小吴蜀黍

HBase Rowkey的散列与预分区设计

HBase中，表会被划分为1...n个Region，被托管在RegionServer中。Region二个重要的属性:StartKey与EndKey表示这个Region维护的rowKey范围，当我们要读/写数据时，如果rowKey落在某个start-end key范围内，那么就会定位到目标region并且读/写到相关的数据。简单地说，有那么一点点类似人群划分，1-15岁为小朋友,16-39岁为年轻人，40-64为中年人,65岁以上为老年人。(这些数值都是拍脑袋出来的，只是举例，非真实),然后某人找队伍，然后根据年龄，处于哪个范围，就找到它所属的队伍。 :(有点废话了。。。。

然后，默认地，当我们只是通过HBaseAdmin指定TableDescriptor来创建一张表时，只有一个region,正处于混沌时期，start-end key无边界,可谓海纳百川。啥样的rowKey都可以接受，都往这个region里装，然而，当数据越来越多，region的size越来越大时，大到一定的阀值，hbase认为再往这个region里塞数据已经不合适了，就会找到一个midKey将region一分为二，成为2个region,这个过程称为分裂(region-split).而midKey则为这二个region的临界，左为N无下界，右为M无上界。< midKey则为阴被塞到N区，> midKey则会被塞到M区。

如何找到midKey?涉及的内容比较多，暂且不去讨论，最简单的可以认为是region的总行数 / 2 的那一行数据的rowKey.虽然实际上比它会稍复杂点。

如果我们就这样默认地，建表，表里不断地Put数据，更严重的是我们的rowkey还是顺序增大的，是比较可怕的。存在的缺点比较明显。

首先是热点写，我们总是会往最大的start-key所在的region写东西，因为我们的rowkey总是会比之前的大，并且hbase的是按升序方式排序的。所以写操作总是被定位到无上界的那个region中。

其次，由于写热点，我们总是往最大start-key的region写记录，之前分裂出来的region不会再被写数据，有点被打进冷宫的赶脚，它们都处于半满状态，这样的分布也是不利的。

如果在写比较频率的场景下，数据增长快，split的次数也会增多，由于split是比较耗时耗资源的，所以我们并不希望这种事情经常发生。

.....

看到这些缺点，我们知道，在集群的环境中，为了得到更好的并行性，我们希望有好的load blance，让每个节点提供的请求处理都是均等的。我们也希望，region不要经常split，因为split会使server有一段时间的停顿，如何能做到呢？
随机散列与预分区。二者结合起来，是比较完美的，预分区一开始就预建好了一部分region,这些region都维护着自己的start-end keys，再配合上随机散列，写数据能均等地命中这些预建的region，就能解决上面的那些缺点，大大地提高了性能。

提供2种思路: hash 与 partition.

hash就是rowkey前面由一串随机字符串组成,随机字符串生成方式可以由SHA或者MD5等方式生成，只要region所管理的start-end keys范围比较随机，那么就可以解决写热点问题。

```
long currentId = 1L;
byte [] rowkey = Bytes.add(MD5Hash.getMD5AsHex(Bytes.toBytes(currentId)).substring(0, 8).getBytes(),
                    Bytes.toBytes(currentId));
```

假设rowKey原本是自增长的long型，可以将rowkey转为hash再转为bytes，加上本身id 转为bytes,组成rowkey，这样就生成随便的rowkey。那么对于这种方式的rowkey设计，如何去进行预分区呢？

1.取样，先随机生成一定数量的rowkey,将取样数据按升序排序放到一个集合里

2.根据预分区的region个数，对整个集合平均分割，即是相关的splitKeys.

3.HBaseAdmin.createTable(HTableDescriptor tableDescriptor,byte[][] splitkeys)可以指定预分区的splitKey，即是指定region间的rowkey临界值.

1.创建split计算器，用于从抽样数据中生成一个比较合适的splitKeys

```
public class HashChoreWoker implements SplitKeysCalculator{
    //随机取机数目
    private int baseRecord;
    //rowkey生成器
    private RowKeyGenerator rkGen;
    //取样时，由取样数目及region数相除所得的数量.
    private int splitKeysBase;
    //splitkeys个数
    private int splitKeysNumber;
    //由抽样计算出来的splitkeys结果
```

```
    private byte[][] splitKeys;

    public HashChoreWoker(int baseRecord, int prepareRegions) {
        this.baseRecord = baseRecord;
        //实例化rowkey生成器
        rkGen = new HashRowKeyGenerator();
        splitKeysNumber = prepareRegions - 1;
        splitKeysBase = baseRecord / prepareRegions;
    }

    public byte[][] calcSplitKeys() {
        splitKeys = new byte[splitKeysNumber][];
        //使用treeset保存抽样数据，已排序过
        TreeSet<byte[]> rows = new TreeSet<byte[]>(Bytes.BYTES_COMPARATOR);
        for (int i = 0; i < baseRecord; i++) {
            rows.add(rkGen.nextId());
        }
        int pointer = 0;
        Iterator<byte[]> rowKeyIter = rows.iterator();
        int index = 0;
        while (rowKeyIter.hasNext()) {
            byte[] tempRow = rowKeyIter.next();
            rowKeyIter.remove();
            if ((pointer != 0) && (pointer % splitKeysBase == 0)) {
                if (index < splitKeysNumber) {
                    splitKeys[index] = tempRow;
                    index ++;
                }
            }
            pointer ++;
        }
        rows.clear();
        rows = null;
        return splitKeys;
    }
}
```

KeyGenerator及实现

```
//interface
public interface RowKeyGenerator {
    byte [] nextId();
}
//implements
public class HashRowKeyGenerator implements RowKeyGenerator {
    private long currentId = 1;
    private long currentTime = System.currentTimeMillis();
    private Random random = new Random();
    public byte[] nextId() {
        try {
            currentTime += random.nextInt(1000);
            byte[] lowT = Bytes.copy(Bytes.toBytes(currentTime), 4, 4);
            byte[] lowU = Bytes.copy(Bytes.toBytes(currentId), 4, 4);
            return Bytes.add(MD5Hash.getMD5AsHex(Bytes.add(lowU, lowT)).substring(0, 8).getBytes(),
                    Bytes.toBytes(currentId));
        } finally {
            currentId++;
        }
```

```
    }
}
```

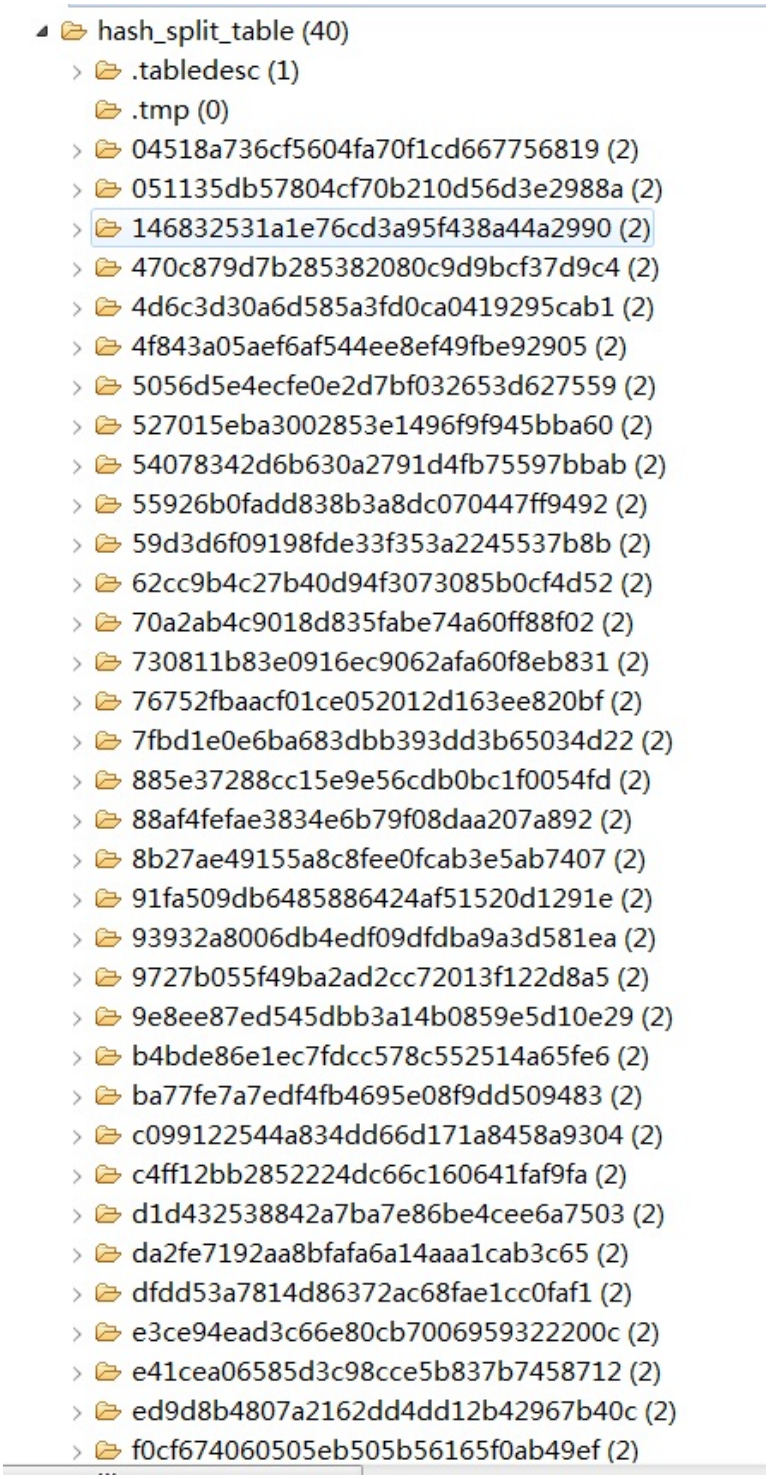
unit test case测试

```
@Test
public void testHashAndCreateTable() throws Exception{
        HashChoreWoker worker = new HashChoreWoker(1000000,10);
        byte [][] splitKeys = worker.calcSplitKeys();

        HBaseAdmin admin = new HBaseAdmin(HBaseConfiguration.create());
        TableName tableName = TableName.valueOf("hash_split_table");

        if (admin.tableExists(tableName)) {
            try {
                admin.disableTable(tableName);
            } catch (Exception e) {
            }
            admin.deleteTable(tableName);
        }

        HTableDescriptor tableDesc = new HTableDescriptor(tableName);
        HColumnDescriptor columnDesc = new HColumnDescriptor(Bytes.toBytes("info"));
        columnDesc.setMaxVersions(1);
        tableDesc.addFamily(columnDesc);

        admin.createTable(tableDesc ,splitKeys);

        admin.close();
    }
```

查看建表结果：执行 scan 'hbase:meta'

```
hbase(main):007:0* scan 'hbase:meta'
ROW                                   COLUMN+CELL
 hash_split_table,,1403363756244.146832 column=info:regioninfo, timestamp=1403363756993, value={ENCODED => 146832531a1e76cd3a95f438a44a2990, NAME => 'has
 531a1e76cd3a95f438a44a2990.           h_split_table,,1403363756244.146832531a1e76cd3a95f438a44a2990.', STARTKEY => '', ENDKEY => '06b539a4\x00\x00\x00\
                                       x00\x00\x0E\xEF\xE4'}
 hash_split_table,,1403363756244.146832 column=info:seqnumDuringOpen, timestamp=1403363757617, value=\x00\x00\x00\x00\x00\x00\x00\x01
 531a1e76cd3a95f438a44a2990.
 hash_split_table,,1403363756244.146832 column=info:server, timestamp=1403363757617, value=master:46734
 531a1e76cd3a95f438a44a2990.
 hash_split_table,,1403363756244.146832 column=info:serverstartcode, timestamp=1403363757617, value=1403362764737
 531a1e76cd3a95f438a44a2990.
 hash_split_table,06b539a4\x00\x00\x00\ column=info:regioninfo, timestamp=1403363756993, value={ENCODED => f0cf674060505eb505b56165f0ab49ef, NAME => 'has
 x00\x00\x0E\xEF\xE4,1403363756244.f0cf h_split_table,06b539a4\x00\x00\x00\x00\x00\x0E\xEF\xE4,1403363756244.f0cf674060505eb505b56165f0ab49ef.', STARTKEY
 674060505eb505b56165f0ab49ef.          => '06b539a4\x00\x00\x00\x00\x00\x0E\xEF\xE4', ENDKEY => '0d71c563\x00\x00\x00\x00\x00\x00\xB7\x14'}
 hash_split_table,06b539a4\x00\x00\x00\ column=info:seqnumDuringOpen, timestamp=1403363757603, value=\x00\x00\x00\x00\x00\x00\x00\x01
 x00\x00\x0E\xEF\xE4,1403363756244.f0cf
 674060505eb505b56165f0ab49ef.
 hash_split_table,06b539a4\x00\x00\x00\ column=info:server, timestamp=1403363757603, value=master:46734
 x00\x00\x0E\xEF\xE4,1403363756244.f0cf
 674060505eb505b56165f0ab49ef.
 hash_split_table,06b539a4\x00\x00\x00\ column=info:serverstartcode, timestamp=1403363757603, value=1403362764737
 x00\x00\x0E\xEF\xE4,1403363756244.f0cf
 674060505eb505b56165f0ab49ef.
 hash_split_table,0d71c563\x00\x00\x00\ column=info:regioninfo, timestamp=1403363756993, value={ENCODED => 70a2ab4c9018d835fabe74a60ff88f02, NAME => 'has
 x00\x00\x00\xB7\x14,1403363756244.70a2 h_split_table,0d71c563\x00\x00\x00\x00\x00\x00\xB7\x14,1403363756244.70a2ab4c9018d835fabe74a60ff88f02.', STARTKEY
 ab4c9018d835fabe74a60ff88f02.          => '0d71c563\x00\x00\x00\x00\x00\x00\xB7\x14', ENDKEY => '142e17a1\x00\x00\x00\x00\x00\x0F\x13i'}
 hash_split_table,0d71c563\x00\x00\x00\ column=info:seqnumDuringOpen, timestamp=1403363757086, value=\x00\x00\x00\x00\x00\x00\x00\x01
 x00\x00\x00\xB7\x14,1403363756244.70a2
 ab4c9018d835fabe74a60ff88f02.
 hash_split_table,0d71c563\x00\x00\x00\ column=info:server, timestamp=1403363757086, value=master:46734
 x00\x00\x00\xB7\x14,1403363756244.70a2
 ab4c9018d835fabe74a60ff88f02.
 hash_split_table,0d71c563\x00\x00\x00\ column=info:serverstartcode, timestamp=1403363757086, value=1403362764737
 x00\x00\x00\xB7\x14,1403363756244.70a2
 ab4c9018d835fabe74a60ff88f02.
 hash_split_table,142e17a1\x00\x00\x00\ column=info:regioninfo, timestamp=1403363756993, value={ENCODED => 93932a8006db4edf09dfdba9a3d581ea, NAME => 'has
 x00\x00\x0F\x13i,1403363756244.93932a8 h_split_table,142e17a1\x00\x00\x00\x00\x00\x0F\x13i,1403363756244.93932a8006db4edf09dfdba9a3d581ea.', STARTKEY =>
 006db4edf09dfdba9a3d581ea.             '142e17a1\x00\x00\x00\x00\x00\x0F\x13i', ENDKEY => '1ae3038d\x00\x00\x00\x00\x00\x08L\x19'}
 hash_split_table,142e17a1\x00\x00\x00\ column=info:seqnumDuringOpen, timestamp=1403363757646, value=\x00\x00\x00\x00\x00\x00\x00\x01
 x00\x00\x0F\x13i,1403363756244.93932a8
 006db4edf09dfdba9a3d581ea.
```

以上我们只是显示了部分region的信息，可以看到region的start-end key 还是比较随机散列的。同样可以查看hdfs的目录结构,的确和预期的38个预分区一致：



以上，就已经按hash方式，预建好了分区，以后在插入数据的时候，也要按照此rowkeyGenerator的方式生成rowkey,有兴趣的话，也可以做些试验，插入些数据，看看数据的分布。

partition故名思义，就是分区式，这种分区有点类似于mapreduce中的partitioner,将区域用长整数(Long)作为分区号，每个region管理着相应的区域数据，在rowKey生成时，将id取模后，然后拼上id整体作为rowKey.这个比较简单，不需要取样，splitKeys也非常简单，直接是分区号即可。直接上代码吧：

```
public class PartitionRowKeyManager implements RowKeyGenerator,
        SplitKeysCalculator {

    public static final int DEFAULT_PARTITION_AMOUNT = 20;
    private long currentId = 1;
```

```
    private int partition = DEFAULT_PARTITION_AMOUNT;
    public void setPartition(int partition) {
        this.partition = partition;
    }

    public byte[] nextId() {
        try {
            long partitionId = currentId % partition;
            return Bytes.add(Bytes.toBytes(partitionId),
                    Bytes.toBytes(currentId));
        } finally {
            currentId++;
        }
    }

    public byte[][] calcSplitKeys() {
        byte[][] splitKeys = new byte[partition - 1][];
        for(int i = 1; i < partition ; i ++) {
            splitKeys[i-1] = Bytes.toBytes((long)i);
        }
        return splitKeys;
    }
}
```

calcSplitKeys方法比较单纯，splitKey就是partition的编号,我们看看测试类:

```
@Test
    public void testPartitionAndCreateTable() throws Exception{

        PartitionRowKeyManager rkManager = new PartitionRowKeyManager();
        //只预建10个分区
        rkManager.setPartition(10);

        byte [][] splitKeys = rkManager.calcSplitKeys();

        HBaseAdmin admin = new HBaseAdmin(HBaseConfiguration.create());
        TableName tableName = TableName.valueOf("partition_split_table");

        if (admin.tableExists(tableName)) {
            try {
                admin.disableTable(tableName);

            } catch (Exception e) {
            }
            admin.deleteTable(tableName);
        }

        HTableDescriptor tableDesc = new HTableDescriptor(tableName);
        HColumnDescriptor columnDesc = new HColumnDescriptor(Bytes.toBytes("info"));
        columnDesc.setMaxVersions(1);
        tableDesc.addFamily(columnDesc);

        admin.createTable(tableDesc ,splitKeys);

        admin.close();
    }
```

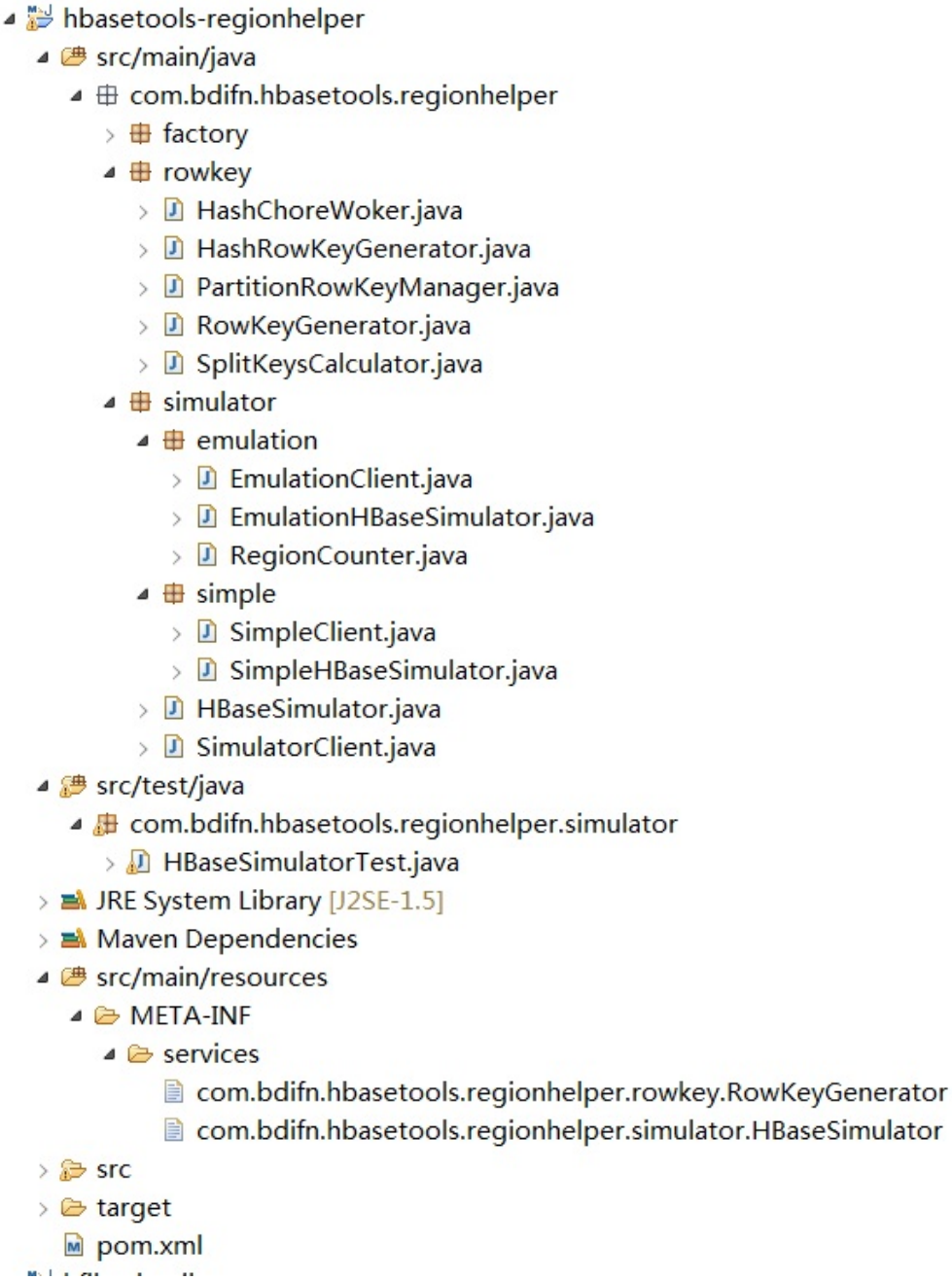
同样我们可以看看meta表和hdfs的目录结果，其实和hash类似，region都会分好区，在这里就不上图了。

通过partition实现的loadblance写的话，当然生成rowkey方式也要结合当前的region数目取模而求得，大家同样也可以做些实验，看看数据插入后的分布。
在这里也顺提一下，如果是顺序的增长型原id,可以将id保存到一个数据库，传统的也好,redis的也好，每次取的时候，将数值设大1000左右，以后id可以在内存内增长，当内存数量已经超过1000的话，再去load下一个，有点类似于oracle中的sqeuence.

随机分布加预分区也不是一劳永逸的。因为数据是不断地增长的，随着时间不断地推移，已经分好的区域，或许已经装不住更多的数据，当然就要进一步进行split了，同样也会出现性能损耗问题，所以我们还是要规划好数据增长速率，观察好数据定期维护，按需分析是否要进一步分行手工将分区再分好，也或者是更严重的是新建表，做好更大的预分区然后进行数据迁移。小吴只是菜鸟，运维方面也只是自己这样认为而已，供大家作简单的参考吧。如果数据装不住了，对于partition方式预分区的话，如果让它自然分裂的话，情况分严重一点。因为分裂出来的分区号会是一样的，所以计算到partitionId的话，其实还是回到了顺序写年代，会有部分热点写问题出现，如果使用partition方式生成主键的话，数据增长后就要不断地调整分区了，比如增多预分区，或者加入子分区号的处理.(我们的分区号为long型，可以将它作为多级partition)

OK,写到这里，基本已经讲完了防止热点写使用的方法和防止频繁split而采取的预分区。但rowkey设计，远远也不止这些，比如rowkey长度，然后它的长度最大可以为char的MAXVALUE,但是看过之前我写KeyValue的分析知道，我们的数据都是以KeyValue方式存储在MemStore或者HFile中的，每个KeyValue都会存储rowKey的信息，如果rowkey太大的话，比如是128个字节，一行10个字段的表，100万行记录，光rowkey就占了1.2G+所以长度还是不要过长，另外设计，还是按需求来吧。

最后题外话是我想分享我在github中建了一个project,希望做一些hbase一些工具：https://github.com/bdifn/hbase-tools,如果本地装了git的话，可以执行命令: git clone https://github.com/bdifn/hbase-tools.git目前加了一个region-helper子项目，也是目前唯一的一个子项目，项目使用maven管理,主要目的是帮助我们设计rowkey做一些参考，比如我们设计的随机写和预分区测试，提供了抽样的功能，提供了检测随机写的功能，然后统计按目前rowkey设计，随机写n条记录后，统计每个region的记录数，然后显示比例等。
测试仿真模块我程为simualtor,主要是模拟hbase的region行为，simple的实现，仅仅是上面提到的预测我们rowkey设计后，建好预分区后，写数据的的分布比例，而emulation是比较逼真的仿真，设想是我们写数据时，会统计数目的大小，根据我们的hbase-site.xml设定，模拟memStore行为，模拟hfile的行为，最终会生成一份表的报表，比如分区的数据大小，是否split了，等等，以供我们去设计hbase表时有一个参考，但是遗憾的是，由于时间关系，我只花了一点业余时间简单搭了一下框架，目前没有更一步的实现，以后有时间再加以完善，当然也欢迎大家一起加入，一起学习吧。



项目使用maven管理，为了方便测试，一些组件的实例化，我使用了java的SPI,download源码后，如果想测试自己的rowKeyGeneator的话，打开com.bdifn.hbasetools.regionhelper.rowkey.RowKeyGenerator文件后，替换到你们的ID生成器就可以了。如果是hash的话，抽样和测试等，都是可以复用的。

如测试代码：

```
public class HBaseSimulatorTest {
    //通过SPI方式获取HBaseSimulator实例,SPI的实现为simgple
    private  HBaseSimulator hbase = BeanFactory.getInstance().getBeanInstance(HBaseSimulator.class);
    //获取RowKeyGenerator实例，SPI的实现为hashRowkey
    private RowKeyGenerator rkGen = BeanFactory.getInstance().getBeanInstance(RowKeyGenerator.class);
    //初如化苦工，去检测100w个抽样rowkey,然后生成一组splitKeys
    HashChoreWoker worker = new HashChoreWoker(1000000,10);

    @Test
    public void testHash(){
        byte [][] splitKeys = worker.calcSplitKeys();
        hbase.createTable("user", splitKeys);
        //插入1亿条记录，看数据分布
        TableName tableName = TableName.valueOf("user");
        for(int i = 0; i < 100000000; i ++) {
```

```
            Put put = new Put(rkGen.nextId());
            hbase.put(tableName, put);
        }
        hbase.report(tableName);
    }

    @Test
    public void testPartition(){
        //default 20 partitions.
        PartitionRowKeyManager rkManager = new PartitionRowKeyManager();
        byte [][] splitKeys = rkManager.calcSplitKeys();

        hbase.createTable("person", splitKeys);

        TableName tableName = TableName.valueOf("person");
        //插入1亿条记录，看数据分布
        for(int i = 0; i < 100000000; i ++) {
            Put put = new Put(rkManager.nextId());
            hbase.put(tableName, put);
        }

        hbase.report(tableName);
    }
}
```

执行结果:

```
Execution Reprort:[StartRowkey:puts requsts:(put ratio)]
:9973569:(1.0015434)
1986344a\x00\x00\x00\x00\x00\x01\x0E\xAE:9999295:(1.0041268)
331ee65f\x00\x00\x00\x00\x00\x0F)g:10012532:(1.005456)
4cbfd4f6\x00\x00\x00\x00\x00\x00o0:9975842:(1.0017716)
664c6388\x00\x00\x00\x00\x00\x02\x1Du:10053337:(1.0095537)
800945e0\x00\x00\x00\x00\x00\x01\xADV:9998719:(1.0040689)
99a158d9\x00\x00\x00\x00\x00\x0BZ\xF3:10000563:(1.0042541)
b33a2223\x00\x00\x00\x00\x00\x07\xC6\xE6:9964921:(1.000675)
ccbcf370\x00\x00\x00\x00\x00\x00*\xE2:9958200:(1.0)
e63b8334\x00\x00\x00\x00\x00\x03g\xC1:10063022:(1.0105262)
total requests:100000000
Execution Reprort:[StartRowkey:puts requsts:(put ratio)]
:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x01:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x02:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x03:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x04:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x05:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x06:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x07:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x08:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x09:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x0A:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x0B:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x0C:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x0D:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x0E:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x0F:5000000:(1.0)
```

```
\x00\x00\x00\x00\x00\x00\x00\x10:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x11:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x12:5000000:(1.0)
\x00\x00\x00\x00\x00\x00\x00\x13:5000000:(1.0)
total requests:100000000
```

原贴地址：http://www.cnblogs.com/bdifn/p/3801737.html ,转载请注明

标签: Hbase

好文要顶 关注我 收藏该文

小吴蜀黍
关注 - 1
粉丝 - 13
+加关注

6 0

« 上一篇：HBase之HFile解析

posted on 2014-06-22 10:34 小吴蜀黍 阅读(10717) 评论(5) 编辑 收藏

评论

#1楼 2015-04-01 21:12 xiangshit

膜拜楼主，能不能提供份源码呀，2508145541@qq.com，谢谢

支持(0) 反对(0)

#2楼 2016-09-18 15:11 g-fantastic

您好，想请教您一个问题：rowkey散列处理后，如何按照rowkey来查询呢？
假如有一个用户id的字段，如果直接把id作为rowkey存到hbase中，那么我要查询某个用户id的数据，直接可以将用户id作为rowkey来查询即可。但是，将用户id散列处理后，rowkey就成了随机的值，无法知道哪个用户id对应哪个rowkey，那应该如何查询呢？当然，有一种方式是将id和散列后rowkey的对应关系保存起来，但如果用户id数据很大的话，那就没办法了。

支持(1) 反对(0)

#3楼 2016-10-08 11:39 summer-R

二楼的问题我也很想知道答案

支持(0) 反对(0)

#4楼 2016-11-17 09:46 八进制

@ g-fantastic
散列以后的rowkey不会是随机值，因为散列函数是固定的。查询时用同样的散列函数处理用户id，再用处理后的结果查询即可。

支持(2) 反对(0)

#5楼 2017-03-26 13:43 张春烽

@ 八进制
正解

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。