**Compaction介绍**

Compaction是buffer->flush->merge的Log-Structured Merge-Tree模型的关键操作，主要起到如下几个作用：

1）合并文件

2）清除删除、过期、多余版本的数据

3）提高读写数据的效率

Minor & Major Compaction的区别

1）Minor操作只用来做部分文件的合并操作以及包括minVersion=0并且设置ttl的过期版本清理，不做任何删除数据、多版本数据的清理工作。

2）Major操作是对Region下的HStore下的所有StoreFile执行合并操作，最终的结果是整理合并出一个文件。

从这个功能上理解，Minor Compaction也不适合做Major的工作，因为部分的数据清理可能没有意义，例如，maxVersions=2，那么在少部分文件中，是否是kv仅有的2个版本也无法判断。

下面是引用：

There are two types of compactions: minor and major. Minor compactions will usually pick up a couple of the smaller adjacent StoreFiles and rewrite them as one. Minors do not drop deletes or expired cells, only major compactions do this. Sometimes a minor compaction will pick up all the StoreFiles in the Store and in this case it actually promotes itself to being a major compaction.

After a major compaction runs there will be a single StoreFile per Store, and this will help performance usually. Caution: major compactions rewrite all of the Stores data and on a loaded system, this may not be tenable; major compactions will usually have to be done manually on large systems.

**Compaction诱发因子**

在什么情况下会发生Compaction呢？

|  |  |  |
| --- | --- | --- |
| 参数名 | 配置项 | 默认值 |
| minFilesToCompact | hbase.hstore.compactionThreshold | 3 |
| maxFilesToCompact | hbase.hstore.compaction.max | 10 |
| maxCompactSize | hbase.hstore.compaction.max.size | Long.MAX\_VALUE |
| minCompactSize | hbase.hstore.compaction.min.size | memstoreFlushSize |

CompactionChecker是RS上的工作线程(Chore)，设置执行周期是通过threadWakeFrequency指定，大小通过hbase.server.thread.wakefrequency配置(默认10000)，然后乘以默认倍数multiple(1000),毫秒时间转换为秒。因此，在不做参数修改的情况下，CompactionChecker大概是2hrs, 46mins, 40sec执行一次。

首先，对于HRegion里的每个HStore进行一次判断，needsCompaction()判断是否足够多的文件触发了Compaction的条件。

条件为：HStore中StoreFIles的个数 – 正在执行Compacting的文件个数 > minFilesToCompact

操作：以最低优先级提交Compaction申请。

步骤1：选出待执行Compact的storefiles。由于在Store中的文件可能已经在进行Compacting，因此，这里取出未执行Compacting的文件，将其加入到Candidates中。

步骤2:执行compactSelection算法，在Candidates中选出需要进行compact的文件，并封装成CompactSelection对象当中。

1) 选出过期的store files。过滤minVersion=0，并且storefile.maxTimeStamp + store.ttl < now\_timestamp。这意味着整个文件最大的时间戳的kv，都已经过期了，从而证明整个storefile都已经过期了。CompactSelection如果发现这样的storefile，会优先选择出来，作为Min然后提交给Store进行处理。

这部分具体操作被封装在ScanQueryMatcher下的ColumnTracker中，在StoreScanner的遍历过程，ScannerQueryMatcher负责kv的过滤。这里的ScanType包括(MAJOR\_COMPACT,MINOR\_COMPACT,USER\_SCAN)，compact操作是对选出的文件执行一次标识ScanType为MAJOR\_COMPACT或者MINOR\_COMPACT类型的scan操作，然后将最终符合标准的kv存储在一个新的文件中。

应用重要参考：根据应用的需求设置ttl，并且设置minVersions=0，根据selectCompation优选清理过期不保留版本的文件的策略，这样会使得这部分数据在CompactionChecker的周期内被清理。

误区：在CompactSplitThread有两个配置项

hbase.regionserver.thread.compaction.large：配置largeCompactions线程池的线程个数，默认个数为1。

hbase.regionserver.thread.compaction.small：配置smallCompactions线程池的线程个数，默认个数为1。

这两个线程池负责接收处理CR(CompactionRequest),这两个线程池不是根据CR来自于Major Compaction和Minor Compaction来进行区分，而是根据一个配置hbase.regionserver.thread.compaction.throttle的设置值(一般在hbase-site.xml没有该值的设置)，而是采用默认值2 \* minFilesToCompact \* memstoreFlushSize，如果cr需要处理的storefile文件的大小总和，大于throttle的值，则会提交到largeCompactions线程池进行处理，反之亦然。

应用重要参考：可以稍微调大一些largeCompactions和smallCompactions线程池内线程的个数，建议都设置成5。

2) 判断是否需要进行majorCompaction，这是很多判断条件的合成，其中最为重要的一个是  
hbase.hregion.majorcompaction设置的值，也就是判断上次进行majorCompaction到当前的时间间隔，如果超过设置值，则满足一个条件，同时另外一个条件是compactSelection.getFilesToCompact().size() < this.maxFilesToCompact。

因此，通过设置hbase.hregion.majorcompaction = 0可以关闭CompactionChecke触发的major compaction，但是无法关闭用户调用级别的mc。

3) 过滤对于大文件进行Compaction操作。判断fileToCompact队列中的文件是否超过了maxCompactSize，如果超过，则过滤掉该文件，避免对于大文件进行compaction。

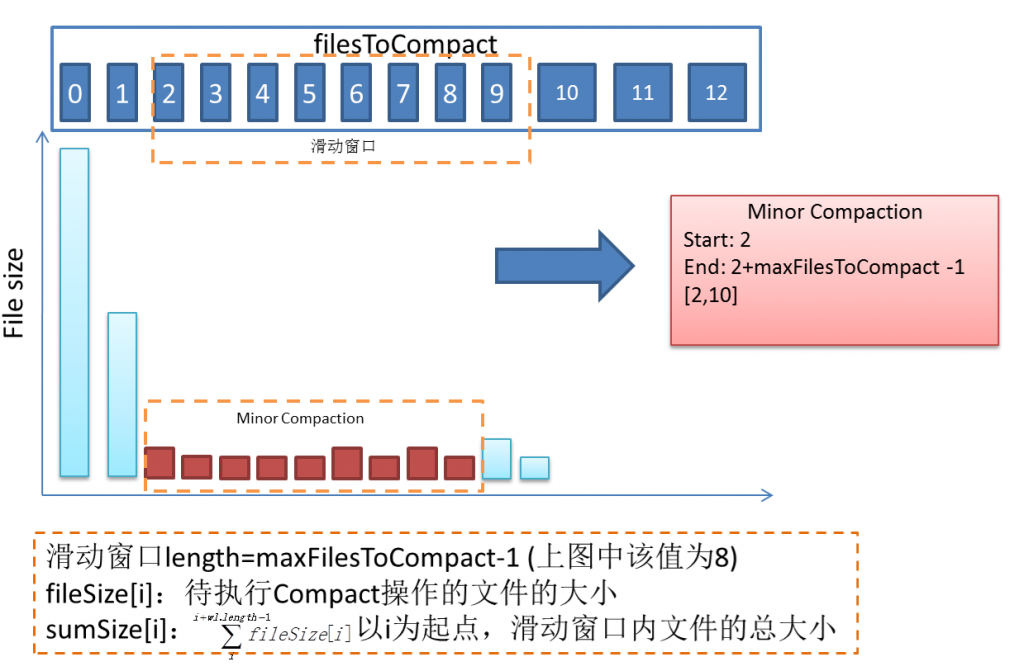
4) 如果确定Minor Compaction方式执行，会检查经过过滤过的fileToCompact的大小是否满足minFilesToCompact最低标准，如果不满足，忽略本次操作。确定执行的Minor Compaction的操作时，会使用一个smart算法，从filesToCompact当中选出匹配的storefiles。

具体算法为：

如果fileSizes[start] > Math.max(minCompactSize, (long)(sumSize[start+1]\*r )，那么继续start++。这里r的含义是compaction比例，它有如下四个参数控制：

|  |  |  |
| --- | --- | --- |
| 配置项 | 默认值 | 含义 |
| hbase.hstore.compaction.ratio | 1.2F |  |
| hbase.hstore.compaction.ratio.offpeak | 5.0F | 与下面两个参数联用 |
| hbase.offpeak.start.hour | -1 | 设置hbase offpeak开始时间[0,23] |
| hbase.offpeak.end.hour | -1 | 设置hbase offpeak结束时间 [0,23] |

如果默认没有设置offpeak时间的话，那么完全按照hbase.hstore.compaction.ration来进行控制。如下图所示，如果filesSize[i]过大，超过后面8个文件总和\*1.2，那么该文件被认为过大，而不纳入minor Compaction的范围。

  
Figure 1 Minor Compaction File Selection Algorithm

这样做使得Compaction尽可能工作在最近刷入hdfs的小文件的合并，从而使得提高Compaction的执行效率。

5) 通过selectCompaction选出的文件，加入到filesCompacting队列中。

6) 创建compactionRequest，提交请求。

总结：

在大多数情况下，Major是发生在storefiles和filesToCompact文件个数相同，并且满足各种条件的前提下执行。这里进行几个参数配置的简介：

hbase.hregion.majorcompaction： 设置系统进行一次MajorCompaction的启动周期，如果设置为0，则系统不会主动触发MC过程。

hbase.hstore.compaction.max：设置执行Compaction(包括Major &Minor)的待合并文件的最大个数。默认值为10，如果超过该设置值，会对部分文件执行一次MinorCompaction，选择算法如Figure1。

hbase.hstore.compactionThreshold: 设置执行Compaction(Major && Minor)操作的阈值，默认是3，如果想降低过频繁的合并操作，可以稍微调大一点，对于HBase负载较重的系统，可以设置成5。

**Compaction对于读写操作的影响**

Compaction与Flush不同之处在于：Flush是针对一个Region整体执行操作，而Compaction操作是针对Region上的一个Store而言，因此，从逻辑上看，Flush操作粒度较大。这属于一个LSM存储模型最核心的设计：

1）Flush操作如果只选择某个Region的Store内的MemStore写入磁盘，而不是统一写入磁盘，那么HLog上key的一致性在Reigon不同ColumnFamily(Store)下的MemStore内就会有不一致的key区间。

如下图所示，我们假定该RegionServer上仅有一个Region，由于不同的Row是在列簇上有所区别，就会出现有些不同Store内占用的内存不一致的情况，这里会根据整体内存使用的情况，或者RS使用内存的情况来决定是否执行Flush操作。如果仅仅刷入使用内存较大的memstore，那么在使用的过程中，一是Scan操作在执行时就不够统一，二是在HLog Replayer还原Region内Memstore故障前的状态，只需根据Hlog的Flush\_marker的标记位来执行Replay即可。  
  
2）Compaction执行结束之后会生成临时文件，临时文件所在的hdfs位置如下：

/hbase-weibo/bi\_weibo\_cluster/ffd87a50c3df3080183d4910d183d0ee/.tmp

ffd87a50c3df3080183d4910d183d0ee 是bi\_weibo\_cluster表格的Region名。临时文件的意义在于，在Compaction执行期间，对于原数据访问没有影响。Compaction执行合并操作生成的文件生效过程，需要对Store的写操作加锁，阻塞Store内的更新操作，直到更新Store的storeFiles完成为止。(注意，这个操作过程执行会影响到更新服务，但是影响不会太大)

3）对于读服务的影响，类似于Flush操作，也是通过ChangedReaderObserver为StoreScanner注册监听类来实现的。具体内容可以参考之前的”***HBase Flush操作流程以及对读写服务的影响***”。