[**HBase性能优化方法总结（一）：表的设计**](http://www.cnblogs.com/panfeng412/archive/2012/03/08/hbase-performance-tuning-section1.html)

本文主要是从HBase应用程序设计与开发的角度，总结几种常用的性能优化方法。有关HBase系统配置级别的优化，可参考：[淘宝Ken Wu同学的博客](http://kenwublog.com/hbase-performance-tuning)。

下面是本文总结的第一部分内容：表的设计相关的优化方法。

**1. 表的设计**

**1.1 Pre-Creating Regions**

默认情况下，在创建HBase表的时候会自动创建一个region分区，当导入数据的时候，所有的HBase客户端都向这一个region写数据，直到这个region足够大了才进行切分。一种可以加快批量写入速度的方法是通过预先创建一些空的regions，这样当数据写入HBase时，会按照region分区情况，在集群内做数据的负载均衡。

有关预分区，详情参见：[Table Creation: Pre-Creating Regions](http://hbase.apache.org/book.html#precreate.regions)，下面是一个例子：

copycode.gif

public static boolean createTable(HBaseAdmin 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;

}

**1.2 Row Key**

HBase中row key用来检索表中的记录，支持以下三种方式：

通过单个row key访问：即按照某个row key键值进行get操作；

通过row key的range进行scan：即通过设置startRowKey和endRowKey，在这个范围内进行扫描；

全表扫描：即直接扫描整张表中所有行记录。

在HBase中，row key可以是任意字符串，最大长度64KB，实际应用中一般为10~100bytes，存为byte[]字节数组，**一般设计成定长的**。

row key是按照**字典序**存储，因此，设计row key时，要充分利用这个排序特点，将经常一起读取的数据存储到一块，将最近可能会被访问的数据放在一块。

举个例子：如果最近写入HBase表中的数据是最可能被访问的，可以考虑将时间戳作为row key的一部分，由于是字典序排序，所以可以使用Long.MAX\_VALUE - timestamp作为row key，这样能保证新写入的数据在读取时可以被快速命中。

**1.3 Column Family**

**不要在一张表里定义太多的column family**。目前Hbase并不能很好的处理超过2~3个column family的表。因为某个column family在flush的时候，它邻近的column family也会因关联效应被触发flush，最终导致系统产生更多的I/O。感兴趣的同学可以对自己的HBase集群进行实际测试，从得到的测试结果数据验证一下。

**1.4 In Memory**

创建表的时候，可以通过HColumnDescriptor.setInMemory(true)将表放到RegionServer的缓存中，保证在读取的时候被cache命中。

**1.5 Max Version**

创建表的时候，可以通过HColumnDescriptor.setMaxVersions(int maxVersions)设置表中数据的最大版本，如果只需要保存最新版本的数据，那么可以设置setMaxVersions(1)。

**1.6 Time To Live**

创建表的时候，可以通过HColumnDescriptor.setTimeToLive(int timeToLive)设置表中数据的存储生命期，过期数据将自动被删除，例如如果只需要存储最近两天的数据，那么可以设置setTimeToLive(2 \* 24 \* 60 \* 60)。

**1.7 Compact & Split**

在HBase中，数据在更新时首先写入WAL 日志(HLog)和内存(MemStore)中，MemStore中的数据是排序的，当MemStore累计到一定阈值时，就会创建一个新的MemStore，并且将老的MemStore添加到flush队列，由单独的线程flush到磁盘上，成为一个StoreFile。于此同时， 系统会在zookeeper中记录一个redo point，表示这个时刻之前的变更已经持久化了**(minor compact)**。

StoreFile是只读的，一旦创建后就不可以再修改。因此Hbase的更新其实是不断追加的操作。当一个Store中的StoreFile达到一定的阈值后，就会进行一次合并**(major compact)**，将对同一个key的修改合并到一起，形成一个大的StoreFile，当StoreFile的大小达到一定阈值后，又会对 StoreFile进行分割**(split)**，等分为两个StoreFile。

由于对表的更新是不断追加的，处理读请求时，需要访问Store中全部的StoreFile和MemStore，将它们按照row key进行合并，由于StoreFile和MemStore都是经过排序的，并且StoreFile带有内存中索引，通常合并过程还是比较快的。

实际应用中，可以考虑必要时手动进行major compact，将同一个row key的修改进行合并形成一个大的StoreFile。同时，可以将StoreFile设置大些，减少split的发生。

[**HBase性能优化方法总结（二）：写表操作**](http://www.cnblogs.com/panfeng412/archive/2012/03/08/hbase-performance-tuning-section2.html)

本文主要是从HBase应用程序设计与开发的角度，总结几种常用的性能优化方法。有关HBase系统配置级别的优化，可参考：[淘宝Ken Wu同学的博客](http://kenwublog.com/hbase-performance-tuning)。

下面是本文总结的第二部分内容：写表操作相关的优化方法。

**2. 写表操作**

**2.1 多HTable并发写**

创建多个HTable客户端用于写操作，提高写数据的吞吐量，一个例子：

static final Configuration conf = HBaseConfiguration.create();

static final String table\_log\_name = “user\_log”;

wTableLog = new HTable[tableN];

for (int i = 0; i < tableN; i++) {

wTableLog[i] = new HTable(conf, table\_log\_name);

wTableLog[i].setWriteBufferSize(5 \* 1024 \* 1024); //5MB

wTableLog[i].setAutoFlush(false);

}

copycode.gif

**2.2 HTable参数设置**

**2.2.1 Auto Flush**

通过调用HTable.setAutoFlush(false)方法可以将HTable写客户端的自动flush关闭，这样可以批量写入数据到HBase，而不是有一条put就执行一次更新，只有当put填满客户端写缓存时，才实际向HBase服务端发起写请求。默认情况下auto flush是开启的。

**2.2.2 Write Buffer**

通过调用HTable.setWriteBufferSize(writeBufferSize)方法可以设置HTable客户端的写buffer大小，如果新设置的buffer小于当前写buffer中的数据时，buffer将会被flush到服务端。其中，writeBufferSize的单位是byte字节数，可以根据实际写入数据量的多少来设置该值。

**2.2.3 WAL Flag**

在HBae中，客户端向集群中的RegionServer提交数据时（Put/Delete操作），首先会先写WAL（Write Ahead Log）日志（即HLog，一个RegionServer上的所有Region共享一个HLog），只有当WAL日志写成功后，再接着写MemStore，然后客户端被通知提交数据成功；如果写WAL日志失败，客户端则被通知提交失败。这样做的好处是可以做到RegionServer宕机后的数据恢复。

因此，对于相对不太重要的数据，可以在Put/Delete操作时，通过调用Put.setWriteToWAL(false)或Delete.setWriteToWAL(false)函数，放弃写WAL日志，从而提高数据写入的性能。

**值得注意的是：谨慎选择关闭WAL日志，因为这样的话，一旦RegionServer宕机，Put/Delete的数据将会无法根据WAL日志进行恢复。**

**2.3 批量写**

通过调用HTable.put(Put)方法可以将一个指定的row key记录写入HBase，同样HBase提供了另一个方法：通过调用HTable.put(List<Put>)方法可以将指定的row key列表，批量写入多行记录，这样做的好处是批量执行，只需要一次网络I/O开销，这对于对数据实时性要求高，网络传输RTT高的情景下可能带来明显的性能提升。

**2.4 多线程并发写**

在客户端开启多个HTable写线程，每个写线程负责一个HTable对象的flush操作，这样结合定时flush和写buffer（writeBufferSize），可以既保证在数据量小的时候，数据可以在较短时间内被flush（如1秒内），同时又保证在数据量大的时候，写buffer一满就及时进行flush。下面给个具体的例子：

copycode.gif

for (int i = 0; i < threadN; i++) {

Thread th = new Thread() {

public void run() {

while (true) {

try {

sleep(1000); //1 second

} catch (InterruptedException e) {

e.printStackTrace();

}

synchronized (wTableLog[i]) {

try {

wTableLog[i].flushCommits();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

};

th.setDaemon(true);

th.start();

}

[**HBase性能优化方法总结（三）：读表操作**](http://www.cnblogs.com/panfeng412/archive/2012/03/08/hbase-performance-tuning-section3.html)

本文主要是从HBase应用程序设计与开发的角度，总结几种常用的性能优化方法。有关HBase系统配置级别的优化，可参考：[淘宝Ken Wu同学的博客](http://kenwublog.com/hbase-performance-tuning)。

下面是本文总结的第三部分内容：读表操作相关的优化方法。

**3. 读表操作**

**3.1 多HTable并发读**

创建多个HTable客户端用于读操作，提高读数据的吞吐量，一个例子：

copycode.gif

static final Configuration conf = HBaseConfiguration.create();

static final String table\_log\_name = “user\_log”;

rTableLog = new HTable[tableN];

for (int i = 0; i < tableN; i++) {

rTableLog[i] = new HTable(conf, table\_log\_name);

rTableLog[i].setScannerCaching(50);

}

copycode.gif

**3.2 HTable参数设置**

**3.2.1 Scanner Caching**

hbase.client.scanner.caching配置项可以设置HBase scanner一次从服务端抓取的数据条数，默认情况下一次一条。通过将其设置成一个合理的值，可以减少scan过程中next()的时间开销，代价是scanner需要通过客户端的内存来维持这些被cache的行记录。

有三个地方可以进行配置：1）在HBase的conf配置文件中进行配置；2）通过调用HTable.setScannerCaching(int scannerCaching)进行配置；3）通过调用Scan.setCaching(int caching)进行配置。**三者的优先级越来越高。**

**3.2.2 Scan Attribute Selection**

scan时指定需要的Column Family，可以减少网络传输数据量，否则默认scan操作会返回整行所有Column Family的数据。

**3.2.3 Close ResultScanner**

通过scan取完数据后，记得要关闭ResultScanner，否则RegionServer可能会出现问题（对应的Server资源无法释放）。

**3.3 批量读**

通过调用HTable.get(Get)方法可以根据一个指定的row key获取一行记录，同样HBase提供了另一个方法：通过调用HTable.get(List<Get>)方法可以根据一个指定的row key列表，批量获取多行记录，这样做的好处是批量执行，只需要一次网络I/O开销，这对于对数据实时性要求高而且网络传输RTT高的情景下可能带来明显的性能提升。

**3.4 多线程并发读**

在客户端开启多个HTable读线程，每个读线程负责通过HTable对象进行get操作。下面是一个多线程并发读取HBase，获取店铺一天内各分钟PV值的例子：

copycode.gif

public class DataReaderServer {

//获取店铺一天内各分钟PV值的入口函数

public static ConcurrentHashMap<String, String> getUnitMinutePV(long uid, long startStamp, long endStamp){

long min = startStamp;

int count = (int)((endStamp - startStamp) / (60\*1000));

List<String> lst = new ArrayList<String>();

for (int i = 0; i <= count; i++) {

min = startStamp + i \* 60 \* 1000;

lst.add(uid + "\_" + min);

}

return parallelBatchMinutePV(lst);

}

//多线程并发查询，获取分钟PV值

private static ConcurrentHashMap<String, String> parallelBatchMinutePV(List<String> lstKeys){

ConcurrentHashMap<String, String> hashRet = new ConcurrentHashMap<String, String>();

int parallel = 3;

List<List<String>> lstBatchKeys = null;

if (lstKeys.size() < parallel ){

lstBatchKeys = new ArrayList<List<String>>(1);

lstBatchKeys.add(lstKeys);

}

else{

lstBatchKeys = new ArrayList<List<String>>(parallel);

for(int i = 0; i < parallel; i++ ){

List<String> lst = new ArrayList<String>();

lstBatchKeys.add(lst);

}

for(int i = 0 ; i < lstKeys.size() ; i ++ ){

lstBatchKeys.get(i%parallel).add(lstKeys.get(i));

}

}

List<Future< ConcurrentHashMap<String, String> >> futures = new ArrayList<Future< ConcurrentHashMap<String, String> >>(5);

ThreadFactoryBuilder builder = new ThreadFactoryBuilder();

builder.setNameFormat("ParallelBatchQuery");

ThreadFactory factory = builder.build();

ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(lstBatchKeys.size(), factory);

for(List<String> keys : lstBatchKeys){

Callable< ConcurrentHashMap<String, String> > callable = new BatchMinutePVCallable(keys);

FutureTask< ConcurrentHashMap<String, String> > future = (FutureTask< ConcurrentHashMap<String, String> >) executor.submit(callable);

futures.add(future);

}

executor.shutdown();

// Wait for all the tasks to finish

try {

boolean stillRunning = !executor.awaitTermination(

5000000, TimeUnit.MILLISECONDS);

if (stillRunning) {

try {

executor.shutdownNow();

} catch (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

} catch (InterruptedException e) {

try {

Thread.currentThread().interrupt();

} catch (Exception e1) {

// TODO Auto-generated catch block

e1.printStackTrace();

}

}

// Look for any exception

for (Future f : futures) {

try {

if(f.get() != null)

{

hashRet.putAll((ConcurrentHashMap<String, String>)f.get());

}

} catch (InterruptedException e) {

try {

Thread.currentThread().interrupt();

} catch (Exception e1) {

// TODO Auto-generated catch block

e1.printStackTrace();

}

} catch (ExecutionException e) {

e.printStackTrace();

}

}

return hashRet;

}

//一个线程批量查询，获取分钟PV值

protected static ConcurrentHashMap<String, String> getBatchMinutePV(List<String> lstKeys){

ConcurrentHashMap<String, String> hashRet = null;

List<Get> lstGet = new ArrayList<Get>();

String[] splitValue = null;

for (String s : lstKeys) {

splitValue = s.split("\_");

long uid = Long.parseLong(splitValue[0]);

long min = Long.parseLong(splitValue[1]);

byte[] key = new byte[16];

Bytes.putLong(key, 0, uid);

Bytes.putLong(key, 8, min);

Get g = new Get(key);

g.addFamily(fp);

lstGet.add(g);

}

Result[] res = null;

try {

res = tableMinutePV[rand.nextInt(tableN)].get(lstGet);

} catch (IOException e1) {

logger.error("tableMinutePV exception, e=" + e1.getStackTrace());

}

if (res != null && res.length > 0) {

hashRet = new ConcurrentHashMap<String, String>(res.length);

for (Result re : res) {

if (re != null && !re.isEmpty()) {

try {

byte[] key = re.getRow();

byte[] value = re.getValue(fp, cp);

if (key != null && value != null) {

hashRet.put(String.valueOf(Bytes.toLong(key,

Bytes.SIZEOF\_LONG)), String.valueOf(Bytes

.toLong(value)));

}

} catch (Exception e2) {

logger.error(e2.getStackTrace());

}

}

}

}

return hashRet;

}

}

//调用接口类，实现Callable接口

class BatchMinutePVCallable implements Callable<ConcurrentHashMap<String, String>>{

private List<String> keys;

public BatchMinutePVCallable(List<String> lstKeys ) {

this.keys = lstKeys;

}

public ConcurrentHashMap<String, String> call() throws Exception {

return DataReadServer.getBatchMinutePV(keys);

}

}

**3.5 缓存查询结果**

对于频繁查询HBase的应用场景，可以考虑在应用程序中做缓存，当有新的查询请求时，首先在缓存中查找，如果存在则直接返回，不再查询HBase；否则对HBase发起读请求查询，然后在应用程序中将查询结果缓存起来。至于缓存的替换策略，可以考虑LRU等常用的策略。

**3.6 Blockcache**

HBase上Regionserver的内存分为两个部分，一部分作为Memstore，主要用来写；另外一部分作为BlockCache，主要用于读。

写请求会先写入Memstore，Regionserver会给每个region提供一个Memstore，当Memstore满64MB以后，会启动 flush刷新到磁盘。当Memstore的总大小超过限制时（heapsize \* hbase.regionserver.global.memstore.upperLimit \* 0.9），会强行启动flush进程，从最大的Memstore开始flush直到低于限制。

读请求先到Memstore中查数据，查不到就到BlockCache中查，再查不到就会到磁盘上读，并把读的结果放入BlockCache。由于BlockCache采用的是LRU策略，因此BlockCache达到上限(heapsize \* hfile.block.cache.size \* 0.85)后，会启动淘汰机制，淘汰掉最老的一批数据。

一个Regionserver上有一个BlockCache和N个Memstore，它们的大小之和不能大于等于heapsize \* 0.8，否则HBase不能启动。默认BlockCache为0.2，而Memstore为0.4。**对于注重读响应时间的系统，可以将 BlockCache设大些，比如设置BlockCache=0.4，Memstore=0.39，以加大缓存的命中率。**

[**HBase性能优化方法总结（四）：数据计算**](http://www.cnblogs.com/panfeng412/archive/2012/03/08/hbase-performance-tuning-section4.html)

本文主要是从HBase应用程序设计与开发的角度，总结几种常用的性能优化方法。有关HBase系统配置级别的优化，可参考：[淘宝Ken Wu同学的博客](http://kenwublog.com/hbase-performance-tuning)。

下面是本文总结的第四部分内容：数据计算相关的优化方法。

**4. 数据计算**

**4.1 服务端计算**

[Coprocessor](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/coprocessor/package-summary.html)运行于HBase RegionServer服务端，各个Regions保持对与其相关的coprocessor实现类的引用，coprocessor类可以通过RegionServer上classpath中的本地jar或HDFS的classloader进行加载。

目前，已提供有几种coprocessor：

Coprocessor：提供对于region管理的钩子，例如region的open/close/split/flush/compact等；

RegionObserver：提供用于从客户端监控表相关操作的钩子，例如表的get/put/scan/delete等；

Endpoint：提供可以在region上执行任意函数的命令触发器。一个使用例子是RegionServer端的列聚合，这里有[代码示例](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/coprocessor/package-summary.html#commandtarget)。

以上只是有关coprocessor的一些基本介绍，本人没有对其实际使用的经验，对它的可用性和性能数据不得而知。感兴趣的同学可以尝试一下，欢迎讨论。

**4.2 写端计算**

**4.2.1 计数**

HBase本身可以看作是一个可以水平扩展的Key-Value存储系统，但是其本身的计算能力有限（Coprocessor可以提供一定的服务端计算），因此，使用HBase时，往往需要从写端或者读端进行计算，然后将最终的计算结果返回给调用者。举两个简单的例子：

PV计算：通过在HBase写端内存中，累加计数，维护PV值的更新，同时为了做到持久化，定期（如1秒）将PV计算结果同步到HBase中，这样查询端最多会有1秒钟的延迟，能看到秒级延迟的PV结果。

分钟PV计算：与上面提到的PV计算方法相结合，每分钟将当前的累计PV值，按照rowkey + minute作为新的rowkey写入HBase中，然后在查询端通过scan得到当天各个分钟以前的累计PV值，然后顺次将前后两分钟的累计PV值相减，就得到了当前一分钟内的PV值，从而最终也就得到当天各个分钟内的PV值。

**4.2.2 去重**

对于UV的计算，就是个去重计算的例子。分两种情况：

如果内存可以容纳，那么可以在Hash表中维护所有已经存在的UV标识，每当新来一个标识时，通过快速查找Hash确定是否是一个新的UV，若是则UV值加1，否则UV值不变。另外，为了做到持久化或提供给查询接口使用，可以定期（如1秒）将UV计算结果同步到HBase中。

如果内存不能容纳，可以考虑采用Bloom Filter来实现，从而尽可能的减少内存的占用情况。除了UV的计算外，判断URL是否存在也是个典型的应用场景。

**4.3 读端计算**

如果对于响应时间要求比较苛刻的情况（如单次http请求要在毫秒级时间内返回），个人觉得读端不宜做过多复杂的计算逻辑，尽量做到读端功能单一化：即从HBase RegionServer读到数据（scan或get方式）后，按照数据格式进行简单的拼接，直接返回给前端使用。当然，如果对于响应时间要求一般，或者业务特点需要，也可以在读端进行一些计算逻辑。

**5. 总结**

作为一个Key-Value存储系统，HBase并不是万能的，它有自己独特的地方。因此，基于它来做应用时，我们往往需要从多方面进行优化改进（表设计、读表操作、写表操作、数据计算等），有时甚至还需要从系统级对HBase进行配置调优，更甚至可以对HBase本身进行优化。这属于不同的层次范畴。

总之，概括来讲，对系统进行优化时，首先定位到影响你的程序运行性能的瓶颈之处，然后有的放矢进行针对行的优化。如果优化后满足你的期望，那么就可以停止优化；否则继续寻找新的瓶颈之处，开始新的优化，直到满足性能要求。