df60e567e73523a633f20866b4b5.
  student,1000,1593244870695.0a4c3ff30a98f79ff6c1e4cc927b3d0d.

这里通过org.apache.hadoop.hbase.util.Merge类来实现，不需要进入hbase shell，直接执行（需要先关闭hbase集群）：

```
hbase org.apache.hadoop.hbase.util.Merge student \
student,,1595256696737.fc3eff4765709e66a8524d3c3ab42d59. \
student,aaa,1595256696737.1d53d6c1ce0c1bed269b16b6514131d0.
```

通过online_merge热合并Region

- 不需要关闭hbase集群，在线进行合并

```
与冷合并不同的是，online_merge的传参是Region的hash值，而Region的hash值就是Region名称的最后那段在两个.之间的字符串部分。

需求：需要把lagou_s表中的2个region数据进行合并：
student,,1587392159085.9ca8689901008946793b8d5fa5898e06. \
student,aaa,1587392159085.601d5741608cedb677634f8f7257e000.

需要进入hbase shell：

merge_region 'c8bc666507d9e45523aebaffa88ffdd6','02a9dfdf6ff42ae9f0524a3d8f4c7777'
```

- 成功后观察界面

第三部分 HBase API应用和优化

第 1 节 HBase API客户端操作

创建Maven工程，添加依赖

```
<dependencies>
        <dependency>
            <groupId>org.apache.hbase</groupId>
            <artifactId>hbase-client</artifactId>
            <version>1.3.1</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>
```

```
public class HbaseClientDemo {
    Configuration   conf=null;
    Connection conn=null;
    HBaseAdmin admin =null;
@Before
public void init () throws IOException {

          conf = HBaseConfiguration.create();
          conf.set("hbase.zookeeper.quorum","linux121,linux122");
          conf.set("hbase.zookeeper.property.clientPort","2181");
            conn = ConnectionFactory.createConnection(conf);

  }


  public void destroy(){
        if(admin!=null){
            try {
                admin.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        if(conn !=null){
            try {
                conn.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
  }
}
```

创建表：

```
  @Test
  public void createTable() throws IOException {
      admin = (HBaseAdmin) conn.getAdmin();
      //创建表描述器
      HTableDescriptor teacher = new HTableDescriptor(TableName.valueOf("teacher"));
      //设置列族描述器
      teacher.addFamily(new HColumnDescriptor("info"));
      //执行创建操作
      admin.createTable(teacher);
      System.out.println("teacher表创建成功！！");
  }
```

插入数据

```
//插入一条数据
        @Test
    public  void putData() throws IOException {

        //获取一个表对象
        Table t = conn.getTable(TableName.valueOf("teacher"));
        //设定rowkey
        Put put = new Put(Bytes.toBytes("110"));
        //列族，列，value
        put.addColumn(Bytes.toBytes("info"), Bytes.toBytes("addr"), Bytes.toBytes("beijing"));
        //执行插入
        t.put(put);
//        t.put();//可以传入list批量插入数据
```

```
        //关闭table对象
        t.close();
        System.out.println("插入成功！！");
    }
```

删除数据：

```
   //删除一条数据
    @Test
    public void deleteData() throws IOException {
        //需要获取一个table对象
        final Table worker = conn.getTable(TableName.valueOf("worker"));

        //准备delete对象
        final Delete delete = new Delete(Bytes.toBytes("110"));
//执行删除
        worker.delete(delete);
        //关闭table对象
        worker.close();
        System.out.println("删除数据成功！！");
    }
```

查询某个列族数据

```
//查询某个列族数据
    @Test
    public  void getDataByCF() throws IOException {

        //获取表对象
        HTable teacher = (HTable) conn.getTable(TableName.valueOf("teacher"));
        //创建查询的get对象
        Get get = new Get(Bytes.toBytes("110"));
        //指定列族信息
//        get.addColumn(Bytes.toBytes("info"), Bytes.toBytes("sex"));
        get.addFamily(Bytes.toBytes("info"));
        //执行查询
        Result res = teacher.get(get);
        Cell[] cells = res.rawCells();//获取改行的所有cell对象
        for (Cell cell :
                cells) {
            //通过cell获取rowkey,cf,column,value
            String cf = Bytes.toString(CellUtil.cloneFamily(cell));
            String column = Bytes.toString(CellUtil.cloneQualifier(cell));
            String value = Bytes.toString(CellUtil.cloneValue(cell));
            String rowkey = Bytes.toString(CellUtil.cloneRow(cell));
            System.out.println(rowkey + "----" + cf + "---" + column + "---" + value);

        }

        teacher.close();//关闭表对象资源
    }
```

通过Scan全表扫描

```
  /**
     * 全表扫描
     */
    @Test
    public void scanAllData() throws IOException {

        HTable teacher = (HTable) conn.getTable(TableName.valueOf("teacher"));

        Scan scan = new Scan();
        ResultScanner resultScanner = teacher.getScanner(scan);
        for (Result result : resultScanner) {
            Cell[] cells = result.rawCells();//获取改行的所有cell对象
            for (Cell cell : cells) {
                //通过cell获取rowkey,cf,column,value
                String cf = Bytes.toString(CellUtil.cloneFamily(cell));
                String column = Bytes.toString(CellUtil.cloneQualifier(cell));
                String value = Bytes.toString(CellUtil.cloneValue(cell));
                String rowkey = Bytes.toString(CellUtil.cloneRow(cell));
                System.out.println(rowkey + "----" + cf + "--" + column + "---" + value);

            }
```

```
        }
        teacher.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 {
         HTable teacher = (HTable) conn.getTable(TableName.valueOf("teacher"));

        Scan scan = new Scan();
        scan.setStartRow("0001".getBytes());
        scan.setStopRow("2".getBytes());
        ResultScanner resultScanner = teacher.getScanner(scan);
        for (Result result : resultScanner) {
            Cell[] cells = result.rawCells();//获取改行的所有cell对象
            for (Cell cell : cells) {
                //通过cell获取rowkey,cf,column,value
                String cf = Bytes.toString(CellUtil.cloneFamily(cell));
                String column = Bytes.toString(CellUtil.cloneQualifier(cell));
                String value = Bytes.toString(CellUtil.cloneValue(cell));
                String rowkey = Bytes.toString(CellUtil.cloneRow(cell));
                System.out.println(rowkey + "----" + cf + "--" + column + "---" + value);

            }
        }
        teacher.close();
    }
```

第 2 节 Hbase 协处理器

2.1 协处理器概述

官方地址

```
http://hbase.apache.org/book.html#cp
```

访问HBase的方式是使用scan或get获取数据，在获取到的数据上进行业务运算。但是在数据量非常大的时候，比如一个有上亿行及十万个列的数据集，再按常用的方式移动获取数据就会遇到性能问题。客户端也需要有强大的计算能力以及足够的内存来处理这么多的数据。

此时就可以考虑使用Coprocessor(协处理器)。将业务运算代码封装到Coprocessor中并在RegionServer上运行，即在数据实际存储位置执行，最后将运算结果返回到客户端。利用协处理器，用户可以编写运行在 HBase Server 端的代码。

Hbase Coprocessor类似以下概念

触发器和存储过程：一个Observer Coprocessor有些类似于关系型数据库中的触发器，通过它我们可以在一些事件（如Get或是Scan）发生前后执行特定的代码。Endpoint Coprocessor则类似于关系型数据库中的存储过程，因为它允许我们在RegionServer上直接对它存储的数据进行运算，而非是在客户端完成运算。

MapReduce：MapReduce的原则就是将运算移动到数据所处的节点。Coprocessor也是按照相同的原则去工作的。

AOP：如果熟悉AOP的概念的话，可以将Coprocessor的执行过程视为在传递请求的过程中对请求进行了拦截，并执行了一些自定义代码。

2.2 协处理器类型

Observer

协处理器与触发器(trigger)类似：在一些特定事件发生时回调函数（也被称作钩子函数，hook）被执行。这些事件包括一些用户产生的事件，也包括服务器端内部自动产生的事件。

协处理器框架提供的接口如下

RegionObserver：用户可以用这种的处理器处理数据修改事件，它们与表的region联系紧密。

MasterObserver：可以被用作管理或DDL类型的操作，这些是集群级事件。

WALObserver：提供控制WAL的钩子函数

Endpoint

这类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 协处理器在Regionserver中执行一段代码，并将 RegionServer 端执行结果返回给客户端进一步处理。

Endpoint常见用途

聚合操作

假设需要找出一张表中的最大数据，即 max 聚合操作，普通做法就是必须进行全表扫描，然后Client代码内遍历扫描结果，并执行求最大值的操作。这种方式存在的弊端是无法利用底层集群的并发运算能力，把所有计算都集中到 Client 端执行,效率低下。

使用Endpoint Coprocessor，用户可以将求最大值的代码部署到 HBase RegionServer 端，HBase 会利用集群中多个节点的优势来并发执行求最大值的操作。也就是在每个 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Client。在Client进一步将多个 Region 的最大值汇总进一步找到全局的最大值。

Endpoint Coprocessor的应用我们后续可以借助于Phoenix非常容易就能实现。针对Hbase数据集进行聚合运算直接使用SQL语句就能搞定。

2.3 Observer 案例

需求

通过协处理器Observer实现Hbase当中t1表插入数据,指定的另一张表t2也需要插入相对应的数据。

```
create 't1','info'

create 't2','info'
```

实现思路

通过Observer协处理器捕捉到t1插入数据时，将数据复制一份并保存到t2表中

开发步骤

1. 编写Observer协处理器

```
package com.lagou.coprocessor;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.Cell;
import org.apache.hadoop.hbase.CoprocessorEnvironment;
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.filter.ByteArrayComparable;
import org.apache.hadoop.hbase.filter.CompareFilter;
import org.apache.hadoop.hbase.regionserver.wal.WALEdit;

import java.io.IOException;

public class MyProcessor extends BaseRegionObserver {


    @Override
    public void prePut(ObserverContext<RegionCoprocessorEnvironment> ce, Put put, WALEdit edit, Durability
durability) throws IOException {
        final HTable t2 = (HTable)ce.getEnvironment().getTable(TableName.valueOf("t2"));
        Cell nameCell = put.get("info".getBytes(), "name".getBytes()).get(0);
        Put put1 = new Put(put.getRow());
        put1.add(nameCell);
        t2.put(put);
        t2.close();
    }


}
```

添加依赖

```
<!-- https://mvnrepository.com/artifact/org.apache.hbase/hbase-server -->
        <dependency>
            <groupId>org.apache.hbase</groupId>
            <artifactId>hbase-server</artifactId>
            <version>1.3.1</version>
        </dependency>
```

2. 打成Jar包，上传HDFS

```
cd /opt/lagou/softwares
mv original-hbaseStudy-1.0-SNAPSHOT.jar  processor.jar
hdfs dfs -mkdir -p /processor
hdfs dfs -put processor.jar /processor
```

3. 挂载协处理器

```
hbase(main):056:0> describe 't1'
hbase(main):055:0> alter 't1',METHOD =>
'table_att','Coprocessor'=>'hdfs://linux121:9000/processor/processor.jar|com.lagou.hbase.processor.MyProcessor|1001|'
#再次查看't1'表,
hbase(main):043:0> describe 't1'
```

4.验证协处理器

向t1表中插入数据(shell方式验证)

```
put 't1','rk1','info:name','lisi'
```

5.卸载协处理器

```
disable 't1'
alter 't1',METHOD=>'table_att_unset',NAME=>'coprocessor$1'
enable 't2'
```

第 4 节 HBase表的RowKey设计

RowKey的基本介绍

ASCII码字典顺序。

012,0,123,234,3.

0,3,012,123,234

0,012,123,234,3

字典序的排序规则。

先比较第一个字节，如果相同，然后比对第二个字节，以此类推，

如果到第X个字节，其中一个已经超出了rowkey的长度，短rowkey排在前面。

2.1 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
  - 设计过长会降低memstore内存的利用率和HFile存储数据的效率。

2.2 RowKey散列原则

建议将rowkey的高位作为散列字段，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。

2.3 RowKey唯一原则

必须在设计上保证其唯一性， 访问hbase table中的行：有3种方式：

- 单个rowkey
- rowkey 的range
- 全表扫描(一定要避免全表扫描)

实现方式：

1）org.apache.hadoop.hbase.client.Get

2）scan方法： org.apache.hadoop.hbase.client.Scan

scan使用的时候注意：

- setStartRow，setEndRow 限定范围， 范围越小，性能越高。

2.4 RowKey排序原则

HBase的Rowkey是按照ASCII有序设计的，我们在设计Rowkey时要充分利用这点.

第 5 节 HBase表的热点

5.1 什么是热点

检索habse的记录首先要通过row key来定位数据行。当大量的client访问hbase集群的一个或少数几个节点，造成少数region server的读/写请求过多、负载过大，而其他region server负载却很小，就造成了“热点”现象

5.2 热点的解决方案

- 预分区

  > 预分区的目的让表的数据可以均衡的分散在集群中，而不是默认只有一个region分布在集群的一个节点上。

- 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。

4个region，[,a),[a,b),[b,c),[c,]

原始数据：abc1,abc2,abc3.

加盐后的rowkey：a-abc1,b-abc2,c-abc3

abc1,a

abc2,b

- 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。

原始数据： abc1，abc2,abc3

哈希：

md5（abc1）=92231b....., 9223-abc1

md5(abc2) =32a131122...., 32a1-abc2

md5(abc3) = 452b1...., 452b-abc3.

- 反转

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。

15X,13X,

第 6 节 HBase的二级索引

HBase表按照rowkey查询性能是最高的。rowkey就相当于hbase表的一级索引！！

为了HBase的数据查询更高效、适应更多的场景，诸如使用非rowkey字段检索也能做到秒级响应，或者支持各个字段进行模糊查询和多字段组合查询等， 因此需要在HBase上面构建二级索引， 以满足现实中更复杂多样的业务需求。

hbase的二级索引其本质就是建立hbase表中列与行键之间的映射关系。

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

第 7 节 布隆过滤器在hbase的应用

- **布隆过滤器应用**

之前再讲hbase的数据存储原理的时候，我们知道hbase的读操作需要访问大量的文件，大部分的实现通过布隆过滤器来避免大量的读文件操作。

- **布隆过滤器的原理**

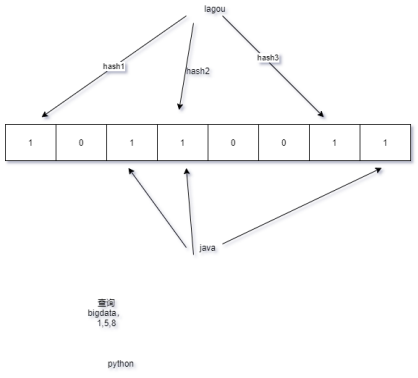
通常判断某个元素是否存在用的可以选择hashmap。但是 HashMap 的实现也有缺点，例如存储容量占比高，考虑到负载因子的存在，通常空间是不能被用满的，而一旦你的值很多例如上亿的时候，那 HashMap 占据的内存大小就变得很可观了。

Bloom Filter是一种空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。

hbase 中布隆过滤器来过滤指定的rowkey是否在目标文件，避免扫描多个文件。使用布隆过滤器来判断。

布隆过滤器返回true,在结果不一定争取，、如果返回false则说明确实不存在。

- **原理示意图**



- **Bloom Filter案例**

布隆过滤器，已经不需要自己实现，Google已经提供了非常成熟的实现。

```
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>27.0.1-jre</version>
</dependency>
```

使用

guava 的布隆过滤器，封装的非常好，使用起来非常简洁方便。

例： 预估数据量1w，错误率需要减小到万分之一。使用如下代码进行创建。

```
public static void main(String[] args) {
    // 1.创建符合条件的布隆过滤器
    // 预期数据量10000，错误率0.0001
    BloomFilter<CharSequence> bloomFilter =
            BloomFilter.create(Funnels.stringFunnel(
                    Charset.forName("utf-8")),10000, 0.0001);
    // 2.将一部分数据添加进去
    for (int i = 0; i < 5000; i++) {
        bloomFilter.put("" + i);
    }
    System.out.println("数据写入完毕");
    // 3.测试结果
    for (int i = 0; i < 10000; i++) {
        if (bloomFilter.mightContain("" + i)) {
            System.out.println(i + "存在");
        } else {
            System.out.println(i + "不存在");
        }
    }
}
```