目录

设计原则

Schema创建

可以使用HBase Shell或者Java API的HBaseAdmin来创建和编辑HBase的Schema。

更新

当表或者列族改变时（包括：编码方式、压力格式、block大小等等），都将会在下次marjor compaction时或者StoreFile重写时生效。

表模式设计经验

- region规模大小在10到50g之间。
- 单个cell不超过10MB，如果超过10MB，请使用mob。
- 一个表大约 50到100个 region，1或者2个列族。记住：每个列族内是连续的，不同列族之间是分割的。
- 尽量让你的列族名短，因为存储时每个value都包含列族名（忽略前缀编码， prefix encoding ）。
- 如果在基于时间的机器上存储数据这和日志信息，row key是由设备ID加上时间得到的，那最后得到这样的模式：除了某个特定的时间段，旧的数据region没有额外的写。这样的情况下，得到的是少量的活跃region和大量没有新写入的旧region。这时由于资源消耗仅仅来自活跃的region，大量的Region能被接受。

列族的数量

现在HBase并不能很好的处理两个或者三个以上的列族，所以尽量让列族数量少一些。
目前， flush 和 compaction 操作是针对一个region。所以当一个列族操作大量数据的时候会引发一个flush。那些邻近的列族也有进行flush操作，尽管它们没有操作多少数据。
compaction操作现在是根据一个列族下的全部文件的数量触发的，而不是根据文件大小触发的。

列族的基数

如果一个表存在多个列族，要注意列族之间基数(如行数)相差不要太大。 例如列族A有100万行，列族B有10亿行，按照行键切分后，列族A可能被分散到很多region(及RegionServer)，这导致扫描列族A十分低效。

版本的数量

行的版本的数量是HColumnDescriptor设置的，每个列族可以单独设置，默认是3。这个设置是很重要的，因为HBase是不会去覆盖一个值的，它只会在后面在追加写，用时间戳来区分、过早的版本会在执行major compaction时删除，这些在HBase数据模型有描述。这个版本的值可以根据具体的应用增加减少。
不推荐将版本最大值设到一个很高的水平 (如成百或更多)，除非老数据对你很重要。因为这会导致存储文件变得极大。

最小版本数

和行的最大版本数一样，最小版本数也是通过HColumnDescriptor在每个列族中设置的。最小版本数缺省值是0，表示该特性禁用。 最小版本数参数和存活时间一起使用，允许配置如“保存最后T秒有价值数据，最多N个版本，但最少约M个版本”(M是最小版本数，M<N)。 该参数仅在存活时间对列族启用，且必须小于行版本数。

支持数据类型

HBase通过Put和Result支持 bytes-in/bytes-out 接口，所以任何可被转为字节数组的东西可以作为值存入。输入可以是字符串、数字、复杂对象、甚至图像，它们能转为字节。
存在值的实际长度限制 (如：保存10-50MB对象到HBase可能对查询来说太长)；搜索邮件列表获取本话题的对话。HBase的所有行都遵循HBase数据模型包括版本化。设计时需考虑到这些，以及列族的块大小。

数据过期时间TTL

列族可以设置TTL秒数，HBase在超时后将自动删除数据，HBase里面TTL时间时区是UTC。过期后的数据会无法读取，但过期后数据真正删除发生在major compaction时。

Rowkey设计原则

HBase的rowkey设计可以说是使用HBase最为重要的事情，直接影响到HBase的性能，常见的RowKey的设计问题及对应访问为：

Hotspotting

Base的行由行键按字典顺序排序，这样的设计优化了扫描，允许存储相关的行或者那些将被一起读的邻近的行。然而，设计不好的行键是导致 hotspotting 的常见原因。当大量的客户端流量（traffic）被定向在集群上的一个或几个节点时，就会发生 hotspotting。这些流量可能代表着读、写或其他操作。流量超过了承载该region的单个机器所能负荷的量，这就会导致性能下降并有可能造成region的不可用。在同一 RegionServer上的其他region也可能会受到其不良影响，因为主机无法提供服务所请求的负载。设计使集群能被充分均匀地使用的数据访问模式是至关重要的。
为了防止在写操作时出现 hotspotting ，设计行键时应该使得数据尽量同时往多个region上写，而避免只向一个region写，除非那些行真的有必要写在一个region里。
下面介绍了集中常用的避免 hotspotting 的技巧，它们各有优劣：

Salting

Salting 从某种程度上看与加密无关，它指的是将随机数放在行键的起始处。进一步说，salting给每一行键随机指定了一个前缀来让它与其他行键有着不同的排序。所有可能前缀的数量对应于要分散数据的region的数量。如果有几个“hot”的行键模式，而这些模式在其他更均匀分布的行里反复出现，salting就能到帮助。下面的例子说明了salting能在多个RegionServer间分散负载，同时也说明了它在读操作时候的负面影响。
假设行键的列表如下，表按照每个字母对应一个region来分割。前缀‘a’是一个region，‘b’就是另一个region。在这张表中，所有以‘f’开头的行都属于同一个region。这个例子关注的行和键如下：

```
foo0001
foo0002
foo0003
foo0004
```

现在，假设想将它们分散到不同的region上，就需要用到四种不同的 salts ：a，b，c，d。在这种情况下，每种字母前缀都对应着不同的一个region。用上这些salts后，便有了下面这样的行键。由于现在想把它们分到四个独立的区域，理论上吞吐量会是之前写到同一region的情况的吞吐量的四倍。

```
a-foo0003
b-foo0001
c-foo0004
d-foo0002
```

如果想新增一行，新增的一行会被随机指定四个可能的salt值中的一个，并放在某条已存在的行的旁边。

```
a-foo0003
b-foo0001
c-foo0003
c-foo0004
d-foo0002
```

由于前缀的指派是随机的，因而如果想要按照字典顺序找到这些行，则需要做更多的工作。从这个角度上看，salting增加了写操作的吞吐量，却也增大了读操作的开销。

Hashing

可用一个单向的 hash 散列来取代随机指派前缀。这样能使一个给定的行在“salted”时有相同的前缀，从某种程度上说，这在分散了RegionServer间的负载的同时，也允许在读操作时能够预测。确定性hash（deterministic hash）能让客户端重建完整的行键，以及像正常的一样用Get操作重新获得想要的行。
考虑和上述salting一样的情景，现在可以用单向hash来得到行键foo0003，并可预测得‘a’这个前缀。然后为了重新获得这一行，需要先知道它的键。可以进一步优化这一方法，如使得将特定的键对总是在相同的region。

Reversing the Key(反转键)

第三种预防hotspotting的方法是反转一段固定长度或者可数的键，来让最常改变的部分（最低显著位，the least significant digit）在第一位，这样有效地打乱了行键，但是却牺牲了行排序的属性。

单调递增行键/时序数据

在一个集群中，一个导入数据的进程锁住不动，所有的client都在等待一个region(因而也就是一个单个节点)，过了一会后，变成了下一个region。 如果使用了单调递增或者时序的key便会造成这样的问题。使用了顺序的key会将本没有顺序的数据变得有顺序，把负载压在一台机器上。所以要尽量避免时间戳或者序列(例如 1，2，3)这样的行键。
如果需要导入时间顺序的文件(例如log)到HBase中，可以学习OpenTSDB的做法。它有一个页面来描述它的HBase模式。OpenTSDB的Key的格式是[metric_type][event_timestamp]，乍一看，这似乎违背了不能将timestamp做key的建议，但是它并没有将timestamp作为key的一个关键位置，有成百上千的metric_type就足够将压力分散到各个region了。因此，尽管有着连续的数据输入流，Put操作依旧能被分散在表中的各个region中。

简化行和列

在HBase中，值是作为一个单元（Cell）保存在系统中的，要定位一个单元，需要行、列名和时间戳。通常情况下，如果行和列的名字要是太大（甚至比value的大小还要大）的话，可能会遇到一些难以预估的情况。在HBase的存储文件

（storefiles）中，有一个索引用来方便值的随机访问，但是访问一个单元的坐标要是太大的话，会占用很大的内存，这个索引会被用尽。要想解决这个问题，可以设置一个更大的块大小，也可以使用更小的行和列名，压缩也能得到更大指数。
大部分时候，细微的低效不会影响很大。但不幸的是，在这里却不能忽略。无论是列族、属性和行键都会在数据中重复上亿次。

列族

尽量使列族名称小，最好一个字符。（如：f表示）

行键长度

让行键短到可读即可，这样对获取数据有帮助(例如 Get vs. Scan)。短键对访问数据无用，并不比长键对get/scan更好。设计行键需要权衡。

倒序时间戳

一个数据库处理的通常问题是找到最近版本的值。采用倒序时间戳作为键的一部分可以对此特定情况有很大帮助。该技术包含追加（Long.MAX_VALUE - timestamp）到key的后面，如 [key][reverse_timestamp] 。 表内[key]的最近的值可以用[key]进行Scan，找到并获取第一个记录。由于HBase行键是排序的，该键排在任何比它老的行键的前面，所以是第一个。 该技术可以用于代替版本数，其目的是保存所有版本到“永远”(或一段很长时间) 。同时，采用同样的Scan技术，可以很快获取其他版本。

行键和列族

行键在列族范围内。所以同样的行键可以在同一个表的每个列族中存在而不会冲突。

行键不可改

行键不能改变。唯一可以**改变**的方式是删除然后再插入。这是一个常问问题，所以要注意开始就要让行键正确(且/或在插入很多数据之前)。

行键和region split的关系

如果已经 pre-split（预分区）了表，接下来关键要了解行键是如何在region边界分布的。为了说明为什么这很重要，可考虑用可显示的16位字符作为键的关键位置（例如 “0000000000000000” to “ffffffffffffffff”）这个例子。通过Bytes.split来分割键的范围（这是当用 Admin.createTable(byte[] startKey, byte[] endKey, numRegions) 创建region时的一种拆分手段），这样会分得10个region。

```
1.48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48                  // 0
2.54 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10   // 6
3.61 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -68  // =
4.68 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -126  // D
5.75 75 75 75 75 75 75 75 75 75 75 75 75 75 72                   // K
6.82 18 18 18 18 18 18 18 18 18 18 18 18 18 14                   // R
7.88 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -44   // X
8.95 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -102  // _
9.102 102 102 102 102 102 102 102 102 102 102 102 102 102 102 102   // f
```

但问题在于，数据将会堆放在前两个region以及最后一个region，这样就会导致某几个region由于数据分布不均匀而特别忙。为了理解其中缘由，需要考虑ASCII Table的结构。根据ASCII表，“0”是第48号，“f”是102号；但58到96号是个巨大的间隙，考虑到在这里仅[0-9]和[a-f]这些值是有意义的，因而这个区间里的值不会出现在键空间（keyspace），进而中间区域的region将永远不会用到。为了pre-split这个例子中的键空间，需要自定义拆分。
预分区表（pre-splitting tables）是个很好的实践，但pre-split时要注意使得所有的region都能在键空间中找到对应。
尽管例子中解决的问题是关于16位键的键空间，但其他任何空间也是同样的道理。16位键（通常用到可显示的数据中）尽管通常不可取，但只要所有的region都能在键空间找到对应，它依旧能和预裂表配合使用。
以下case说明了如何16位键预分区。

```
1.public static boolean createTable(Admin admin, HTableDescriptor table, byte[][] splits)
2.throws IOException {
3.  try {
4.    admin.createTable( table, splits );
5.    return true;
6.  } catch (TableExistsException e) {
7.    logger.info("table " + table.getNameAsString() + " already exists");
8.    // the table already exists...
9.    return false;
10.  }
11.}
12.
13.public static byte[][] getHexSplits(String startKey, String endKey, int numRegions) {
14.  byte[][] splits = new byte[numRegions-1][];
15.  BigInteger lowestKey = new BigInteger(startKey, 16);
16.  BigInteger highestKey = new BigInteger(endKey, 16);
```

```
17.  BigInteger range = highestKey.subtract(lowestKey);
18.  BigInteger regionIncrement = range.divide(BigInteger.valueOf(numRegions));
19.  lowestKey = lowestKey.add(regionIncrement);
20.  for(int i=0; i < numRegions-1;i++) {
21.    BigInteger key = lowestKey.add(regionIncrement.multiply(BigInteger.valueOf(i)));
22.    byte[] b = String.format("%016x", key).getBytes();
23.    splits[i] = b;
24.  }
25.  return splits;
26.}
```