一、HBase相关

1、HBase压缩问题？是否使用了压缩？使用了哪种压缩

一般有Snappy 跟GZIP两种压缩方式，HBase默认使用Gzip的压缩方式

压缩可以节省空间，但是在读取的时候更占用cpu

2、HBase的优化问题

1）行键的优化：散列行键，以解决热点问题（行键过于散列之后会使查询的速度变慢）

2）列族的优化：

3、HBase写入数据的过程

a)  Client发起了一个HTable.put(Put)请求给HRegionServer

b)  HRegionServer会将请求匹配到某个具体的HRegion上面

c)  决定是否写WAL log。WAL log文件是一个标准的Hadoop SequenceFile，文件中存储了HLogKey，这些Keys包含了和实际数据对应的序列号，主要用于崩溃恢复。

d) Put数据保存到MemStore中，同时检查MemStore状态，如果满了，则触发Flush to Disk请求。

e)  HRegionServer处理Flush to Disk的请求，将数据写成HFile文件并存到HDFS上，并且存储最后写入的数据序列号，这样就可以知道哪些数据已经存入了永久存储的HDFS中。

由于不同的列族会共享region，所以有可能出现，一个列族已经有1000万行，而另外一个才100行。当一个要求region分割的时候，会导致100行的列会同样分布到多个region中。所以，一般建议不要设置多个列族。

4、HBase读取数据的过程

　　Client访问用户数据之前需要首先访问zookeeper，然后访问-ROOT-表，接着访问.META.表，最后才能找到用户数据的位置去访问，中间需要多次网络操作，不过client端会做cache缓存。

1)Client会通过内部缓存的相关的-ROOT-中的信息和.META.中的信息直接连接与请求数据匹配的HRegion server；

    2)然后直接定位到该服务器上与客户请求对应的region，客户请求首先会查询该region在内存中的缓存——memstore(memstore是是一个按key排序的树形结构的缓冲区)；

    3)如果在memstore中查到结果则直接将结果返回给client；

    4)在memstore中没有查到匹配的数据，接下来会读已持久化的storefile文件中的数据。storefile也是按key排序的树形结构的文件——并且是特别为范围查询或block查询优化过的，；另外hbase读取磁盘文件是按其基本I/O单元(即 hbase block)读数据的。具体就是过程就是：

    如果在BlockCache中能查到要造的数据则这届返回结果，否则就读去相应的storefile文件中读取一block的数据，如果还没有读到要查的数据，就将该数据block放到HRegion Server的blockcache中，然后接着读下一block块儿的数据，一直到这样循环的block数据直到找到要请求的数据并返回结果；如果将该region中的数据都没有查到要找的数据，最后接直接返回null，表示没有找的匹配的数据。当然blockcache会在其大小大于一的阀值（heapsize \* hfile.block.cache.size \* 0.85）后启动基于LRU[**算法**](http://lib.csdn.net/base/datastructure)的淘汰机制，将最老最不常用的block删除。

5、HBase中两个特殊的表（root 、meta） 及两个表的作用

HBase中有两张特殊的Table，-ROOT-和.META.

.META.：记录了用户表的Region信息，.META.可以有多个regoin，以及RegionServer的服务器地址。

-ROOT-：记录了.META.表的Region信息，-ROOT-只有一个region

&Oslash; Zookeeper中记录了-ROOT-表的location

6、HBase的接口类 ： 应该是ConnectionFactory 和 HBaseConfiguration

7、HBase的Region的分裂条件：

从 HBase 0.94之后，默认的分裂策略是IncreasingToUpperBoundRegionSplitPolicy

，思想就是当Region的大小超过某个阈值时，即进行分裂。

这个阈值主要由如下几个因素决定:

hbase.hregion.max.filesize

hbase.increasing.policy.initial.size

当前Region所在RegionServer上和当前Region属于同一张表的Region个数

根据以上三个因素算出一个阈值后，如果当前Region有某个Store的大小大于这个阈值，则认为该Region可分裂，这里对于Store还有一个条件就是Store下不能存在reference类型的StoreFile，这种reference类型的StoreFile是一次分裂后产生的，后续会详说。

对于IncreasingToUpperBoundRegionSplitPolicy来说，使用基类中默认的getSplitPoint()函数，即将Region中size最大的Store下最大的StoreFile拿出来，然后根据block index找出StoreFile中间的block，那么这个block的startkey就是split point

8、HBase的架构（各个组件及其作用）：

HMaster HRegionServer HRegion 列族（family） 行键（row-key）

9、HBase读取性能的优化

这里只从开发的角度谈HBase的优化

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

2）HTable参数设置

① Scanner Caching

通过调用HTable.setScannerCaching(int scannerCaching)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。通过将此值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。

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 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） 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。

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：

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。

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。**

10、HBase的模糊查询

使用scan设置startKey和endKey实现，或者使用Filter进行实现，例如：

scan 'scores',{FILTER=>org.apache.hadoop.hbase.filter.PrefixFilter.new(org.apache.hadoop.hbase.util.Bytes.toBytes('Jerry'))}

11、HBase使用Hive创建索引

12、MemStore flush的条件：

注意每次flush操作，会flush一个HRegion中所有的MemStore，而不是只flush一个MemStore。以下情况可以触发MemStore的flush动作：

    当一个MemStore的大小超过了hbase.hregion.memstore.flush.size的大小，默认128MB。此时当前的HRegion中所有的MemStore会flush到HDFS中。这个参数表示单个MemStore的大小的阈值。这个时候是不阻塞写操作的。

    当一个Region的所有MemStore的总量达到hbase.hregion.memstore.block.multiplier \* hbase.hregion.memstore.flush.size（默认4\*128M=512M）时，会阻塞这个region的写操作，并强制刷写到HFile。触发这个刷新只会发生在MemStore即将写满128M时put了一个巨大的记录的情况，这时会阻塞写操作，强制刷新成功才能继续写入。当强制刷新时，会有一个后台线程检查MemStore的内存变化，检查周期是hbase.server.thread.wakefrequency配置的，默认是10秒。

一个RegionServer会有很多个Region，很多的MemStore，可能单个Region并没有超过阈值，但是整个RegionServer的内存已经占用非常多了，这时候还有另外两个参数控制内存的刷写：hbase.regionserver.global.memstore.upperLimit 默认0.4，当RegionServer上全部的MemStore占用超过heap(heap的大小在hbase-env.sh中设置HBASE\_HEAPSIZE，默认1G)的40%时，强制阻塞所有的写操作，将所有的MemStore刷写到HFile； hbase.regionserver.global.memstore.lowerLimit 默认0.38，表示所有的MemStore占用超过heap的38%时，会选择一些占用内存比较大的MemStore阻塞写操作并进行flush，这是为了降低阻塞全部写操作flush带来的问题。所以，如果一个RegionServer中的Region数量很多，那么很容易达到upperLimit的阈值，导致flush频繁发生。

定期会进行MemStore的刷新，hbase.regionserver.optionalcacheflushinterval 默认3600000毫秒，即1小时，确保MemStore的数据不会长时间停留在内存中。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20秒左右的随机延时。

手工执行flush操作命令

13、RegoinServer挂掉后会怎样？

数据不会丢失，但是无法读写数据

RegionServer宕机是由zookeeper首先感知到的，而zookeeper感知到RegionServer宕机事件是需要一定时间的，这段时间默认会有3min。也就是说，在RegionServer宕机之后的3min之内系统并不知晓它实际上已经宕机了，所有的读写路由还会正常落到它上面，可想而知，这些读写必然都会失败。(当然，并不是所有RegionServer宕机都需要3min中才能被Zookeeper感知。如果RegionServer在运行过程中产生自身难以解决的问题，它会自己abort自己，并且RegionServer会主动通知Zookeeper自己已经宕机的事实。这种场景下，影响用户读写的时间会极大的缩短到秒级)Zookeeper一旦感知到RegionServer宕机之后，就会第一时间通知集群的管理者Master，Master首先会将这台RegionServer上所有Region移到其他RegionServer上，再将HLog分发给其他RegionServer进行回放，这个过程通常会很快。完成之后再修改路由，业务方的读写才会恢复正常。

14、如何进行多条件查询？ 使用filter

15、如何查看表中有多少条数据

数据量小的时候可以使用 cout tablename1；

数据量大的时候可以使用compressor

二、HDFS相关：

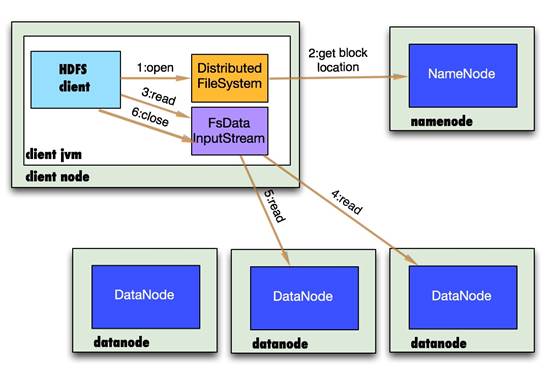
1、HDFS的架构？

NameNode、secondryNameNode、DataNode

2、HDFS的secondryNameNode的作用

SecondaryNameNode节点启动之后会不断的对NameNode节点保存的元数据进行备份(定时备份)，具体的说来就是：SecondaryNameNode的run方法每隔一段时间就会执行doCheckpoint()方法，SecondaryNameNode的主要工作都在这个方法里。这个方法会从NameNode上取下FSImage和操作日志(当然也包括版本文件和fstime)，然后在本地合并，然后再把合并后的FSImage传回NameNode。

3、**数据存储：读文件解析**



  1)首先调用FileSystem对象的open方法，其实是一个DistributedFileSystem的实例

  2)DistributedFileSystem通过rpc获得文件的第一个block的locations，同一block按照副本数会返回多个locations，这些locations按照hadoop拓扑结构排序，距离客户端近的排在前面.

  3)前两步会返回一个FSDataInputStream对象，该对象会被封装成DFSInputStream对象，DFSInputStream可以方便的管理datanode和namenode数据流。客户端调用read方法，DFSInputStream最会找出离客户端最近的datanode并连接。

  4)数据从datanode源源不断的流向客户端。

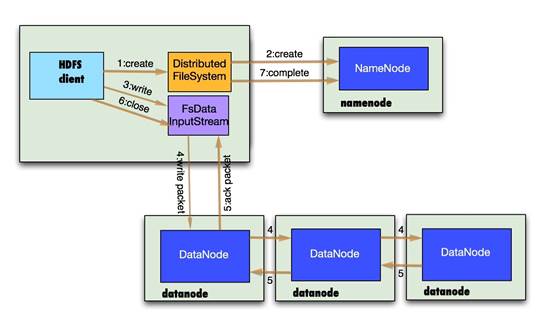
  5)如果第一块的数据读完了，就会关闭指向第一块的datanode连接，接着读取下一块。这些操作对客户端来说是透明的，客户端的角度看来只是读一个持续不断的流。

  6)如果第一批block都读完了，DFSInputStream就会去namenode拿下一批blocks的location，然后继续读，如果所有的块都读完，这时就会关闭掉所有的流。

如果在读数据的时候，DFSInputStream和datanode的通讯发生异常，就会尝试正在读的block的排第二近的datanode,并且会记录哪个datanode发生错误，剩余的blocks读的时候就会直接跳过该datanode。DFSInputStream也会检查block数据校验和，如果发现一个坏的block,就会先报告到namenode节点，然后DFSInputStream在其他的datanode上读该block的镜像

该设计的方向就是客户端直接连接datanode来检索数据并且namenode来负责为每一个block提供最优的datanode，namenode仅仅处理block location的请求，这些信息都加载在namenode的内存中，hdfs通过datanode集群可以承受大量客户端的并发访问。

**4、数据存储：写文件解析**



  1）客户端通过调用DistributedFileSystem的create方法创建新文件

  2）DistributedFileSystem通过RPC调用namenode去创建一个没有blocks关联的新文件，创建前，namenode会做各种校验，比如文件是否存在，客户端有无权限去创建等。如果校验通过，namenode就会记录下新文件，否则就会抛出IO异常.

  3）前两步结束后会返回FSDataOutputStream的对象，象读文件的时候相似，FSDataOutputStream被封装成DFSOutputStream.DFSOutputStream可以协调namenode和datanode。客户端开始写数据到DFSOutputStream,DFSOutputStream会把数据切成一个个小packet，然后排成队列data quene。

  4）DataStreamer会去处理接受data queue，他先问询namenode这个新的block最适合存储的在哪几个datanode里，比如副本数是3，那么就找到3个最适合的datanode，把他们排成一个pipeline.DataStreamer把packet按队列输出到管道的第一个datanode中，第一个datanode又把packet输出到第二个datanode中，以此类推。

  5）DFSOutputStream还有一个对列叫ack queue，也是有packet组成，等待datanode的收到响应，当pipeline中的所有datanode都表示已经收到的时候，这时akc queue才会把对应的packet包移除掉。

如果在写的过程中某个datanode发生错误，会采取以下几步：

①pipeline被关闭掉；

②为了防止丢包ack queue里的packet会同步到data queue里；

③把产生错误的datanode上当前在写但未完成的block删掉；

④block剩下的部分被写到剩下的两个正常的datanode中；

⑤namenode找到另外的datanode去创建这个块的复制。当然，这些操作对客户端来说是无感知的。

 6）客户端完成写数据后调用close方法关闭写入流

7）DataStreamer把剩余得包都刷到pipeline里然后等待ack信息，收到最后一个ack后，通知datanode把文件标示为已完成。

  另外要注意得一点，客户端执行write操作后，写完得block才是可见的，正在写的block对客户端是不可见的，只有调用sync方法，客户端才确保该文件被写操作已经全部完成，当客户端调用close方法时会默认调用sync方法。是否需要手动调用取决你根据程序需要在数据健壮性和吞吐率之间的权衡。

三、Kafka：

1、相关组件及其作用：topic partition replication-factor

2、Kafka数据延迟问题如何解决？（当Kafka的一个节点宕机时会出现数据延迟，对于实时处理的项目影响很大）

四、Storm （Storm问的相对较少，现在基本都用SparkStreaming）

1、Storm的优化问题

2、Storm跟SparkStreaming的区别：

1）Storm是一条条处理数据 SparkStreaming是按照时间窗口，一批批处理

2）Storm运行依赖自己的集群，SparkStreaming可以运行在Spark集群上，也可以运行在Hadoop集群上

3）Strom的精度更高（毫秒级），SparkStreaming的吞吐量更大

3、storm针对某个渠道交易进行处理，假如渠道A 90%数据量 渠道B 只有10%，那会出现什么问题，然后怎么解决的？ 回到增加下游的子bolt，扩充并行度再分流

4、也到了项目中storm的处理，用多少种运算？相应的运算数据存储在哪？意思是永久或临时吧

5、ack具体实现机制

6、如何防止处理的数据丢失，从kafka 的备份因子说的。。。

五、MR

1、运行过程

1） map任务处理

1.1 框架使用InputFormat类的子类把输入文件(夹)划分为很多InputSplit，默认，每个HDFS的block对应一个InputSplit。通过RecordReader类，每个InputSplit解析成一个个<k1,v1>。默认，框架对每个InputSplit中的每一行，解析成一个<k1,v1>。

1.2 框架调用Mapper类中的map(...)函数，map函数的形参是<k1,v1>对，输出是<k2,v2>对。一个InputSplit对应一个map task。程序员可以覆盖map函数，实现自己的逻辑。

1.3 (假设reduce不存在)框架对map结果直接输出到HDFS中。

      (假设reduce存在)框架对map输出的<k2,v2>进行分区。不同的分区中的<k2,v2>由不同的reduce task处理。默认只有1个分区。

1.4 (假设reduce存在)框架对每个分区内部的数据，按照k2进行排序、分组。分组指的是相同k2的v2分成一个组。注意：分组不会减少<k2,v2>数量。

1.5 (假设reduce存在，可选)在map节点，框架可以执行reduce归约。

1.6 (假设reduce存在)框架会对map task输出的<k2,v2>写入到linux 的磁盘文件中。

至此，整个map阶段结束。

2）.reduce任务处理

2.1 框架对多个map任务的输出，按照不同的分区，通过网络copy到不同的reduce节点。这个过程称作shuffle。

2.2 框架对reduce端接收的<k2,v2>数据进行合并、排序、分组。

2.3 框架调用Reducer类中的reduce方法，reduce方法的形参是<k2,{v2}>分组，输出是<k3,v3>。一个<k2,{v2}>分组调用一次reduce函数。程序员可以覆盖reduce函数，实现自己的逻辑。

2.4 框架把reduce的输出保存到HDFS中。

至此，整个reduce阶段结束。

2、一次MR数据写入磁盘的次数

有map端的spill时 是3次数据写磁盘，没有的时候是2次

3、优化

MapReduce计算模型的优化涉及了方方面面的内容，但是主要集中在两个方面：

一是计算性能方面的优化；

二是I/O操作方面的优化。这其中，又包含六个方面的内容。

1) 任务调度

任务调度是[**Hadoop**](http://lib.csdn.net/base/hadoop)中非常重要的一环，这个优化又涉及两个方面的内容。计算方面：Hadoop总会优先将任务分配给空闲的机器，使所有的任务能公平地分享系统资源。I/O方面：Hadoop会尽量将Map任务分配给InputSplit所在的机器，以减少网络I/O的消耗。

2) 数据预处理与InputSplit的大小

MapReduce任务擅长处理少量的[**大数据**](http://lib.csdn.net/base/hadoop)，而在处理大量的小数据时，MapReduce的性能就会逊色很多。因此在提交MapReduce任务前可以先对数据进行一次预处理，将数据合并以提高MapReduce任务的执行效率，这个办法往往很有效。如果这还不行，可以参考Map任务的运行时间，当一个Map任务只需要运行几秒就可以结束时，就需要考虑是否应该给它分配更多的数据。通常而言，一个Map任务的运行时间在一分钟左右比较合适，可以通过设置Map的输入数据大小来调节Map的运行时间。在FileInputFormat中（除了CombineFileInputFormat），Hadoop会在处理每个Block后将其作为一个InputSplit，因此合理地设置block块大小是很重要的调节方式。除此之外，也可以通过合理地设置Map任务的数量来调节Map任务的数据输入。

3) Map和Reduce任务的数量

合理地设置Map任务与Reduce任务的数量对提高MapReduce任务的效率是非常重要的。默认的设置往往不能很好地体现出MapReduce任务的需求，不过，设置它们的数量也要有一定的实践经验。

首先要定义两个概念—Map/Reduce任务槽。Map/Reduce任务槽就是这个集群能够同时运行的Map/Reduce任务的最大数量。比如，在一个具有1200台机器的集群中，设置每台机器最多可以同时运行10个Map任务，5个Reduce任务。那么这个集群的Map任务槽就是12000，Reduce任务槽是6000。任务槽可以帮助对任务调度进行设置。

设置MapReduce任务的Map数量主要参考的是Map的运行时间，设置Reduce任务的数量就只需要参考任务槽的设置即可。一般来说，Reduce任务的数量应该是Reduce任务槽的0.95倍或是1.75倍，这是基于不同的考虑来决定的。当Reduce任务的数量是任务槽的0.95倍时，如果一个Reduce任务失败，Hadoop可以很快地找到一台空闲的机器重新执行这个任务。当Reduce任务的数量是任务槽的1.75倍时，执行速度快的机器可以获得更多的Reduce任务，因此可以使负载更加均衡，以提高任务的处理速度。

4) Combine函数

Combine函数是用于本地合并数据的函数。在有些情况下，Map函数产生的中间数据会有很多是重复的，比如在一个简单的WordCount程序中，因为词频是接近与一个zipf分布的，每个Map任务可能会产生成千上万个<the, 1>记录，若将这些记录一一传送给Reduce任务是很耗时的。所以，MapReduce框架运行用户写的combine函数用于本地合并，这会大大减少网络I/O操作的消耗。此时就可以利用combine函数先计算出在这个Block中单词the的个数。合理地设计combine函数会有效地减少网络传输的数据量，提高MapReduce的效率。

在MapReduce程序中使用combine很简单，只需在程序中添加如下内容：

job.setCombinerClass(combine.class);

在WordCount程序中，可以指定Reduce类为combine函数，具体如下：

job.setCombinerClass(Reduce.class);

5) 压缩

编写MapReduce程序时，可以选择对Map的输出和最终的输出结果进行压缩（同时可以选择压缩方式）。在一些情况下，Map的中间输出可能会很大，对其进行压缩可以有效地减少网络上的数据传输量。对最终结果的压缩虽然会减少数据写HDFS的时间，但是也会对读取产生一定的影响，因此要根据实际情况来选择（第7章中提供了一个小实验来验证压缩的效果）。

6) 自定义comparator

在Hadoop中，可以自定义数据类型以实现更复杂的目的，比如，当读者想实现k-means[**算法**](http://lib.csdn.net/base/datastructure)（一个基础的聚类算法）时可以定义k个整数的集合。自定义Hadoop数据类型时，推荐自定义comparator来实现数据的二进制比较，这样可以省去数据序列化和反序列化的时间，提高程序的运行效率

4、shuffle机制

        在Hadoop这样的集群环境中，大部分map task与reduce task的执行是在不同的节点上。当然很多情况下Reduce执行时需要跨节点去拉取其它节点上的map task结果。如果集群正在运行的job有很多，那么task的正常执行对集群内部的网络资源消耗会很严重。这种网络消耗是正常的，我们不能限制，能做的就是最大化地减少不必要的消耗。还有在节点内，相比于内存，磁盘IO对job完成时间的影响也是可观的。从最基本的要求来说，我们对Shuffle过程的期望可以有：

完整地从map task端拉取数据到reduce 端。

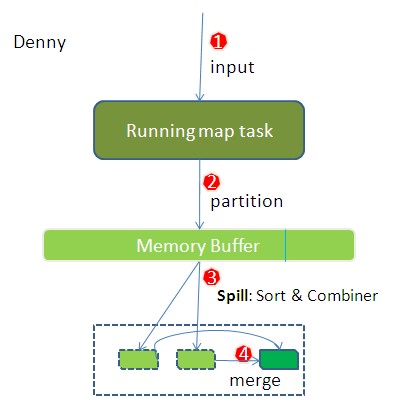
在跨节点拉取数据时，尽可能地减少对带宽的不必要消耗。

减少磁盘IO对task执行的影响。

        OK，看到这里时，大家可以先停下来想想，如果是自己来设计这段Shuffle过程，那么你的设计目标是什么。我想能优化的地方主要在于减少拉取数据的量及尽量使用内存而不是磁盘。

        我的分析是基于Hadoop0.21.0的源码，如果与你所认识的Shuffle过程有差别，不吝指出。我会以WordCount为例，并假设它有8个map task和3个reduce task。从上图看出，Shuffle过程横跨map与reduce两端，所以下面我也会分两部分来展开。

        先看看map端的情况，如下图：



        上图可能是某个map task的运行情况。拿它与官方图的左半边比较，会发现很多不一致。官方图没有清楚地说明partition， sort与combiner到底作用在哪个阶段。我画了这张图，希望让大家清晰地了解从map数据输入到map端所有数据准备好的全过程。

        整个流程我分了四步。简单些可以这样说，每个map task都有一个内存缓冲区，存储着map的输出结果，当缓冲区快满的时候需要将缓冲区的数据以一个临时文件的方式存放到磁盘，当整个map task结束后再对磁盘中这个map task产生的所有临时文件做合并，生成最终的正式输出文件，然后等待reduce task来拉数据。

        当然这里的每一步都可能包含着多个步骤与细节，下面我对细节来一一说明：

**1**）在map task执行时，它的输入数据来源于HDFS的block，当然在MapReduce概念中，map task只读取split。Split与block的对应关系可能是多对一，默认是一对一。在WordCount例子里，假设map的输入数据都是像“aaa”这样的字符串。

**2**）在经过mapper的运行后，我们得知mapper的输出是这样一个key/value对： key是“aaa”， value是数值1。因为当前map端只做加1的操作，在reduce task里才去合并结果集。前面我们知道这个job有3个reduce task，到底当前的“aaa”应该交由哪个reduce去做呢，是需要现在决定的。

        MapReduce提供Partitioner接口，它的作用就是根据key或value及reduce的数量来决定当前的这对输出数据最终应该交由哪个reduce task处理。默认对key hash后再以reduce task数量取模。默认的取模方式只是为了平均reduce的处理能力，如果用户自己对Partitioner有需求，可以订制并设置到job上。

        在我们的例子中，“aaa”经过Partitioner后返回0，也就是这对值应当交由第一个reducer来处理。接下来，需要将数据写入内存缓冲区中，缓冲区的作用是批量收集map结果，减少磁盘IO的影响。我们的key/value对以及Partition的结果都会被写入缓冲区。当然写入之前，key与value值都会被序列化成字节数组。

        整个内存缓冲区就是一个字节数组，它的字节索引及key/value存储结构我没有研究过。如果有朋友对它有研究，那么请大致描述下它的细节吧。

**3**） 这个内存缓冲区是有大小限制的，默认是100MB。当map task的输出结果很多时，就可能会撑爆内存，所以需要在一定条件下将缓冲区中的数据临时写入磁盘，然后重新利用这块缓冲区。这个从内存往磁盘写数据的过程被称为Spill，中文可译为溢写，字面意思很直观。这个溢写是由单独线程来完成，不影响往缓冲区写map结果的线程。溢写线程启动时不应该阻止map的结果输出，所以整个缓冲区有个溢写的比例spill.percent。这个比例默认是0.8，也就是当缓冲区的数据已经达到阈值（buffer size \* spill percent = 100MB \* 0.8 = 80MB），溢写线程启动，锁定这80MB的内存，执行溢写过程。Map task的输出结果还可以往剩下的20MB内存中写，互不影响。

        当溢写线程启动后，需要对这80MB空间内的key做排序(Sort)。排序是MapReduce模型默认的行为，这里的排序也是对序列化的字节做的排序。

        在这里我们可以想想，因为map task的输出是需要发送到不同的reduce端去，而内存缓冲区没有对将发送到相同reduce端的数据做合并，那么这种合并应该是体现是磁盘文件中的。从官方图上也可以看到写到磁盘中的溢写文件是对不同的reduce端的数值做过合并。所以溢写过程一个很重要的细节在于，如果有很多个key/value对需要发送到某个reduce端去，那么需要将这些key/value值拼接到一块，减少与partition相关的索引记录。

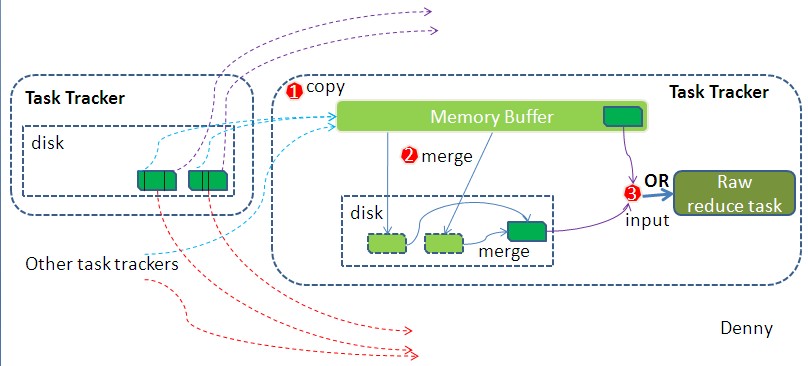
        在针对每个reduce端而合并数据时，有些数据可能像这样：“aaa”/1， “aaa”/1。对于WordCount例子，就是简单地统计单词出现的次数，如果在同一个map task的结果中有很多个像“aaa”一样出现多次的key，我们就应该把它们的值合并到一块，这个过程叫reduce也叫combine。但MapReduce的术语中，reduce只指reduce端执行从多个map task取数据做计算的过程。除reduce外，非正式地合并数据只能算做combine了。其实大家知道的，MapReduce中将Combiner等同于Reducer。

        如果client设置过Combiner，那么现在就是使用Combiner的时候了。将有相同key的key/value对的value加起来，减少溢写到磁盘的数据量。Combiner会优化MapReduce的中间结果，所以它在整个模型中会多次使用。那哪些场景才能使用Combiner呢？从这里分析，Combiner的输出是Reducer的输入，Combiner绝不能改变最终的计算结果。所以从我的想法来看，Combiner只应该用于那种Reduce的输入key/value与输出key/value类型完全一致，且不影响最终结果的场景。比如累加，最大值等。Combiner的使用一定得慎重，如果用好，它对job执行效率有帮助，反之会影响reduce的最终结果。

**4）** 每次溢写会在磁盘上生成一个溢写文件，如果map的输出结果真的很大，有多次这样的溢写发生，磁盘上相应的就会有多个溢写文件存在。当map task真正完成时，内存缓冲区中的数据也全部溢写到磁盘中形成一个溢写文件。最终磁盘中会至少有一个这样的溢写文件存在(如果map的输出结果很少，当map执行完成时，只会产生一个溢写文件)，因为最终的文件只有一个，所以需要将这些溢写文件归并到一起，这个过程就叫做Merge。Merge是怎样的？如前面的例子，“aaa”从某个map task读取过来时值是5，从另外一个map 读取时值是8，因为它们有相同的key，所以得merge成group。什么是group。对于“aaa”就是像这样的：{“aaa”, [5, 8, 2, …]}，数组中的值就是从不同溢写文件中读取出来的，然后再把这些值加起来。请注意，因为merge是将多个溢写文件合并到一个文件，所以可能也有相同的key存在，在这个过程中如果client设置过Combiner，也会使用Combiner来合并相同的key。

        至此，map端的所有工作都已结束，最终生成的这个文件也存放在TaskTracker够得着的某个本地目录内。每个reduce task不断地通过RPC从JobTracker那里获取map task是否完成的信息，如果reduce task得到通知，获知某台TaskTracker上的map task执行完成，Shuffle的后半段过程开始启动。

        简单地说，reduce task在执行之前的工作就是不断地拉取当前job里每个map task的最终结果，然后对从不同地方拉取过来的数据不断地做merge，也最终形成一个文件作为reduce task的输入文件。见下图：



        如map 端的细节图，Shuffle在reduce端的过程也能用图上标明的三点来概括。当前reduce copy数据的前提是它要从JobTracker获得有哪些map task已执行结束，这段过程不表，有兴趣的朋友可以关注下。Reducer真正运行之前，所有的时间都是在拉取数据，做merge，且不断重复地在做。如前面的方式一样，下面我也分段地描述reduce 端的Shuffle细节：

**1**) Copy过程，简单地拉取数据。Reduce进程启动一些数据copy线程(Fetcher)，通过HTTP方式请求map task所在的TaskTracker获取map task的输出文件。因为map task早已结束，这些文件就归TaskTracker管理在本地磁盘中。

**2**) Merge阶段。这里的merge如map端的merge动作，只是数组中存放的是不同map端copy来的数值。Copy过来的数据会先放入内存缓冲区中，这里的缓冲区大小要比map端的更为灵活，它基于JVM的heap size设置，因为Shuffle阶段Reducer不运行，所以应该把绝大部分的内存都给Shuffle用。这里需要强调的是，merge有三种形式：1)内存到内存  2)内存到磁盘  3)磁盘到磁盘。默认情况下第一种形式不启用，让人比较困惑，是吧。当内存中的数据量到达一定阈值，就启动内存到磁盘的merge。与map 端类似，这也是溢写的过程，这个过程中如果你设置有Combiner，也是会启用的，然后在磁盘中生成了众多的溢写文件。第二种merge方式一直在运行，直到没有map端的数据时才结束，然后启动第三种磁盘到磁盘的merge方式生成最终的那个文件。

**3)** Reducer的输入文件。不断地merge后，最后会生成一个“最终文件”。为什么加引号？因为这个文件可能存在于磁盘上，也可能存在于内存中。对我们来说，当然希望它存放于内存中，直接作为Reducer的输入，但默认情况下，这个文件是存放于磁盘中的。至于怎样才能让这个文件出现在内存中，之后的[性能优化篇](http://langyu.iteye.com/blog/1341267)我再说。当Reducer的输入文件已定，整个Shuffle才最终结束。然后就是Reducer执行，把结果放到HDFS上。

5、MR的二次排序

6、MR的join操作

7、MR一次性读取多行的实现方法

8、MR运行中内存达到多少时，数据会flush到磁盘？

默认100M \* 0.8 即 io.sort.mb \* io.sort.spill.percent

六、Hive

优化问题

七、Spark:

1、Spark的组件有哪些？

Spark Core、 SparkStreaming 、 SparkSql、 MLib、 GraphX

2、Spark的优化问题

3、Spark可以运行在那几种集群上？

yarn（hadoop）集群 spark集群 Apache mesos集群

4、使用的Spark版本？ Spark2.0跟Spark1.x的区别？

5、简述RDD

八、java基础部分：

1、生产者消费者模型

2、hashmap 的底层结构

ArrayList的底层结构

3、线程池的种类及机制

4、List Map Set分别继承的什么接口

九、redis

1、redis使用的版本

2、集群问题

3、redis在内存满了的时候是否能自动持久化？

默认不自动持久化redis的持久化有rdb和aof两种。

rdb是记录一段时间内的操作，一盘的配置是一段时间内操作超过多少次就持久化。

aof可以实现每次操作都持久化。

这里我们使用aof。

配置方式，打开redis的配置文件。找到appendonly。默认是appendonly no。改成appendonly yes。

再找到appendfsync

默认是：

# appendfsync always   #每次收到写命令就立即强制写入磁盘，最慢的，但是保证完全的持久化，不推荐使用

appendfsync everysec     #每秒钟强制写入磁盘一次，在性能和持久化方面做了很好的折中，推荐

# appendfsync no    #完全依赖os，性能最好,持久化没保证

默认每秒持久化满足我的需求。

其实改下appendonly 就ok了。

十、yarn原理

**1、MRv2在Yarn上执行流程：**

MR JobClient向resourceManager(AsM)提交一个job

AsM向Scheduler请求一个供MR AM运行的container，然后启动它

  MR AM启动起来后向AsM注册

  MR JobClient向AsM获取到MR AM相关的信息，然后直接与MR AM进行通信

  MR AM计算splits并为所有的map构造资源请求

  MR AM做一些必要的MR OutputCommitter的准备工作

  MR AM向RM(Scheduler)发起资源请求，得到一组供map/reduce task运行的container，然后与NM一起对每一个container执行一些必要的任务，包括资源本地化等

  MR AM 监视运行着的task 直到完成，当task失败时，申请新的container运行失败的task

  当每个map/reduce task完成后，MR AM运行MR OutputCommitter的cleanup 代码，也就是进行一些收尾工作

  当所有的map/reduce完成后，MR AM运行OutputCommitter的必要的job commit或者abort APIs

  MR AM退出。

十一、java部分

1、JVM知道吗？还有内存，新生代、老年代、永久代、minor GC、major GC做什么的

2、多线程相关的比如具体怎么使用多线程的，还有线程池（列举几种，每种的优缺点自己主动描述），对线程池调优，涉及到某些参数，这个没回答上来。。。。直接说当时没用到这个，只是设置线程池固定大小。。。

3、volatile

4、HashMap底层结构，ArrayList底层结构，ArrayList与LinkedList的不同

十二、架构设计部分：

1、开放性设计：设计一套架构 ， 将财务系统中的各个关系型表中的数据迁移到大数据平台，注意不是简单的迁移，中间会涉及字段的合并或运算