# HBase性能调优

## 操作系统：

### 1.内存：

内存尽可能的大，不要饿着HBase。

### 2.64-bit

使用64位的操作系统。

### 3.Swapping

当心交换。swappiness设置为0。

Linux 移动那些一段时间没有被访问的内存页到 swap 空间，即使它有足够可用的内存。这叫做 swap out。换一句话说，从 swap 空间读 swapped out 的数据到内存中叫做 swap in。Swapping 在大多数情形是有必要的，但是因为 **Java Virtual Machine(JVM)** 在 swapping 下不是表现的很好，如果 swapped 了，HBase 运行可能会遇到问题。ZooKeeper 的 session 过期或许是 被 swap 引入的典型问题。

把 vm.swappiness 设置成 0，这将使得内核避免把进程尽可能的从物理内存中交换出去。这对 HBase 是非常有用的，因为 HBase 的进程消费大量的内存，一个高的 vm.swappiness 值将使得 HBase 交换频繁并遭遇非常慢的垃圾回收。随着 ZooKeeper session 超时，这可能会导致 RegionServer 进程被杀死。

如何修改swappiness：

查看

cat /proc/sys/vm/swappiness

修改

sysctl vm.swappiness=10

永久修改

**echo** "vm.swappiness = 10" >> /etc/sysctl.**conf**

### 4.CPU

确保您已经设置了您的Hadoop使用本地的、硬件的校验和。（core-site.xml io.native.lib.available=true）

## 二．网络：

## 三．java

### 1.GC垃圾回收器

在HBase中有两种stop-the-world GC的情况: CMS（并发GC）失效模式和老年代碎片问题。

第一个问题：

启动CMS前，将-XX:CMSInitiatingOccupancyFraction参数设置为低于默认值（92），以60或70％启动（降低阈值越低，完成的GCing越多，使用的CPU越多）

第二个问题：

本地memstore分配缓冲区MSLAB --代价是更加浪费堆空间

缓解由于memstore的扰动（不断创建和释放内存空间）造成region server 内存碎片问题。

原理：只允许从堆中分配相同的对象。一旦这些对象分配并且最终被回收，他们将在堆中留下固定大小的孔洞。之后调用相同大小的新对象将会重新使用这些孔洞：这样就不会产生提升错误，因此就不需要应用程序停止压缩回收了。默认在0.92版中被启用，0.90版本未启用，可以修改参数：**hbase.hregion.memstore.mslab.enabled**来覆盖

要注意的是，当启用这个属性时，每个MemStore实例将至少占用一个MSLAB内存实例。如果你在大量列族中都有成千上万个region，可能会浪费很大的堆空间，严重时导致OOME。在这种情况下要禁用MSLAB或者降低它的内存使用量，或者减少region server 中的region数量。

如果写压力过大，减少年轻代GC的配置。目的是让mslab直接写入老年代。

在hbase-env.sh 中配置GC **-xx:PretenureSizeThreshold**小于hbase.hregion.memstore.mslab.chunksize（默认2M）的大小。

## 四．HBase 配置参数

### 1. hbase.regionserver.handler.count

hbase.regionserver.handler.count 该属性定义了响应外部用户访问数据表请求的线程数。默认值30。对于较大的写入和使用大缓存的扫描，线程数设得比较小；而当单次请求开销较小时（如get，较小的put，increment和delete等操作）可以将工作线程数设得高一些。

### 2. hfile.block.cache.size

默认值：0.4  
说明：storefile的读缓存占用Heap的大小百分比，0.2表示20%。该值直接影响数据读的性能。

越大越好，如果写比读少很多，开到0.4-0.5也没问题。如果读写较均衡，0.3左右。如果写比读多，果断默认吧。设置这个值的时候，你同时要参考?hbase.regionserver.global.memstore.upperLimit?被hbase.regionserver.global.memstore.size代替，该值是memstore占heap的最大百分比，两个参数一个影响读，一个影响写。如果两值加起来超过80-90%，会有OOM的风险，谨慎设置。

### 3. Blockcache预取选项

HBase shell：

hbase> create 'MyTable', { NAME => 'myCF', PREFETCH\_BLOCKS\_ON\_OPEN => 'true' }

API：设置到某个列族

HTableDescriptor tableDesc = **new** HTableDescriptor("myTable");

HColumnDescriptor cfDesc = **new** HColumnDescriptor("myCF");

cfDesc.setPrefetchBlocksOnOpen(true);

tableDesc.addFamily(cfDesc);

### 4. hbase.regionserver.global.memstore.size(默认HEAP\_SIZE\*0.4)

### 5. hbase.regionserver.global.memstore.size.lower.limit

源码：MemstoreFlusher.java

upperlimit说明：hbase.hregion.memstore.flush.size 这个参数的作用是当单个Region，内所有的memstore大小总和超过指定值时，flush该region的所有memstore。RegionServer的flush是通过将请求添加一个队列，模拟生产消费模式来异步处理的。那这里就有一个问题，当队列来不及消费，产生大量积压请求时，可能会导致内存陡增，最坏的情况是触发OOM。  
这个参数的作用是防止内存占用过大，当ReigonServer内所有region的memstores所占用内存总和达到heap的40%时，HBase会强制block所有的更新并flush这些region以释放所有memstore占用的内存。

lowerLimit说明： 同upperLimit，只不过lowerLimit在所有region的memstores所占用内存达到hbase.regionserver.global.memstore.size的95%时，不flush所有的region。它会找一个适合的region，做个别flush，此时写更新还是会被block。lowerLimit算是一个在所有region强制flush导致性能降低前的补救措施。在日志中，表现为 “\*\* Flush thread woke up with memory above low water.”

调优：这是一个Heap内存保护参数，默认值已经能适用大多数场景。  
参数调整会影响读写，如果写的压力大导致经常超过这个阀值，则调小读缓存hfile.block.cache.size，增大该阀值，或者Heap余量较多时，只增加这个阀值。  
如果在高压情况下，也没超过这个阀值，那么建议你适当调小这个阀值再做压测，确保触发次数不要太多，然后还有较多Heap余量的时候，调大hfile.block.cache.size提高读性能。  
还有一种可能性是?hbase.hregion.memstore.flush.size保持不变，但RS维护了过多的region，要知道 region数量直接影响占用内存的大小。hfile.block.cache.size

### 6.hbase.hstore.blockingStoreFiles

默认值：7  
说明：在flush时（触发时机），当一个region中的Store（Coulmn Family）内有超过7个storefile时，则block所有的写请求进行compaction，以减少storefile数量。

调优：block写（compaction？）请求会严重影响当前regionServer的响应时间，但过多的storefile也会影响读性能（数据更分散了）。从实际应用来看，为了获取较平滑的响应时间，可将值设为无限大，无论有多少storeFile 都不会自动触发compaction。如果能容忍响应时间出现较大的波峰波谷，那么默认或根据自身场景调整即可。If there is blocking in the RegionServer logs,可以增大这个值。

### 7. hbase.hregion.memstore.block.multiplier

默认值：4

说明：当一个region里的memstore占用内存大小超过hbase.hregion.memstore.flush.size这些hbase.hregion.memstore.block.multiplier倍的大小时，block该region的所有请求，进行flush，释放内存。  
虽然我们设置了region所占用的memstores总内存大小，比如64M，但想象一下，在最后63.9M的时候，我Put了一个200M的数据，此时memstore的大小会瞬间暴涨到超过预期的hbase.hregion.memstore.flush.size的几倍。这个参数的作用是当memstore的大小增至超过hbase.hregion.memstore.flush.size 4倍时，block所有请求，遏制风险进一步扩大。

调优： 这个参数的默认值还是比较靠谱的。如果你预估你的正常应用场景（不包括异常）不会出现突发写或写的量可控，那么保持默认值即可。如果正常情况下，你的写请求量就会经常暴长到正常的几倍，那么你应该调大这个倍数并调整其他参数值，比如hfile.block.cache.size和hbase.regionserver.global.memstore.upperLimit/lowerLimit，以预留更多内存，防止HBase server OOM

### 8. hbase.regionserver.checksum.verify

 默认false，设置true，hbase使用自己的数据校验，而不是hdfs的校验；让HBase把校验和写到数据库中，并保存每次读取时都要进行校验和查找。（不用hdfs验有什么好处）

### 9. **hbase.hregion.max.filesize**

默认10G，一个region下，任一列簇的hfiles的大小，超过这个值，该region将split成2个region。

   note:如果你的数据量增长的比较快，那么还是建议把这个大小调高，可以调成100G，因为越少的region你的集群越流畅，100G的阈值基本可以避免你的region增长过快，甚至你的region数目会长期不变。当然大region在compaction时也会更加缓慢。几十G的region启动和compaction都非常的慢，如果storefile较多，一个compaction可能会持续几天。

### 10. hbase.regionserver.maxlogs

默认为32。它主要是控制WAL文件flush的频率

如果写操作比较多，那么可以设置高一点。调低可以让rs更快的把数据持久化，那么就可以直接弃掉WAL了。其实写操作比较多可以直接把WAL关闭，这样更省事了。

### 11. hbase.hstore.compactionThreshold/hbase.hregion.majorcompaction

hbase.hstore.compactionThreshold执行compaction的store数量，默认值是3，如果需要提高查询性能，当然是storefile的数量越小，性能越好，但是执行compaction本身有性能资源的开消，如果regionserver频繁在 compacion对性能影响也很大。hbase.hregion.majorcompaction表示majorcompaction的周期，默认是1 天，majorcompaction与普通的compaction的区别是majorcompaction会清除过期的历史版本数据，同时合并 storefile，而普通的compaction只做合并，通常都是majorcompaction

调为0，然后手工定期的去执行一下 majorcompaction，适当调小点compacionThreshold。

### 12. hbase.regionserver.codecs

启用数据压缩，推荐Snappy或者LZO压缩

### 13. hbase.master.wait.on.regionservers.mintostart

如果您有一个具有很多区域的群集，则在主程序启动后，所有其他RegionServers都会落后的情况下，Regionserver可能会进行短暂检查。这个第一个要签到的服务器将被分配给所有不是最优的区域，默认值1

增大数值

### 14. fail.fast.expired.active.master

如果master過期，那麼不需要從zk恢复，直接終止，默認是false；

改为true

### 15.zookeeper.session.timeout

默认值：3分钟（180000ms）  
说明：RegionServer与Zookeeper间的连接超时时间。当超时时间到后，ReigonServer会被Zookeeper从RS集群清单中移除，HMaster收到移除通知后，会对这台server负责的regions重新balance，让其他存活的RegionServer接管.

这个timeout决定了RegionServer是否能够及时的failover。设置成1分钟或更低，可以减少因等待超时而被延长的failover时间。  
不过需要注意的是，对于一些Online应用，RegionServer从宕机到恢复时间本身就很短的（网络闪断，crash等故障，运维可快速介入），如果调低timeout时间，反而会得不偿失。因为当ReigonServer被正式从RS集群中移除时，HMaster就开始做balance了（让其他RS根据故障机器记录的WAL日志进行恢复）。当故障的RS在人工介入恢复后，这个balance动作是毫无意义的，反而会使负载不均匀，给RS带来更多负担。特别是那些固定分配regions的场景。

### 16. dfs.datanode.failed.volumes.tolerated

hdfs-site.xml        决定停止数据节点提供服务充许卷的出错次数。默认值0，意思是任何卷出错都要停止数据节点

改成2合适

## 五．客户端API

当通过客户端使用接口读写数据时：

### 1.禁止自动刷写

### 2.使用扫描缓存

如果HBase被用作一个MAOReduce作业的输入源，请最好将作为MapReduce作业输入扫描器实例的缓存用setCaching（）方法设置为比默认值1大得多的值。使用默认的值意味着map任务会在处理每条记录时都请求region服务器。例如，将这个值设置为500，则一次可以传送500行数据到客户端进行处理。这里用户需要权衡传输数据的开销和内存的开销，因为缓存更大之后，无论是客户端还是服务器端都将消耗更多内存缓存数据，所以大的缓存不一定最好。

### 3.限定扫描范围

当scan被用来处理大量行时（特别是被用作MapReduce输入源时），注意哪些属性被选中了。如果Scan.addFamily()被调用了，那么特定列族中的所有列都将会被返回到客户端。如果只处理少数列，则应当只有这些列被添加到Scan的输入中，因为选择了过多的列将导致在大数据集上极大的效率损失，这可不是一件小事。

### 4.关闭ResultScanner

在try/catch中的finally块中关闭ResultScanner，避免一些问题。

### 5.块缓存用法

Scan实例能够通过setCacheBlocks（）方法来设置使用region服务器中的块缓存。如果MapReduce作业中使用扫描，这个方法应当被设置成false。对于那些频繁访问的行，建议使用块缓存。

### 6.优化获取行键的方式

当执行一个表的扫描以获取需要的行键时（没有列族/列名/列值和时间戳），在Scan中用setFilter（）方法添加一个带MUST\_PASS\_ALL操作符的FILterList。FilterList中包含FirstKeyFilter和KeyOnlyFilter两个过滤器。使用以上组合的过滤器将会把发现的第一个KeyValue行键（也就是第一列的行键）返回给客户端，这将会最大程度地减少网络传输。

### 7.关闭Put上的WAL（危险）

## 六．HBase写入

### 1.如果可以的话，用bulk load。

### 2.预创建分区

### 3.延迟日志刷新

WAL（预写式日志）默认行为是立即写入，如果开启延迟刷新，日志会先写入内存中直到刷新周期，优点是聚合和异步写入，但有一个风险，如果服务器宕机，未刷新的数据会宕机，但是总比不用WAL要安全。

延迟刷新可以在[HTableDescriptor](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/HTableDescriptor.html)中设置，也可以设置属性hbase.regionserver.optionallogflushinterval（默认值1000ms）。

### 4.禁止自动刷写

当有大量的写入操作时，使用setAutoFlush（false）方法，确认HTable自动刷写的特性已经被关闭。否则Put实例将会被逐个传送到region服务器。通过HTable.add(Put)和HTable.add(Put)添加的put实例都会添加到一个相同的写入缓存中，如果用户禁用了自动刷写，这些操作直到写缓冲区被填满时才会被送出。如果要显式地刷写数据，用户可以调用flushCommits()方法。调用HTable实例的close方法也会隐式地调用flushCommits()。

### 5.关闭Put上的WAL

一个经常讨论的提高写吞吐量的方式是使用Put的writeToWAL（false）来关闭WAL。这样服务器端就不会把这个Put写入到WAL中，而只是把它写到memstore里，不过一旦region服务器出现故障就会丢失数据。如果用户使用了writeToWAL（false），请小心。用户可能会发现把数据在集群间分布均匀后，关闭日志所带来的性能提升并不明显。

总而言之，最好是在写入数据时使用WAL，并且如果特别关心吞吐量的话，就用批量导入（bulk load）技术。

### 6.按RegionServer进行puts分组。

除了使用writeBuffer之外，按regionserver分组可以减少每个写缓冲区刷新时的客户端RPC调用的数量。可以调用HTableUtil实现，如果还在用0.90版本，可以自己实现。

### 7.跳过reducer

当将来自MR job的大量的数据写入一个HBase表时，特别是从mapper释放出来的，尽可能跳过reducer步骤（因为如果有reducer，数据先会写入磁盘，然后sort，shuffle。。）。如果把HBase作为source和sink的话，数据会从reducer写出（比如计算值并写出结果），这是另一种情况。

### 8.避免热点region

## 七.HBase读取

### 1. 使用扫描缓存

如果HBase被用作一个MAOReduce作业的输入源，请最好将作为MapReduce作业输入扫描器实例的缓存用setCaching（）方法设置为比默认值1大得多的值。使用默认的值意味着map任务会在处理每条记录时都请求region服务器。例如，将这个值设置为500，则一次可以传送500行数据到客户端进行处理。这里用户需要权衡传输数据的开销和内存的开销，因为缓存更大之后，无论是客户端还是服务器端都将消耗更多内存缓存数据，所以大的缓存不一定最好。

### 2. 扫描属性选择

无论何时使用扫描来处理大量的行(特别是当作为MapReduce源使用时)，请注意选择了哪些属性。如果调用scan.addFamily，在指定的ColumnFamily中所有的属性将被返回给客户端。如果只处理少量可用属性，那么只需要在输入扫描中指定这些属性，因为属性的过度选择对于大型数据集来说是一个不小的性能损失。

### 3. 避免扫描搜索

当使用scan.addColumn显式选择列时。HBase将在选定的列之间安排查找操作。当行数很少时，每个列只有几个版本，这可能是低效的。如果不至少查找5-10个列/版本或512-1024个字节，查找操作通常比较慢。

为了能适时地查看一些列/版本，以查看下一个列/版本是否可以在一个查找操作之前找到，可以在扫描对象上设置一个新的属性Scan.HINT\_LOOKAHEAD。下面的代码指导RegionServer寻找目标之前尝试下两个迭代:

Scan scan = **new** Scan();

scan.addColumn(...);

scan.setAttribute(Scan.HINT\_LOOKAHEAD, Bytes.toBytes(2));

table.getScanner(scan);

### 4. MapReduce - Input Splits

对于使用HBase表作为源的MapReduce作业，如果有一种情况，“慢”的map任务似乎有相同的输入分割，看这里：

[Case Study #1 (Performance Issue On A Single Node)](http://hbase.apache.org/1.2/book.html" \l "casestudies.slownode).

### 5.关闭ResultScanner

在try/catch中的finally块中关闭ResultScanner，避免一些问题。

### 6.块缓存用法

Scan实例能够通过setCacheBlocks（）方法来设置使用region服务器中的块缓存。如果MapReduce作业中使用扫描，这个方法应当被设置成false。对于那些频繁访问的行，建议使用块缓存。

### 7.优化获取行键的方式

当执行一个表的扫描以获取需要的行键时（没有列族/列名/列值和时间戳），在Scan中用setFilter（）方法添加一个带MUST\_PASS\_ALL操作符的FILterList。FilterList中包含FirstKeyFilter和KeyOnlyFilter两个过滤器。使用以上组合的过滤器将会把发现的第一个KeyValue行键（也就是第一列的行键）返回给客户端，这将会最大程度地减少网络传输。

### 8. 并发性:监测数据传播

在执行大量并发读操作时，监视目标表的数据传播。如果目标表的region太少，那么读操作可能会从很少的节点进行。

### 9. Bloom 过滤器

### 10. 对冲读取

对冲读取是HDFS-5776中引入的HDFS的一个特性。通常，为每个读请求生成一个线程。但是，如果启用了对冲操作，客户端会等待一些可配置的时间，如果读取不返回，则客户端会生成第二个read请求，针对相同数据的不同块复制。无论使用哪个读取返回，都将丢弃另一个读取请求。当一个罕见的慢读操作是由一个错误的磁盘或脆弱的网络连接引起的短暂的错误时，对其进行对冲操作可能会有帮助。

因为HBase RegionServer是一个HDFS客户端，所以可以在HBase中启用对冲操作，通过将以下属性添加到RegionServer节点。

<property>

<name>dfs.client.hedged.read.threadpool.size</name>

<value>20</value> *<!-- 20 threads 设置为0不启用-->*

</property>

<property>

<name>dfs.client.hedged.read.threshold.millis</name>

<value>10</value> *<!-- 10 milliseconds 启动第二个个线程等待的毫秒数-->*

</property>

## 八.HBase删除

#### 1.使用HBase表作为队列

HBase表有时被用作队列。在这种情况下，必须采取特别的措施，定期对以这种方式使用的表执行major compactions。正如在数据模型中记录的那样，标记行被删除会创建额外的store文件，然后需要在读取过程中进行处理。残留只有在major compactions情况下才会被清理干净。

#### 2.删除 RPC行为

注意，table.delete(Delete)不使用writeBuffer。它将在每次调用时执行一个RegionServer RPC。对于大量的删除操作，请考虑table.delete(List)。

## 九.HDFS

配置短读：

<property>

<name>dfs.client.read.shortcircuit</name>

<value>true</value>

<description>

This configuration parameter turns on short-circuit local reads.

</description>

</property>