1. HBase高级应用

1.1. 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（布隆过滤器）

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',{NAME => 'baseinfo' BLOOMFILTER => 'ROW'}

作用：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（版本号）

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',{ NAME => 'baseinfo' VERSIONS=>'2'}
```

附：MIN_VERSIONS => '0'是说在compact操作执行之后，至少要保留的版本，只有在设置了TTL的时候生效

3、COMPRESSION（压缩）

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，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

标准：压缩率 压缩速率

4、TTL（Time To Live）

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。

5、Alter（修改表）

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。Hbase没有修改的操作需求。建表的时候一般就只是制定了 表名 和列簇的定义。 列是插入数据的时候指定的。

6、describe/desc（查看表详细信息）

这个命令查看了create table 的各项参数或者是默认值。

使用方式：

```
describe 'user_info'
```

7、disable_all/enable_all

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:y
```

8、drop_all

这个命令和disable_all的使用方式是一样的， 以上三个操作都支持正则

9、HBase预分区

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 table with specific split points
hbase>create 'table1','f1',SPLITS => ['\x10\x00', '\x20\x00', '\x30\x00',
'\x40\x00']

# create table with four regions based on random bytes keys
hbase>create 'table2','f1', { NUMREGIONS => 8 , SPLITALGO => 'UniformSplit' }

# create table with five regions based on hex keys
hbase>create 'table3','f1', { NUMREGIONS => 10, SPLITALGO => 'HexStringSplit' }
```

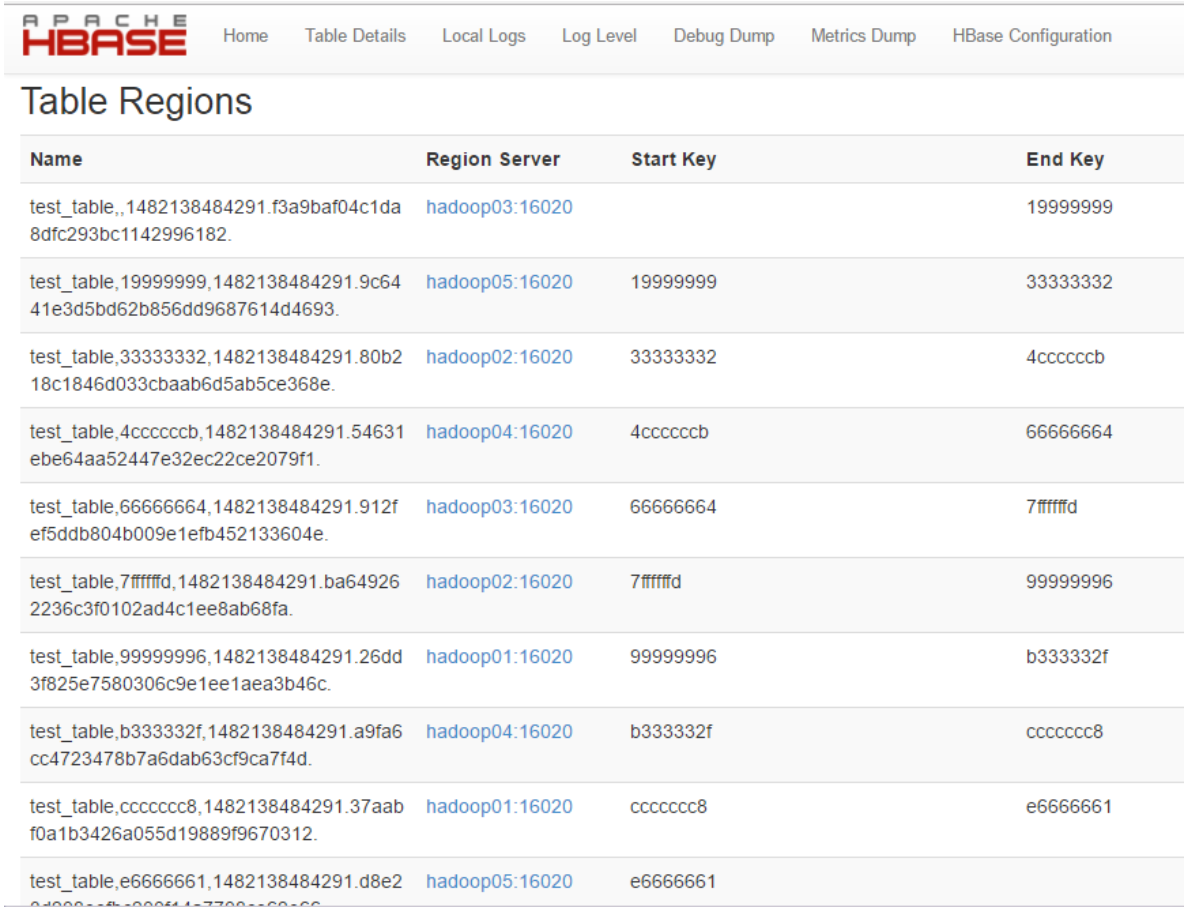
也可以使用api的方式:

```
hbase org.apache.hadoop.hbase.util.RegionSplitter test_table HexStringSplit -c
10 -f info
hbase org.apache.hadoop.hbase.util.RegionSplitter splitTable HexStringSplit -c
10 -f info
```

参数：

```
test_table          表名
HexStringSplit      split 方式
-c                  指定初始region数量
-f                  指定family信息
```

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。

1.2. 表设计

1.2.1. 列簇设计

追求的原则是：在合理范围内能尽量少的减少列簇就尽量减少列簇。

最优设计是：将所有相关性很强的key-value都放在同一个列簇下，这样既能做到查询效率最高，也能保持尽可能少的访问不同的磁盘文件

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

hbase的列簇越少越好！尽量就是1个

1.2.2. RowKey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

那怎么快速精准的定位到我们想要操作的数据，就在于我们的 rowkey 的设计了

Rowkey设计三原则

一、rowkey长度原则

Cell对象中，真是情况是存储了一个key-value 但是这个cell也存储了rowkey

> Rowkey是一个二进制码流，Rowkey的长度被很多开发者建议说设计在10~100个字节，不过建议是越短越好，不要超过16个字节。
>
> 原因如下：
>
> 1、数据的持久化文件HFile中是按照KeyValue存储的，如果Rowkey过长比如100个字节，1000万列数据光Rowkey就要占用100*1000万=10亿个字节，将近1G数据，这会极大影响HFile的存储效率；
>
> 2、MemStore将缓存部分数据到内存，如果Rowkey字段过长内存的有效利用率会降低，系统将无法缓存更多的数据，这会降低检索效率。因此Rowkey的字节长度越短越好。
>
> 3、目前操作系统是都是64位系统，内存8字节对齐。控制在16个字节，8字节的整数倍利用操作系统的最佳特性。

二、rowkey散列原则

> 如果Rowkey是按时间戳的方式递增，不要将时间放在二进制码的前面，建议将Rowkey的高位作为散列字段，由程序循环生成，低位放时间字段，这样将提高数据均衡分布在每个Regionserver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将产生所有新数据都在一个 RegionServer上堆积的热点现象，这样在做数据检索的时候负载将会集中在个别RegionServer，降低查询效率。

三、rowkey唯一原则

> 必须在设计上保证其唯一性。rowkey是按照字典顺序排序存储的，因此，设计rowkey的时候，要充分利用这个排序的特点，将经常读取的数据存储到一块，将最近可能会被访问的数据放到一块。

1.2.3. 数据热点

HBase中的行是按照rowkey的字典顺序排序的，这种设计优化了scan操作，可以将相关的行以及会被一起读取的行存取在临近位置，便于scan。然而糟糕的rowkey设计是热点的源头。 热点发生在大量的client直接访问集群的一个或极少数个节点（访问可能是读，写或者其他操作）。大量访问会使热点region所在的单个机器超出自身承受能力，引起性能下降甚至region不可用，这也会影响同一个RegionServer上的其他region，由于主机无法服务其他region的请求。 设计良好的数据访问模式以使集群被充分，均衡的利用。

为了避免写热点，设计rowkey使得不同行在同一个region，但是在更多数据情况下，数据应该被写入集群的多个region，而不是一个。

防止数据热点的有效措施：

1、加盐

这里所说的加盐不是密码学中的加盐，而是在rowkey的前面增加随机数，具体就是给rowkey分配一个随机前缀以使得它和之前的rowkey的开头不同。分配的前缀种类数量应该和你想使用数据分散到不同的region的数量一致。加盐之后的rowkey就会根据随机生成的前缀分散到各个region上，以避免热点。

加随机前缀

2、哈希

哈希会使同一行永远用一个前缀加盐。哈希也可以使负载分散到整个集群，但是读却是可以预测的。使用确定的哈希可以让客户端重构完整的rowkey，可以使用get操作准确获取某一个行数据

两个非常类似的字符串经过hash只有得道的值会完全不一样。

3、反转

第三种防止热点的方法是反转固定长度或者数字格式的rowkey。这样可以使得rowkey中经常改变的部分（最没有意义的部分）放在前面。这样可以有效的随机rowkey，但是牺牲了rowkey的有序性。

反转rowkey的例子以手机号为rowkey，可以将手机号反转后的字符串作为rowkey，这样的就避免了以手机号那样比较固定开头导致热点问题。

按照日期，可能比较集中。 20200620　20062020

4、时间戳反转

一个常见的数据处理问题是快速获取数据的最近版本，使用反转的时间戳作为rowkey的一部分对这个问题十分有用，可以用 Long.Max_Value - timestamp 追加到key的末尾，例如 [key] [reverse_timestamp] , [key] 的最新值可以通过scan [key]获得[key]的第一条记录，因为HBase中rowkey是有序的，第一条记录是最后录入的数据。比如需要保存一个用户的操作记录，按照操作时间倒序排序，在设计rowkey的时候，可以这样设计
[userId反转] [Long.Max_Value - timestamp]，在查询用户的所有操作记录数据的时候，直接指定反转后的userId，startRow是[userId反转] [000000000000],stopRow是[userId反转] [Long.Max_Value - timestamp]
如果需要查询某段时间的操作记录，startRow是[user反转] [Long.Max_Value - 起始时间]，stopRow是[userId反转] [Long.Max_Value - 结束时间] [Long.Max_Value - 结束时间]

电话号码，身份证，纯数字的类型，都可以使用这个方式

总结：

数据散列不能太散列，数据集中不能太集中。结合业务（站在什么角度），进行取舍。

hbase的数据访问有三种方式：

1、单个 rowkey
    非常频繁的一些 rokwey 尽量分散
2、范围 查询
    如果要查询的数据量很大，但是数据很分散，那么就需要从很多regoinserver去查询
    如果要查询的数据量很大，但是又都集中在一个节点，那么这个节点的压力就比较大。
3、全表扫描