第一部分 初识hbase

1. hbase简介

1.1 hbase是什么

hbase是建立的hdfs之上分布式数据库。

hdfs的缺点。

mysql：

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

mysql浪费存储空间

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

列的设计

列簇的设计

列簇中的列存储在一起。

name列：rowkey：1 name ： 小明

name列 : rowkey:2 name:小红

age列：---- rowkey：1 age：23

work列簇 rowkkey：1 job：学生

work列簇 rowkey：2 salary：10w

work列簇 rowkkey：2 job：律师

1.2 hbase的概念

- hbase基于Google的BigTable论文，是建立的hdfs之上，提供**高可靠性**、**高性能**、**列存储**、**可伸缩**、**实时读写**的分布式数据库系统。在需要实时读写随机访问超大规模数据集时，可以使用hbase。

1.3 hbase的特点

- **海量存储**
  - 可以存储大批量的数据 hdfs
- **列式存储**
  - hbase表的数据是基于列族进行存储的，列族是在列的方向上的划分。
- **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节点服务(机器)就可以了
  - 可以通过增加服务器来提高集群的存储能力
- **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

1.4 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：海量明细数据的存储，并且后期需要有很好的查询性能

2、hbase整体架构

- 1、**Client**
  - 客户端
    - Client包含了访问Hbase的接口
    - 另外Client还维护了对应的cache来加速Hbase的访问，比如cache的.META.元数据的信息
- 2、**Zookeeper**
  - zookeeper集群
    - 作用
      - 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
- 3、**HMaster**
  - 作用
    - 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

- 4、**HRegionServer**
    - 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
- 5、**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。

集群的高可用

多个集群，主从集群。

WAL

实现方法：hbase replication原理

hbase的高可用不支持DDL操作。在master上的DDL操作，不会影响slave上的数据。

3、hbase表的数据模型

rowkey	timestamp	CF1	CF2	CF3
11212	t6		CF2:q2=v2	CF3:q3=val3
11212	t3		CF2:q2=v1	
11211	t2	CF1:q1=v1		

- ==**rowkey**==

  决定一行数据，

  按照字典顺序排序

  只能存储64k字节数据，rowkey越短越好。
- ==**Column Family**==

列族：必须作为表模式定义的一部分预先给出来。

CF2：q1，CF2：q3,q100,新的列可以随后按需，动态加入。

权限控制，存储以及调优都在列簇层面进行。

- ==**Timestamp**==
  - 时间戳
    - 每次数据操作对应的时间戳，可以看作是数据的version number版本号
    - 64位整型
- ==**Column**==
  - 列
    - 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
- ==**cell**==
  - 单元格
    - 由{row key, column(= +), version} 唯一确定的单元
    - cell中的数据是没有类型的，全部是以字节数组进行存储
    - 内容是未解析的字节数组，没有数据类型。

4、hbase集群安装部署

- 1、下载安装包
  - http://archive.apache.org/dist/
  - hbase-1.3.1-bin.tar.gz
- 2、规划安装目录
  - /home/teacher/opt/module
- 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 /home/teacher/opt/module/

- 5、修改配置文件
  - 1、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

```
cd /home/teacher/opt/module/hadoop-2.7.2/etc/hadoop;

cp core-site.xml  hdfs-site.xml    /home/teacher/software/hbase-1.3.1/conf/
```

  - 2、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://teacher1:9000/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>teacher1:2181,teacher2:2181,teacher3:2181</value>
        </property>
</configuration>
```

- 修改regionservers文件

```
#指定regionserver节点
teacher1
teacher2
teacher3
```

- hbase的conf目录下创建文件backup-masters

```
teacher3
```

- 6、配置hbase的环境变量

```
export HBASE_HOME=/opt/bigdata/hbase
export PATH=$PATH:$HBASE_HOME/bin
```

- 7、分发hbase目录和环境变量到其他节点

```
scp -r hbase-1.3.1 teacher1:/home/teacher/opt/module/
scp -r hbase-1.3.1 teacher3:/home/teacher/opt/module/
```

- 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

- 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

5、hbase shell 命令基本操作

第二部分 hbase原理深入

1. hbase的数据存储原理

1、不同存储模型的应用情况

每一种数据存储系统，对应有一种存储模型，或者叫存储引擎。我们今天要介绍的是三种比较流行的存储模型，分别是：

1、Hash存储模型

- redis
- memcache

2、B-Tree存储模型

- MySQL（以及大多数的关系型数据库）

- MongoDB

3、LSM树存储模型

- HBase
- RocksDB

不同存储模型介绍

1、Hash存储模型

Hash存储模型其实就是HashMap（哈希表）的持久化实现。这种模型的特点是与HashMap有密切关系的。我们知道HashMap可以支持：put（key）增加/修改、delete（key）删除、get（key）随机获取操作

2、B-Tree存储模型

B-Tree存储模型由于是树状结构存储，所以，它是不支持随机读写的。就像我们学习二叉树时，查找数据得通过遍历树的方式来查找数据。

3、LSM树存储引擎

LSM树的思想很容易理解，就是将数据的新增、修改增量数据先保存在内存中，到达指定的大小限制后将修改操作批量写入到磁盘。读取时，需要合并磁盘中的历史数据和内存中最近的修改操作。LSM的优势在于有效地随机写入问题，但读取可能需要访问较多的磁盘文件。

- Level 0 ： 日志/内存
-
  - 先写入预写日志，再写内存
  - 写入日志是为了保障可用性
- Level 1：日志/内存，当Level 0写入达到阈值，通过异步方式将部分数据刷写到硬盘上
- Level 2：合并，由于不断刷写会产生大量小文件，这样不利于管理和查询。需要在合适的时机启动一个异步线程进行合并操作生成一个大文件

2、LSM算法原理

1、Log Structured Merge Trees(LSM) 介绍

LSM是当前被用在许多产品的文件结构策略：HBase, Cassandra, LevelDB, SQLite,甚至在mangodb3.0中也带了一个可选的LSM引擎（Wired Tiger 实现的）。

简单的说，LSM被设计来提供比传统的B+树更好的写操作吞吐量，通过消去随机的本地更新操作来达到这个目标。

磁盘随机操作慢，顺序读写快。

顺序读写磁盘，甚至快于随机读写内存。

设计数据库系统，数据落地到磁盘。以下几个方法：

（1） 写操作敏感的系统，简单的把数据追加到文件中。基于日志的策略，同时有一些缺点，从日志文件中读一些数据比写操作花费更多的时间，需要倒序扫描。

WAL

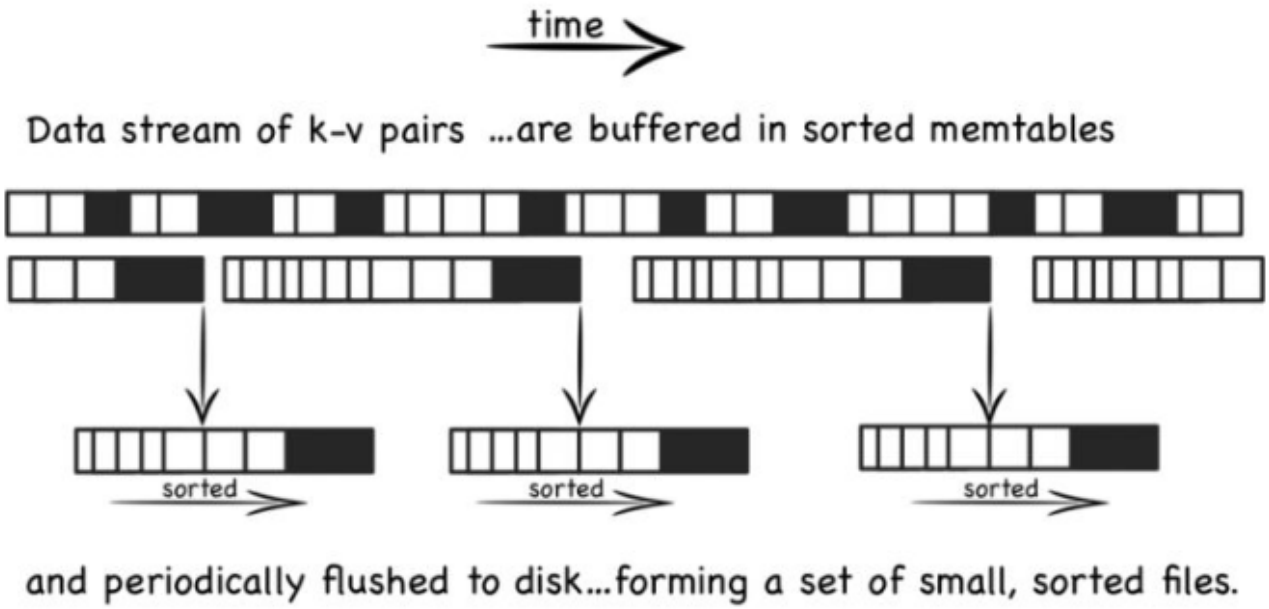
（2）B+树，logn，

哈希方法，

二分查找。

提供高效的读性能，写性能降低。

（3） lsm ，保持日志文件写的性能，读操作性能小。本质就是让所有的操作顺序化。



SSTable

The Google **SSTable** file format is used internally to store Bigtable data. An **SSTable** provides a **persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings.** Operations are provided to look up the value associated with a specified key, and to iterate over all key/value pairs in a specified key range

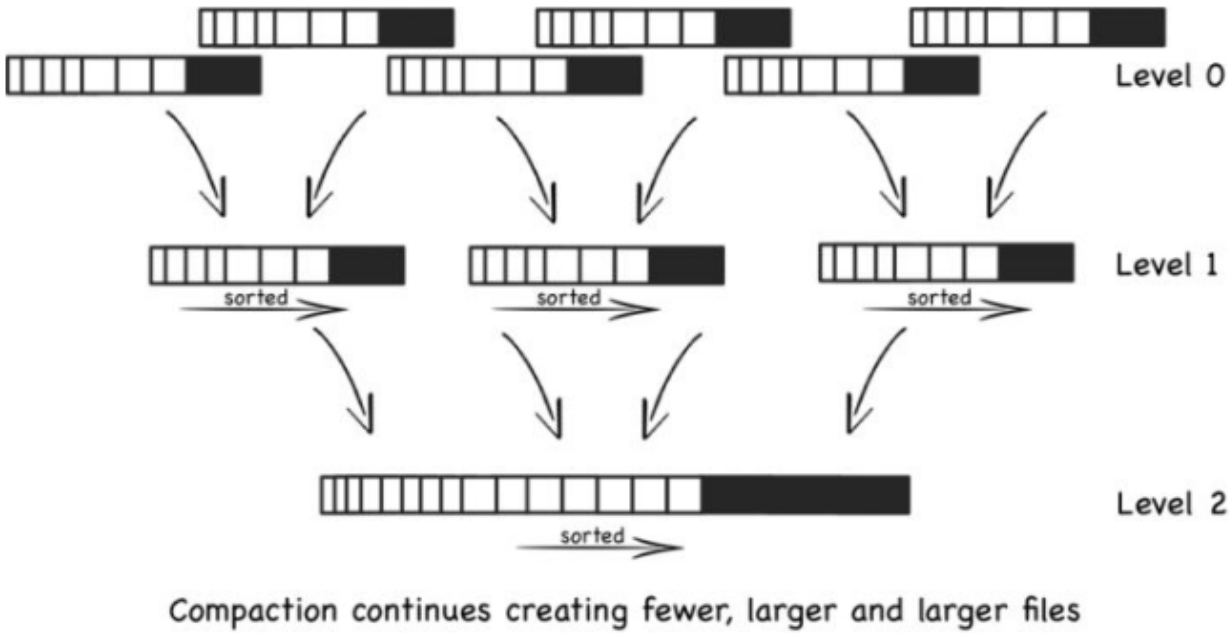
简单的非直译：

SSTable是Bigtable内部用于数据的文件格式，它的格式为文件本身就是一个排序的、不可变的、持久的Key/Value对Map，其中Key和value都可以是任意的byte字符串。使用Key来查找Value，或通过给定Key范围遍历所有的Key/Value对。

2、合并方法

1）基本合并：

最坏情况下，所有的文件都要搜索。



2）层级合并：

实现了一个分层的，而不是根据文件大小来执行合并操作。这个方法可以减少在最坏情况下需要检索的文件个数，同时也减少了一次合并操作的影响。

按层合并的策略相对于上述的按文件大小合并的策略有二个关键的不同：

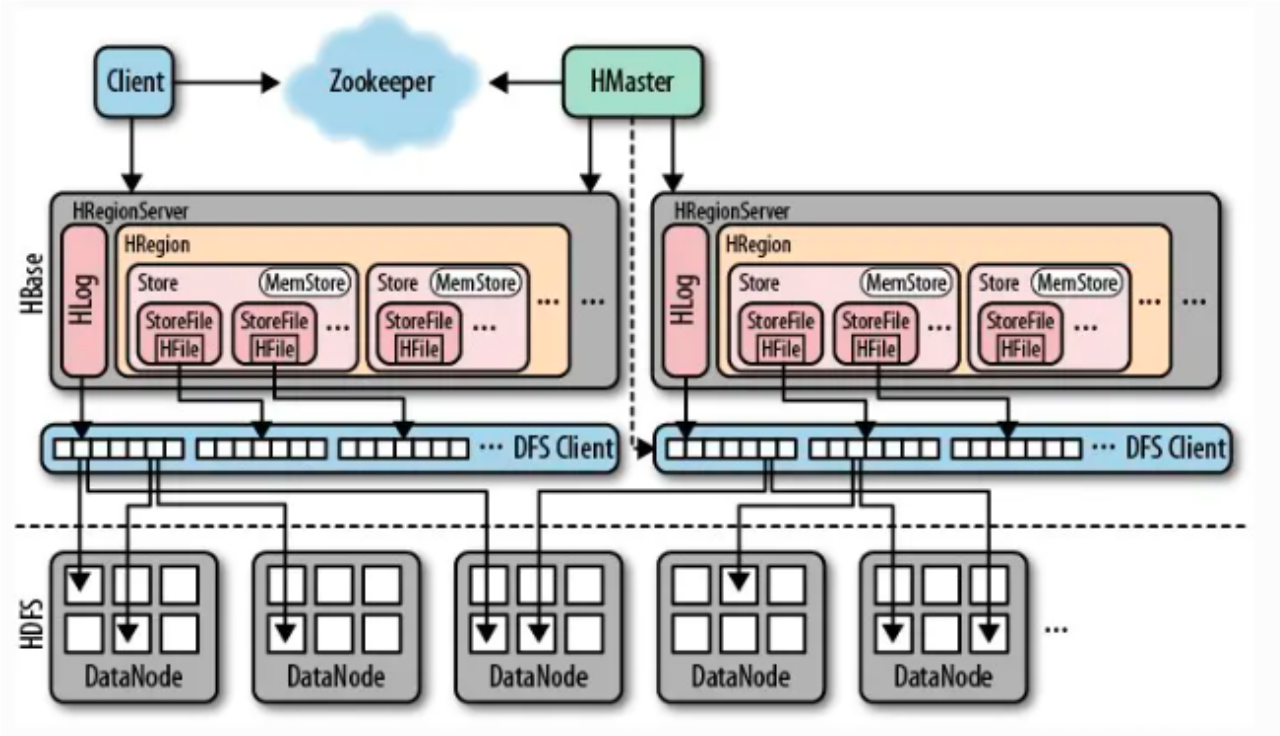
1. 每一层可以维护指定的文件个数，同时保证不让key重叠。也就是说把key分区到不同的文件。因此在一层查找一个key，只用查找一个文件。第一层是特殊情况，不满足上述条件，key可以分布在多个文件中。
2. 每次，文件只会被合并到上一层的一个文件。当一层的文件数满足特定个数时，一个文件会被选出并合并到上一层。这明显不同与另一种合并方式：一些相近大小的文件被合并为一个大文件。

总结：LSM是日志和传统的单文件索引（B+，）的综合。管理多个独立的索引文件（sstable）。

B+，随机IO昂贵

LSM：读取多个大量的sstable，让随机IO变得更快，一些IO被合并操作消耗。

2. hbase读数据流程



- **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。

3. 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

4. hbase的flush、compact机制

4.1 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
```

4.2 Compact合并机制

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

      > 这个过程中，达到TTL（记录保留时间）会被移除，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 compaction触发条件由一下几个参数共同决定：

```
<!--表示至少需要三个满足条件的store file时，minor compaction才会启动-->
<property>
    <name>hbase.hstore.compaction.min</name>
    <value>3</value>
</property>

<!--表示一次minor compaction中最多选取10个store file-->
<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>
```

- **major compaction 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
```

5. 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 .

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

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

```
<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 'test1', {METADATA => {'SPLIT_POLICY' =>
'org.apache.hadoop.hbase.regionserver.IncreasingToUpperBoundRegionSplitPolicy'}},{NAME => 'cf1'}
```

6. hbase表的预分区

6.1 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

手动指定预分区

```
create 'person','info1','info2',SPLITS => ['1000','2000','3000','4000']
```

- 也可以把分区规则创建于文件中

  ```
  cd /opt/bigdata/

  vim split.txt
  ```

  - 文件内容

    ```
    aaa
    bbb
    ccc
    ddd
    ```

  - 执行

    ```
    create 'student','info',SPLITS_FILE => '/opt/bigdata/split.txt'
    ```

HexStringSplit 算法

```
HexStringSplit会将数据从“00000000”到“FFFFFFFF”之间的数据长度按照n等分之后算出每一段的起始rowkey和结束rowkey，以此作为拆分点。

例如：
create 'mytable','base_info','extra_info',{NUMREGIONS => 15, SPLITALGO => 'HexStringSplit'}
```

7. region 合并

7.1 region合并说明

region的合并不是为了性能，而是出于维护的目的。

7.2 如何进行region合并

7.2.1 通过Merge类冷合并Region

- 需要先关闭hbase集群

```
创建一张hbase表：create 'test','info1',SPLITS => ['1000','2000','3000']

需求：需要把test表中的2个region数据进行合并：
test,,1565940912661.62d28d7d20f18debd2e7dac093bc09d8.
test,1000,1565940912661.5b6f9e8dad3880bcc825826d12e81436.

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 test
```

- 成功后界面观察

7.2.2 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test表中的2个region数据进行合并：
test,2000,1565940912661.c2212a3956b814a6f0d57a90983a8515.
test,3000,1565940912661.553dd4db667814cf2f050561167ca030.

需要进入hbase shell：

merge_region 'c2212a3956b814a6f0d57a90983a8515','553dd4db667814cf2f050561167ca030'
```

- 成功后观察界面

第三部分 hbase应用和优化

1. hbase客户端API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>
     </dependencies>
```

创建表：

```
//创建hbase表
public static void createTable() throws IOException {
    //获取连接对象
    Connection conn = ConnectionFactory.createConnection(conf);
    //通过连接获取hbase客户端对象
    HBaseAdmin admin = (HBaseAdmin) conn.getAdmin();
    //创建表描述器
    HTableDescriptor teacher = new HTableDescriptor(TableName.valueOf("teacher"));
    //设置列族描述器
    student1.addFamily(new HColumnDescriptor("info"));
    //执行创建操作
    admin.createTable(teacher);
    System.out.println("lagou1表创建成功！！");
}
```

插入数据

```
//插入一条数据
    public static void putData() throws IOException {
        //获取连接对象
        Connection conn = ConnectionFactory.createConnection(conf);
        //通过连接获取hbase客户端对象
        HBaseAdmin admin = (HBaseAdmin) conn.getAdmin();
        //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：

```
//删除一行数据
public static void deleteData() throws IOException {
    //获取连接对象
    Connection conn = ConnectionFactory.createConnection(conf);
    //通过连接获取hbase客户端对象
    HBaseAdmin admin = (HBaseAdmin) conn.getAdmin();
    //获取一个表对象
    Table t = conn.getTable(TableName.valueOf("teacher"));
    //创建一个delete对象
    Delete delete = new Delete(Bytes.toBytes("110"));
    t.delete(delete);
    System.out.println("删除指定数据成功！");
}


//删除多行数据
public static void deleteMultiRow(String tableName, String... rows) throws IOException{
        HTable hTable = new HTable(conf, tableName);
        List<Delete> deleteList = new ArrayList<Delete>();
        for(String row : rows){
                Delete delete = new Delete(Bytes.toBytes(row));
                deleteList.add(delete);
        }
        hTable.delete(deleteList);
        hTable.close();
}
```

获取所有数据：

```
//查询某个列族数据
    public static void scanDataByCF() throws IOException {
        //获取连接对象
        Connection conn = ConnectionFactory.createConnection(conf);
        //通过连接获取hbase客户端对象
        HBaseAdmin admin = (HBaseAdmin) conn.getAdmin();
        //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
    }
```

2. 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。

rowkey 100字节， 1000* 1000万 =10亿字节，将近1G数据。影响了HFile的存储效率。

64位 ，8字节的整数倍，利用操作系统的最佳特性。

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。如果没有散列字段，首字段直接是时间信息。

rowkey ， jvm full gc region too busy。

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：

1、单个rowkey 2 rowkey 的range 3 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使用的时候注意以下两点：

1）setCaching，setBatch方法，以空间换时间

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

2.4rowkey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.

3. 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

- 1、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。

- 2、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

- 3、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.

- 4、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,

4、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.

hbase过滤器。

多条件组合查询的情况，

全表扫描，过滤器，选出目标数据，（很低效）

- hbase的二级索引
-

  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。

5.布隆过滤器在hbase的应用

1、**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的读操作需要倒序访问大量的文件，大部分的实现通过布隆过滤器来避免大量的读文件操作。

2 、**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。

3、**画图展示原理**