HBase性能优化方法总结（三）：读表操作

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。

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：

```
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);
}
```

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：

```
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。**

有关BlockCache机制，请参考这里：HBase的Block cache，HBase的blockcache机制，hbase中的缓存的计算与使用。