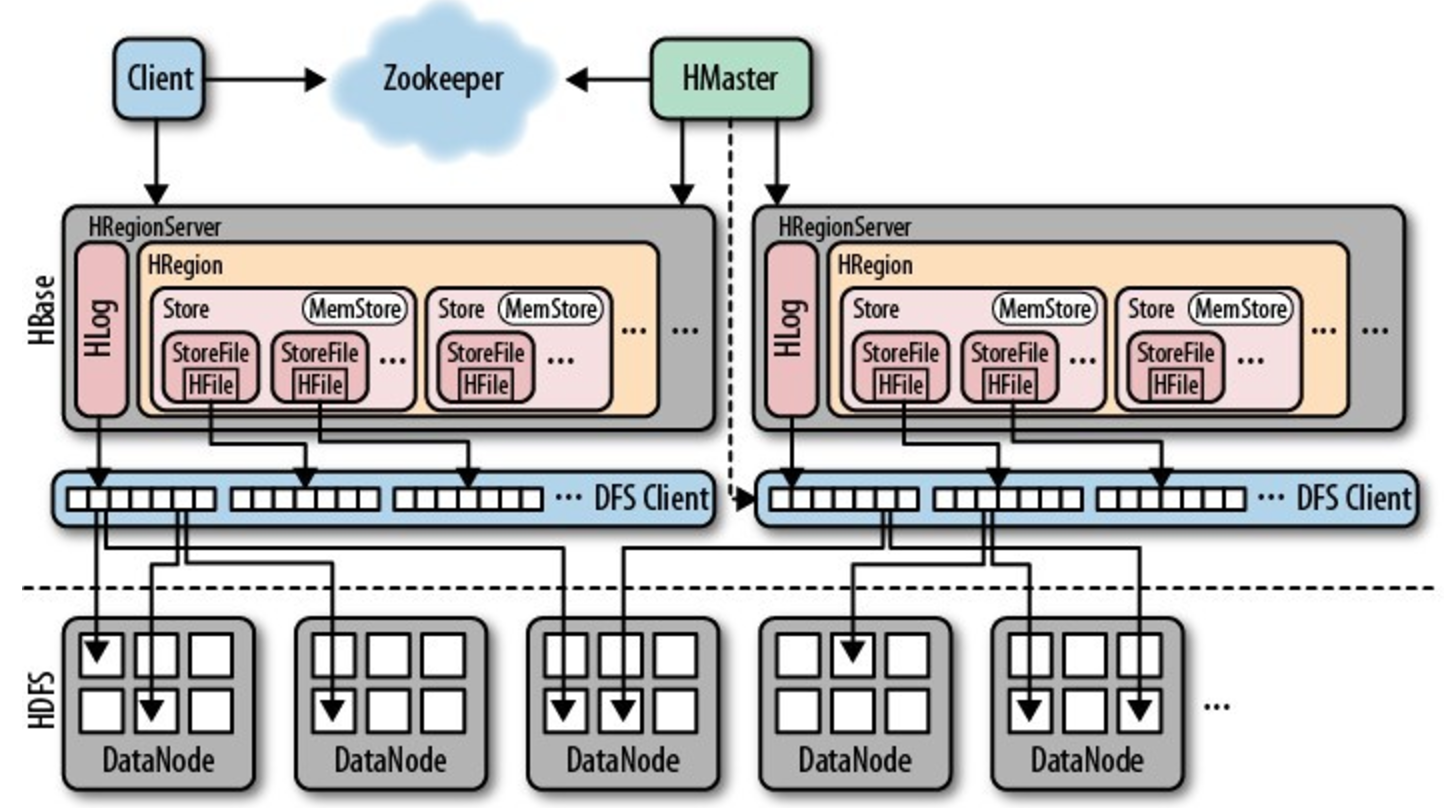
# 1：hbase 存储结构

## 存储结构图：



（图中有个错误，所有的Hregionserver共享一个Hlog）

## Hmaster

* 为Region server分配region
* 负责Region server的负载均衡
* 发现失效的Region server并重新分配其上的region
* HDFS上的垃圾文件回收
* 处理schema更新请求

## HRegionserver

* 维护master分配给他的region，处理对这些region的io请求
* 负责切分正在运行过程中变的过大的region

client访问hbase上的数据并不需要master参与（寻址访问zookeeper和region server，数据读写访问region server），master仅仅维护table和region的元数据信息（table的元数据信息保存在zookeeper上），负载很低。

HRegionServer存取一个子表时，会创建一个HRegion对象，然后对表的每个列族创建一个Store实例，每个Store都会有一个MemStore和0个或多个StoreFile与之对应，每个StoreFile都会对应一个HFile， HFile就是实际的存储文件。因此，一个HRegion有多少个列族就有多少个Store。

一个HRegionServer会有多个HRegion和一个HLog。

## HRegion

table在行的方向上分隔为多个Region。Region是HBase中分布式存储和负载均衡的最小单元，即不同的region可以分别在不同的Region Server上，但同一个Region是不会拆分到多个server上。

Region按大小分隔，每个表一行是只有一个region。随着数据不断插入表，region不断增大，**当region的某个列族达到一个阈值**（默认256M）时就会分成两个新的region。

## Store

每一个region有一个或多个store组成，至少是一个store，hbase会把一起访问的数据放在一个store里面，即为每个ColumnFamily建一个store，如果有几个ColumnFamily，也就有几个Store。一个Store由一个memStore和0或者多个StoreFile组成。

HBase以store的大小来判断是否需要切分region。

在一个region中有一个或多个stroe，每个stroe对应一个column families(列族)。一个store中包含一个memstore 和 0 或 多个store files。

## MemStore

memStore 是放在内存里的。保存修改的数据即keyValues。当memStore的大小达到一个阀值（默认128MB）时，memStore会被flush到文件，即生成一个快照。目前hbase 会有一个线程来负责memStore的flush操作。

## StoreFile

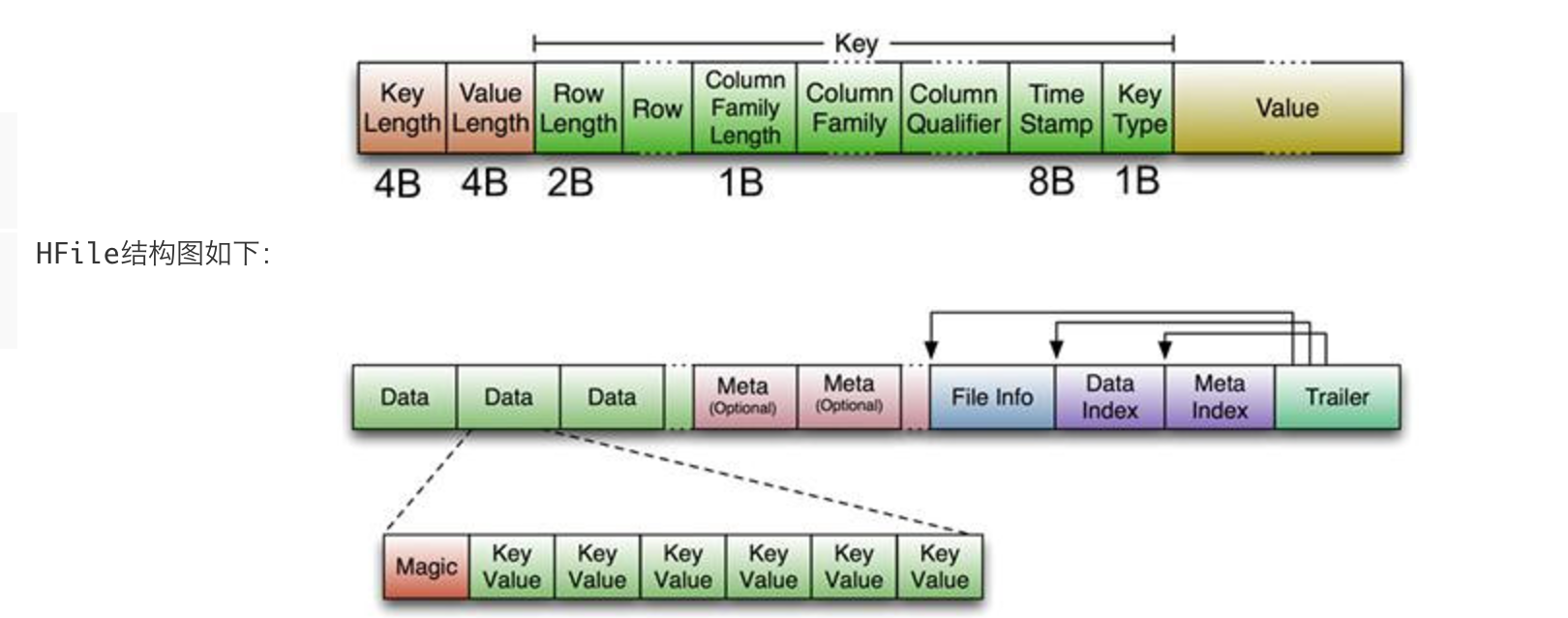
memStore内存中的数据写到文件后就是StoreFile，StoreFile底层是以HFile的格式保存。

## HFile

HBase中KeyValue数据的存储格式，是hadoop的二进制格式文件。

首先HFile文件是不定长的，长度固定的只有其中的两块：Trailer和FileInfo。Trailer中又指针指向其他数据块的起始点，FileInfo记录了文件的一些meta信息。

Data Block是hbase io的基本单元，为了提高效率，HRegionServer中又基于LRU的block cache机制。每个Data块的大小可以在创建一个Table的时候通过参数指定（默认块大小64KB），大号的Block有利于顺序Scan，小号的Block利于随机查询。每个Data块除了开头的Magic以外就是一个个KeyValue对拼接而成，Magic内容就是一些随机数字，目的是烦着数据损坏，结构如下。



Data Block段用来保存表中的数据，这部分可以被压缩。

Meta Block段（可选的）用来保存用户自定义的kv段，可以被压缩。

FileInfo段用来保存HFile的元信息，本能被压缩，用户也可以在这一部分添加自己的元信息。

Data Block Index段（可选的）用来保存Meta Blcok的索引。

Trailer这一段是定长的。保存了每一段的偏移量，读取一个HFile时，会首先读取Trailer，Trailer保存了每个段的起始位置(段的Magic Number用来做安全check)，然后，DataBlock Index会被读取到内存中，这样，当检索某个key时，不需要扫描整个HFile，而只需从内存中找到key所在的block，通过一次磁盘io将整个 block读取到内存中，再找到需要的key。DataBlock Index采用LRU机制淘汰。

HFile的Data Block，Meta Block通常采用压缩方式存储，压缩之后可以大大减少网络IO和磁盘IO，随之而来的开销当然是需要花费cpu进行压缩和解压缩。目标HFile的压缩支持两种方式：gzip、lzo。

## HLog

其实HLog文件就是一个普通的Hadoop Sequence File， Sequence File的value是key时HLogKey对象，其中记录了写入数据的归属信息，除了table和region名字外，还同时包括sequence number和timestamp，timestamp是写入时间，equence number的起始值为0，或者是最近一次存入文件系统中的equence number。

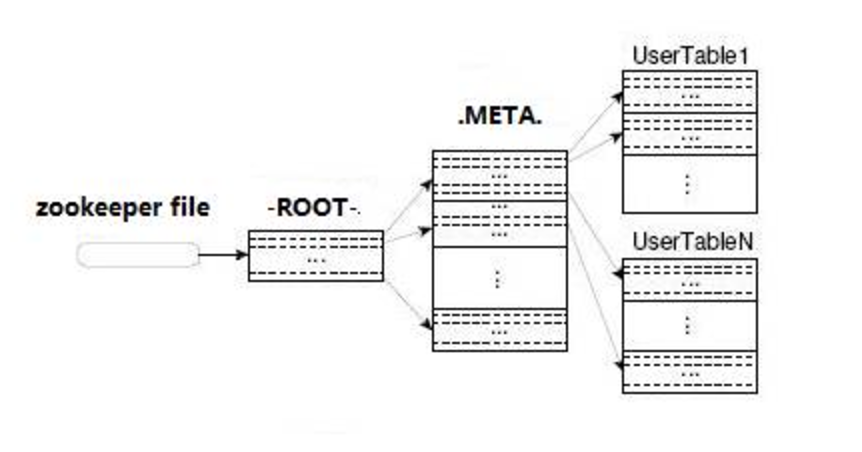
HLog(WAL log)：WAL意为write ahead log，用来做灾难恢复使用，HLog记录数据的所有变更，一旦region server 宕机，就可以从log中进行恢复。

# 2：查询流程

## 官方的解释：

HBase使用三层结构来定位region：

* 1、 通过zk里的文件/hbase/rs得到-ROOT-表的位置。-ROOT-表只有一个region。
* 2、通过-ROOT-表查找.META.表的第一个表中相应的region的位置。其实-ROOT-表是.META.表的第一个region；.META.表中的每一个region在-ROOT-表中都是一行记录。
* 3、通过.META.表找到所要的用户表region的位置。用户表中的每个region在.META.表中都是一行记录。



-ROOT-表永远不会被分隔为多个region，保证了最多需要三次跳转，就能定位到任意的region。

## 简单源码逻辑：

StoreScanner

StoreScanner

RegionScanner

MemStoreScanner

StoreFileScanner

HFileScanner

## Spark api操作(以Sql.Row为例)：

**val** hbconf = HBaseConfiguration.*create*()  
hbconf.set(**"hbase.zookeeper.quorum"**, HbaseUtil.*ZOOKEEPER\_IP*)  
hbconf.set(**"hbase.zookeeper.property.clientPort"**, HbaseUtil.*HBASE\_CLIENT\_PORT*)  
*//specify job output to hbase table***val** jobConf = **new** JobConf(hbconf, **this**.getClass)  
jobConf.setOutputFormat(*classOf*[TableOutputFormat])  
jobConf.set(TableOutputFormat.*OUTPUT\_TABLE*, *hbase\_table*)

rdd.map(row => {  
 **val** arr = row.toSeq.toArray  
 **val** rowkey = Bytes.*toBytes*(arr(0).toString)  
 **val** put = **new** Put(rowkey)  
 **for** (i <- 0 until arr.length) {  
 **if** (!row.isNullAt(i)){  
 put.addColumn(Bytes.*toBytes*(**"info"**), Bytes.*toBytes*(column\_name), Bytes.*toBytes*(arr(i).toString))  
 }  
 }  
 (**new** ImmutableBytesWritable(rowkey), put)  
}).repartition(20).saveAsHadoopDataset(jobConf)

# 3：插入流程

相对于hbase的查询，插入流程就相对比较简单。

1. client提交put请求
2. 记录日志WAL
3. 写入到memstore
4. 相应客户端插入成功
5. 从memstore flush到hdfs

hbase插入经验，最好批处理 list的长度：

*2\*hbase.client.write.buffer/put.heapSize()*

List<Put> puts = **new** ArrayList<>();  
**for**(**int** j=25000\*finalI; j<25000\*(finalI+1); j++) {  
 Put put = **new** Put(Bytes.*toBytes*((1000000 + j) + **""**));  
 put.addColumn(Bytes.*toBytes*(**"cf"**), Bytes.*toBytes*(**"aa"**), Bytes.*toBytes*(**"asjdflksjfk"**));  
 puts.add(put);  
 **if** (puts.size() == 9039) {  
 write.insert\_hbase(puts);  
 }  
}

**public void** insert\_hbase(List<Put> puts) {  
 **try** {  
 *//2\*hbase.client.write.buffer/put.heapSize()* **this**.**table**.put(puts);  
 } **catch** (IOException e) {  
 System.***out***.println(**"insert failed ...."**);  
 e.printStackTrace();  
 }  
}

# 4：hbase 重要功能解释

## compact

随着memstore的数据不断的flush到HFile，HFile文件的数量会逐渐增多，过多的文件数量，会增加IO的负载，因此HBase会尝试不断对这些文件进行合并。这个合并的过程叫做compact

compact分为两部分：

* Minor Compaction是指选取一些小的、相邻的StoreFile将他们合并成一个更大的StoreFile，在这个过程中不会处理已经Deleted或Expired的Cell。一次Minor Compaction的结果是更少并且更大的StoreFile。
* Major Compaction是指将所有的StoreFile合并成一个StoreFile，这个过程还会清理三类无意义数据：被删除的数据、TTL过期数据、版本号超过设定版本号的数据。另外，一般情况下，Major Compaction时间会持续比较长，整个过程会消耗大量系统资源，对上层业务有比较大的影响。因此线上业务都会将关闭自动触发Major Compaction功能，改为手动在业务低峰期触发。

HBase中可以触发compaction的因素有很多，最常见的因素有这么三种：Memstore Flush、后台线程周期性检查、手动触发。

1. Memstore Flush: 应该说compaction操作的源头就来自flush操作，memstore flush会产生HFile文件，文件越来越多就需要compact。因此在每次执行完Flush操作之后，都会对当前Store中的文件数进行判断，一旦文件数＃ > ，就会触发compaction。需要说明的是，compaction都是以Store为单位进行的，而在Flush触发条件下，整个Region的所有Store都会执行compact，所以会在短时间内执行多次compaction。

2. 后台线程周期性检查：后台线程CompactionChecker定期触发检查是否需要执行compaction，检查周期为：hbase.server.thread.wakefrequency\*hbase.server.compactchecker.interval.multiplier。和flush不同的是，该线程优先检查文件数＃是否大于，一旦大于就会触发compaction。如果不满足，它会接着检查是否满足major compaction条件，简单来说，如果当前store中hfile的最早更新时间早于某个值mcTime，就会触发major compaction，HBase预想通过这种机制定期删除过期数据。上文mcTime是一个浮动值，浮动区间默认为［7-7\*0.2，7+7\*0.2］，其中7为hbase.hregion.majorcompaction，0.2为hbase.hregion.majorcompaction.jitter，可见默认在7天左右就会执行一次major compaction。用户如果想禁用major compaction，只需要将参数hbase.hregion.majorcompaction设为0

3. 手动触发：一般来讲，手动触发compaction通常是为了执行major compaction，原因有三，其一是因为很多业务担心自动major compaction影响读写性能，因此会选择低峰期手动触发；其二也有可能是用户在执行完alter操作之后希望立刻生效，执行手动触发major compaction；其三是HBase管理员发现硬盘容量不够的情况下手动触发major compaction删除大量过期数据；无论哪种触发动机，一旦手动触发，HBase会不做很多自动化检查，直接执行合并。

### Compact相关配置优化

1. **<property>**
2. **<name>**hbase.hregion.majorcompaction**</name>**
3. **<value>**0**</value>**
4. **<description>**禁止majorcompaction，这里虽然禁止了，但是还是得做，是通过linux定时任务在空闲时间执行**</description>**
5. **</property>**
6. **<property>**
7. **<name>**hbase.hstore.compactionThreshold **</name>**
8. **<value>**3**</value>**
9. **<description>**一个store下面的storefile数量超过这个值，就会合并storefile为一个，默认是3，可以根据需要提高，一般提高到4 **</description>**
10. **</property>**

## split

随着region容量的增加，单个region的大小达到了限制，此时，就需要对region进行split，目前的split策略有：

ConstantSizeRegionSplitPolicy,IncreasingToUpperBoundRegionSplitPolicy还有有 KeyPrefixRegionSplitPolicy.

* ConstantSizeRegionSplitPolicy:当某个store（对应一个column family）的大小大于配置值 ‘hbase.hregion.max.filesize’的时候（默认10G）region就会自动分裂。
* IncreasingToUpperBoundRegionSplitPolicy:目前版本中默认的策略，这个策略中，最小的分裂大小和table的某个region server的region 个数有关，当store file的大小大于如下公式得出的值的时候就会split，公式如下

Min (R^2 \* “hbase.hregion.memstore.flush.size”, “hbase.hregion.max.filesize”) R为同一个table中在同一个region server中region的个数。

例如:

hbase.hregion.memstore.flush.size 默认值 128MB。

hbase.hregion.max.filesize默认值为10GB 。

* 如果初始时R=1,那么Min(128MB,10GB)=128MB,也就是说在第一个flush的时候就会触发分裂操作。
* 当R=2的时候Min(2\*2\*128MB,10GB)=512MB ,当某个store file大小达到512MB的时候，就会触发分裂。
* 如此类推，当R=9的时候，store file 达到10GB的时候就会分裂，也就是说当R>=9的时候，store file 达到10GB的时候就会分裂。
* KeyPrefixRegionSplitPolicy：可以保证相同的前缀的row保存在同一个region中，适合固定前缀的rowkey

### Split相关配置优化

<property>

        <name>hbase.regionserver.region.split.policy</name>

        <value>org.apache.hadoop.hbase.regionserver.ConstantSizeRegionSplitPolicy</value>

   </property>

<property>

         <name>hbase.hregion.max.filesize</name>

        <value>53687091200</value>

        <description>设置每个数据表中单个region存储的hfile最大值50G，只有超过此值才做split</description>

  </property>

这两个配置，会使自动split基本不会发生，这样就不会造成影响线上的查询、插入性能，但是需要手动的控制split。

## JVM

Hbase中的很多性能都是基于jvm的比例，所以jvm的大小，非常影响hbase的性能，比如：



hbase堆内存的设置，一般在默认情况下不需要修改。

## flush

### 触发flush

HBase会在如下几种情况下触发flush操作，需要注意的是MemStore的最小flush单元是HRegion而不是单个MemStore。可想而知，如果一个HRegion中Memstore过多，每次flush的开销必然会很大，因此我们也建议在进行表设计的时候尽量减少ColumnFamily的个数。

1. Memstore级别限制：当Region中任意一个MemStore的大小达到了上限（hbase.hregion.memstore.flush.size，默认128MB），会触发Memstore刷新。
2. Region级别限制：当Region中所有Memstore的大小总和达到了上限（hbase.hregion.memstore.block.multiplier \* hbase.hregion.memstore.flush.size，默认 2\* 128M = 256M），会触发memstore刷新。
3. Region Server级别限制：当一个Region Server中所有Memstore的大小总和达到了上限（hbase.regionserver.global.memstore.upperLimit ＊ hbase\_heapsize，默认 40%的JVM内存使用量），会触发部分Memstore刷新。Flush顺序是按照Memstore由大到小执行，先Flush Memstore最大的Region，再执行次大的，直至总体Memstore内存使用量低于阈值（hbase.regionserver.global.memstore.lowerLimit ＊ hbase\_heapsize，默认 38%的JVM内存使用量）。
4. 当一个Region Server中HLog数量达到上限（可通过参数hbase.regionserver.maxlogs配置）时，系统会选取最早的一个 HLog对应的一个或多个Region进行flush
5. HBase定期刷新Memstore：默认周期为1小时，确保Memstore不会长时间没有持久化。为避免所有的MemStore在同一时间都进行flush导致的问题，定期的flush操作有20000左右的随机延时。
6. 手动执行flush：用户可以通过shell命令 flush ‘tablename’或者flush ‘region name’分别对一个表或者一个Region进行flush。

### Flush流程

为了减少flush过程对读写的影响，HBase采用了类似于两阶段提交的方式，将整个flush过程分为三个阶段：

1. prepare阶段：遍历当前Region中的所有Memstore，将Memstore中当前数据集kvset做一个快照snapshot，然后再新建一个新的kvset。后期的所有写入操作都会写入新的kvset中，而整个flush阶段读操作会首先分别遍历kvset和snapshot，如果查找不到再会到HFile中查找。prepare阶段需要加一把updateLock对写请求阻塞，结束之后会释放该锁。因为此阶段没有任何费时操作，因此持锁时间很短。
2. flush阶段：遍历所有Memstore，将prepare阶段生成的snapshot持久化为临时文件，临时文件会统一放到目录.tmp下。这个过程因为涉及到磁盘IO操作，因此相对比较耗时。
3. commit阶段：遍历所有的Memstore，将flush阶段生成的临时文件移到指定的ColumnFamily目录下，针对HFile生成对应的storefile和Reader，把storefile添加到HStore的storefiles列表中，最后再清空prepare阶段生成的snapshot。

### Flush日志查看

Log文件中找：

/\*\*\*\*\*\*\* prepare阶段 \*\*\*\*\*\*\*\*/

Started memstore flush for

/\*\*\*\*\*\*\* flush阶段 \*\*\*\*\*\*\*\*/

Flushed

/\*\*\*\*\*\*\* commit阶段 \*\*\*\*\*\*\*\*/

regionserver.HStore: Added

### flush对读写的影响

一般情况下，比如memstore定期flush、手动执行flush操作、触发Memstore级别限制、触发HLog数量限制以及触发Region级别限制等，不会阻塞读写。

但是，一旦触发Region Server级别限制导致flush，就会对用户请求产生较大的影响。会阻塞所有落在该Region Server上的更新操作，阻塞时间很长，甚至可以达到分钟级别。一般情况下Region Server级别限制很难触发，但在一些极端情况下也不排除有触发的可能，下面分析一种可能触发这种flush操作的场景：

相关JVM配置以及HBase配置：

hbase.regionserver.global.memstore.upperLimit = 0.35

hbase.regionserver.global.memstore.lowerLimit = 0.30

针对这种情况，为了避免某个regionserver出现这种情况的flush，需要对这个节点上的region数量进行限制。举例：

regionserver heap size = 8G

hbase.regionserver.global.memstore.upperlimit=0.4

memstore.flush.size=256M

那么最大的region数量是

8\*1024\*0.4/256=12.8个region

# 压力测试

## 机器配置：

单核 16个cpu

|  |  |
| --- | --- |
| 核数 | 物理cpu个数 |
| 1 | 16 |

## 插入压力测试

插入方式对插入速度的影响：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 分区数 | 插入方式 | 并发数 | 列数 | 插入数量 | 耗时(单位S) | Qps |
| 1 | 逐个插入 | 20 | 1 | 45 6613 | 239.955 | 1902.55 |
| 1 | 批处理 | 20 | 1 | 100 0000 | 5.394 | 185391.17 |

并发数对插入速度的影响

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 分区数 | 并发数 | 列数 | 插入数量 | 耗时(单位S) | Qps |
| 1 | 20 | 1 | 100 0000 | 5.394 | 185391.17 |
| 1 | 40 | 1 | 100 0000 | 4.620 | 216450.21 |
| 1 | 50 | 1 | 100 0000 | 4.736 | 211148.64 |

多列方式

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 分区数 | 插入方式 | 列数 | 插入数量 | 耗时(单位S) | Qps |
| 2 | 逐个插入 | 1036 | 83757 | 267.451 | 313 |
| 2 | Hadoop保存 | 1036 | 83757 | 194.749 | 431 |
| 2 | 批处理 | 1036 | 83757 | 184.072 | 455 |
| 2 | 批处理 | 1036 | 83757 | 180.797 |  |

参考文档：

<https://yq.aliyun.com/articles/26336>

<http://abloz.com/hbase/book.html#config.files>

<https://github.com/apache/hbase/blob/rel/1.2.5/hbasecommon/src/main/resources/hbase-default.xml>

<http://www.cnblogs.com/shitouer/archive/2013/02/05/configuring-hbase-memstore-what-you-should-know.html>

<http://wangneng-168.iteye.com/blog/2067741>