云数据库 HBase

开发指南

开发指南

HBase API

HBase API不同的版本有一些细微的差异，1.x系列的主要API基本未变。详细请参考：Apache hbase shell

数据定义

创建表

```
语法：create '<table>', '<column_family>' [, '<column_family>', ...]
例子：新建一张包括f1、f2列族的表名威test的表：create 'test', 'f1', 'f2'
```

- 检测表模式

```
hbase(main):002:0> list
TABLE
users
1 row(s) in 0.0180 seconds

=> ["users"]
hbase(main):003:0> desc 'users'
Table users is ENABLED
users
COLUMN FAMILIES DESCRIPTION
{NAME => 'info', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false', KEEP_DELETED_CELLS =>
'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL
=> 'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536',
REPLICATION_SCOPE => '0'}
1 row(s) in 0.0750 seconds
```

HBase最佳实践

数据压缩与编码

压缩算法

目前阿里云平台支持压缩算法有： LZO ZSTD GZ LZ4 SNAPPY NONE，其中NONE就代表不开启压缩

不同压缩算法在不同场景的压缩比，及解压速度对比如下，都是来自线上真实场景：

业务类型	无压缩表大小	LZO（压缩率/解压速度MB/s）	ZSTD（压缩率/解压速度MB/s）	LZ4（压缩率/解压速度MB/s）
监控类	419.75T	5.82/372	13.09/256	5.19/463.8
日志类	77.26T	4.11/333	6.0/287	4.16/496.1
风控类	147.83T	4.29/297.7	5.93/270	4.19/441.38
消费记录	108.04T	5.93/316.8	10.51/288.3	5.55/520.3

我们建议在：

- 对rt要求极高，建议使用 lz4 压缩算法
- 对rt要求不高，特别是 监控、物联网等场景，建议使用 zstd 压缩算法

编码

hbase很早就支持了DataBlockEncoding，也就是是通过减少hbase keyvalue中重复的部分来压缩数据。我们推荐DATA_BLOCK_ENCODING使用diff

操作

修改压缩编码的步骤：

1、修改表的属性，此为压缩编码
alter 'test', {NAME => 'f', COMPRESSION => 'lz4', DATA_BLOCK_ENCODING =>'DIFF'}
2、压缩编码并不会立即生效，需要major_compact，此会耗时较长，注意在业务低峰期进行
major_compact 'test'

具体压缩编码的细节参考：阿里HBase数据压缩编码探索

read读取优化

其实HBase还是比较灵活的，关键看你是否使用得当，以下主要列举一些读的优化。HBase在生产中往往会遇到Full GC、进程OOM、RIT问题、读取延迟较大等一些问题，使用更好的硬件往往可以解决一部分问题，但是还是需要使用的方式。

我们把优化分为：

客户端优化、服务端优化、平台优化(ApsaraDB for HBase平台完成)

客户端优化

get请求是否可以使用批量请求

这样可以成倍减小客户端与服务端的rpc次数，显著提高吞吐量

```
Result[] re= table.get(List<Get> gets)
```

大scan缓存是否设置合理

scan一次性需求从服务端返回大量的数据，客户端发起一次请求，客户端会分多批次返回客户端，这样的设计是避免一次性传输较多的数据给服务端及客户端有较大的压力。目前 数据会加载到本地的缓存中，默认100条数据大小。 一些大scan需要获取大量的数据，传输数百次甚至数万的rpc请求。 我们建议可以 适当放开 缓存的大小。

```
scan.setCaching(int caching) //大scan可以设置为1000
```

请求指定列族或者列名

HBase是列族数据库，同一个列族的数据存储在一块，不同列族是分开的，为了减小IO，建议指定列族或者列名

离线计算访问Hbase建议禁止缓存

当离线访问HBase时，往往就是一次性的读取，此时读取的数据没有必要存放在blockcache中，建议在读取时禁止缓存

```
scan.setBlockCache(false)
```

可干预服务端优化

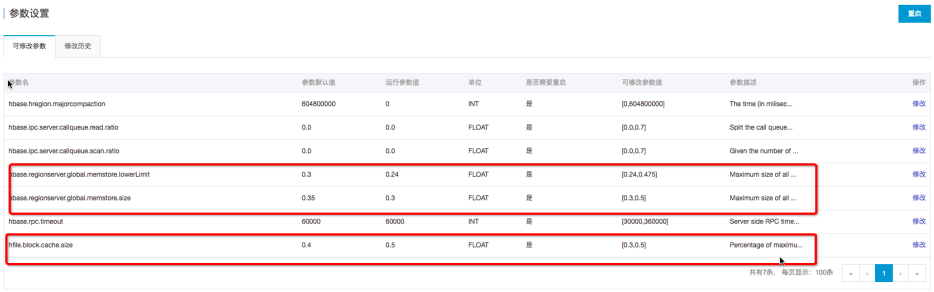
请求是否均衡

读取的压力是否都在一台或者几台之中，在业务高峰期间可以查看下，可以到 HBase管控平台查看HBase的ui。如果有明显的热点，一劳永逸是重新设计rowkey，短期是 把热点region尝试拆分

BlockCache是否合理

BlockCache作为读缓存，对于读的性能比较重要，如果读比较多，建议内存使用1:4的机器，比如：8cpu32g或者16pu64g的机器。当前可以调高 BlockCache 的数值，降低 Memstore 的数值。

在ApsaraDB for HBase控制台可以完成：hfile.block.cache.size 改为0.5 ；hbase.regionserver.global.memstore.size 改为0.3；再重启



HFile文件数目

因为读取需要频繁打开HFile，如果文件越多，IO次数就越多，读取的延迟就越高此 需要主要定时做 major compaction，如果晚上的业务压力不大，可以在晚上做major compaction

Compaction是否消耗较多的系统资源

compaction主要是将HFile的小文件合并成大文件，提高后续业务的读性能，但是也会带来较大的资源消耗。Minor Compaction一般情况下不会带来大量的系统资源消耗，除非因为配置不合理。 切勿在高峰期间做major compaction。 建议在业务低峰期做major compaction。

Bloomfilter设置是否合理

Bloomfilter主要用来在查询时，过滤HFile的，避免不需要的IO操作。Bloomfilter能提高读取的性能，一般情况下创建表，都会默认设置为：BLOOMFILTER => ‘ROW’

平台端优化

数据本地率是否太低?（平台已经优化）

Hbase 的HFile，在本地是否有文件，如果有文件可以走Short-Circuit Local Read目前平台在重启、磁盘扩容时，都会自动拉回移动出去的region，不降低数据本地率；另外 定期做major compaction也有益于提高本地化率

Short-Circuit Local Read （已经默认开启）

当前HDFS读取数据需要经过DataNode，开启Short-Circuit Local Read后，客户端可以直接读取本地数据

Hedged Read （已经默认开启）

优先会通过Short-Circuit Local Read功能尝试本地读。但是在某些特殊情况下，有可能会出现因为磁盘问题或者网络问题引起的短时间本地读取失败，为了应对这类问题，开发出了Hedged Read。该机制基本工作原理为：客户端发起一个本地读，一旦一段时间之后还没有返回，客户端将会向其他DataNode发送相同数据的请求。哪一个请求先返回，另一个就会被丢弃

关闭swap区（已经默认关闭）

swap是当物理内存不足时，拿出部分的硬盘空间当做swap使用，解决内存不足的情况。但是会有较大的延迟的问题，所以我们HBase平台默认关闭。 但是关闭swap导致anon-rss很高，page reclaim没办法reclaim足够的page，可能导致内核挂住，平台已经采取相关隔离措施避免此情况。

write写入优化

HBase基于LSM模式，写是写HLOG及Memory的，也就是基本没有随机的IO，所以在写链路上性能高校还比较平稳。很多时候，写都是用可靠性来换取性能。

客户端优化

批量写

也是为了减少rpc的次数

```
HTable.put(List<Put>)
```

Auto Flush

autoflush=false可以提升几倍的写性能，但是还是要注意，直到数据超过2M(hbase.client.write.buffer决定)或用户执行了hbase.flushcommits()时才向regionserver提交请求。需要注意并不是写到了远端。

HTable.setWriteBufferSize(writeBufferSize) 可以设置buffer的大小

服务端优化

WAL Flag

不写WAL可以成倍提升性能，因为不需要写HLog，减少3次IO，写MemStore是内存操作

是以数据可靠性为代价的，在数据导入时，可以关闭WAL

增大memstore的内存

当前可以调高Memstore 的数值，降低 BlockCache 的数，跟读优化的思路正好相反

大量的HFile产生

如果写很快，很容易带来大量的HFile，因为此时HFile合并的速度还没有写入的速度快

需要在业务低峰期做majorcompaction，充分利用系统资源；如果HFile降低不下来，则需要添加节点

读写分离

HBase有三个典型的API：read(get、scan)、write ，我们有时候希望这三个访问尽可能的互相不影响，可以参考如下配置：（线上默认没有配置读写分离）

场景

- 写请求与读请求都比较高，业务往往接受：写请求慢点可以，读请求越快越好，最好有单独的资源保障
- scan与get都比较多，业务希望scan不影响get（因为scan比较消耗资源）

相关配置：

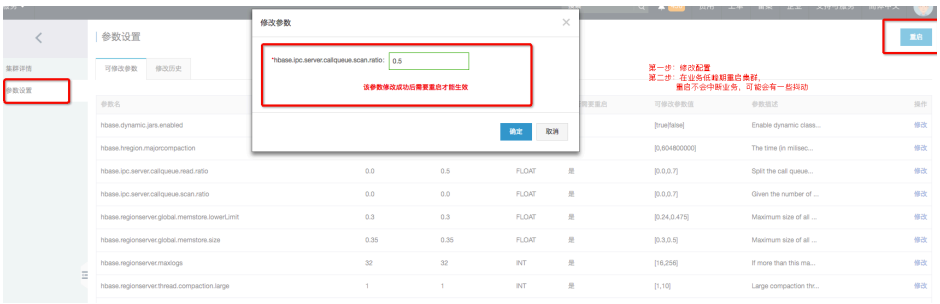
- hbase.ipc.server.callqueue.read.ratio
- hbase.ipc.server.callqueue.scan.ratio

具体含义：

- hbase.ipc.server.callqueue.read.ratio 设置为0.5，代表有50%的线程数处理读请求
- 如果再设置hbase.ipc.server.callqueue.scan.ratio 设置为0.5，则代表在50%的读线程之中，再有50%的线程处理scan，也就是全部线程的25%

操作步骤

- 打开HBase控制台，找到实例，点击进去，找到 - 参数设置
- 修改配置，按照业务读写情况
- 不重启不会生效，请在业务低峰期重启集群，重启不会中断业务，可能会有一些抖动



请根据实际的业务配置以上数值，默认情况下是没有配置的，也就是读写都共享。

预分区

初次接触HBase的客户，在创建HBase表的时候，不指分区的数目，另外就是rowkey设计不合理，导致热点。

最为常见的建表语句为：

create ‘t3’,{NAME => ‘f1’,COMPRESSION => ‘snappy’}, { NUMREGIONS => 50, SPLITALGO => ‘HexStringSplit’ }

- 其中 NUMREGIONS 为 region的个数，一般按每个region 6~8GB左右来计算region数量，集群规模大，region数量可以适当取大一些

  SPLITALGO 为 rowkey分割的算法：Hbase自带了三种pre-split的算法，分别是 HexStringSplit、DecimalStringSplit 和 UniformSplit。

  各种Split算法适用场景：

  - HexStringSplit: rowkey是十六进制的字符串作为前缀的
  - DecimalStringSplit: rowkey是10进制数字字符串作为前缀的
  - UniformSplit: rowkey前缀完全随机

  关于rowkey的设计可以参考：RowKey设计

- COMPRESSION压缩算法，参考：数据压缩与编码

HBase表设计

Rowkey设计

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

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(e.g. 1, 2, 3)这样的行键。

如果需要导入时间顺序的文件(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)保存在系统的中的，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，可能会遇到一些有趣的情况。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

尽量使列族名小，最好一个字符。（如：f 表示）

属性

详细属性名 (如:”myVeryImportantAttribute”）易读，最好还是用短属性名 (e.g., “via”）保存到HBase.

行键长度

让行键短到可读即可，这样对获取数据有帮助(e.g., 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

字节模式

long类型有8字节。8字节内可以保存无符号数字到18,446,744,073,709,551,615。 如果用字符串保存——假设一个字节一个字符——需要将近3倍的字节数。

下面是示例代码，可以自己运行一下:

```
// long
//
long l = 1234567890L;
byte[] lb = Bytes.toBytes(l);
System.out.println("long bytes length: " + lb.length); // returns 8

String s = String.valueOf(l);
byte[] sb = Bytes.toBytes(s);
System.out.println("long as string length: " + sb.length); // returns 10

// hash
//
MessageDigest md = MessageDigest.getInstance("MD5");
byte[] digest = md.digest(Bytes.toBytes(s));
System.out.println("md5 digest bytes length: " + digest.length); // returns 16

String sDigest = new String(digest);
byte[] sbDigest = Bytes.toBytes(sDigest);
System.out.println("md5 digest as string length: " + sbDigest.length); // returns 26
```

不幸的是，用二进制表示会使数据在代码之外难以阅读。下例便是当需要增加一个值时会看到的shell：

```
hbase(main):001:0> incr 't', 'r', 'f:q', 1
COUNTER VALUE = 1

hbase(main):002:0> get 't', 'r'
COLUMN CELL
f:q timestamp=1369163040570, value=\x00\x00\x00\x00\x00\x00\x00\x01
1 row(s) in 0.0310 seconds
```

这个shell尽力在打印一个字符串，但在这种情况下，它决定只将进制打印出来。当在region名内行键会发生相同的情况。如果知道储存的是什么，那自是没问题，但当任意数据都可能被放到相同单元的时候，这将会变得难以阅读。这是最需要权衡之处。

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 （预裂）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（e.g., “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。

```
48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48                 // 0
54 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 -10 // 6
61 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -67 -68 // =
68 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -124 -126 // D
75 75 75 75 75 75 75 75 75 75 75 75 75 75 75 72 // K
82 18 18 18 18 18 18 18 18 18 18 18 18 18 18 14 // R
88 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -40 -44 // X
95 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -97 -102 // _
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。

教程1:预裂表（ 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。

教程2:16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。

一下case说明了如何16位键预分区

```
public static boolean createTable(Admin admin, HTableDescriptor table, byte[][] splits)
throws IOException {
try {
admin.createTable( table, splits );
return true;
} catch (TableExistsException e) {
logger.info("table " + table.getNameAsString() + " already exists");
// the table already exists...
return false;
}
}

public static byte[][] getHexSplits(String startKey, String endKey, int numRegions) {
byte[][] splits = new byte[numRegions-1][];
BigInteger lowestKey = new BigInteger(startKey, 16);
BigInteger highestKey = new BigInteger(endKey, 16);
BigInteger range = highestKey.subtract(lowestKey);
BigInteger regionIncrement = range.divide(BigInteger.valueOf(numRegions));
lowestKey = lowestKey.add(regionIncrement);
for(int i=0; i < numRegions-1;i++) {
BigInteger key = lowestKey.add(regionIncrement.multiply(BigInteger.valueOf(i)));
byte[] b = String.format("%016x", key).getBytes();
splits[i] = b;
}
return splits;
}
```

schema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

当修改列族时，建议先将这张表下线（disable）：

```
Configuration config = HBaseConfiguration.create();
HBaseAdmin admin = new HBaseAdmin(config);
String table = "Test";
```

```
admin.disableTable(table); // 将表下线

HColumnDescriptor f1 = ...;
admin.addColumn(table, f1); // 增加新的列族
HColumnDescriptor f2 = ...;
admin.modifyColumn(table, f2); // 修改列族
HColumnDescriptor f3 = ...;
admin.modifyColumn(table, f3); // 修改列族

admin.enableTable(table);
```

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(目前 ApsaraDB for HBase不支持，2.0会支持)，再大可以考虑直接存在OSS中
- 一个典型的表中含有1-3个列族，hbase表不应该设计成类似RDBMS的表
- 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。

当很多的列族在flush和compaction时,会造成很多没用的I/O负载(要想解决这个问题，需要将flush和compaction操作只针对一个列族) 。

尽量在模式中只针对一个列族操作。将使用率相近的列归为一列族，这样每次访问时就只用访问一个列族，提高效率。

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。设置方法：

```
# 建表时设置,TTL单位为秒，此例中列簇为'f'的数据保留30天（2592000秒）
hbase(main):002:0>create 'table', {NAME => 'f', TTL => 2592000}
# 通过修改表属性修改已有表的TTL
hbase(main):002:0>alter 'table', {NAME => 'f', TTL => 2592000}
```