浅谈Spark应用程序的性能调优

5

云计算 (http://www.csdn.net/tag/云计算/news) 大数据 (http://www.csdn.net/tag/大数据/news)

Spark (http://www.csdn.net/tag/Spark/news) 调优 (http://www.csdn.net/tag/调优/news)

Spark是基于内存的分布式计算引擎，以处理的高效和稳定著称。然而在实际的应用开发过程中，开发者还是会遇到种种问题，其中一大类就是和性能相关。在本文中，笔者将结合自身实践，谈谈如何尽可能地提高应用程序性能。

分布式计算引擎在调优方面有四个主要关注方向，分别是CPU、内存、网络开销和I/O，其具体调优目标如下：

1. 提高CPU利用率。
2. 避免OOM。
3. 降低网络开销。
4. 减少I/O操作。

第1章 数据倾斜

数据倾斜意味着某一个或某几个Partition中的数据量特别的大，这意味着完成针对这几个Partition的计算需要耗费相当长的时间。

如果大量数据集中到某一个Partition，那么这个Partition在计算的时候就会成为瓶颈。图1是Spark应用程序执行并发的示意图，在Spark中，同一个应用程序的不同Stage是串行执行的，而同一Stage中的不同Task可以并发执行，Task数目由Partition数来决定，如果某一个Partition的数据量特别大，则相应的task完成时间会特别长，由此导致接下来的Stage无法开始，整个Job完成的时间就会非常长。

要避免数据倾斜的出现，一种方法就是选择合适的key，或者是自己定义相关的partitioner。在Spark中Block使用了ByteBuffer来存储数据，而ByteBuffer能够存储的最大数据量不超过2GB。如果某一个key有大量的数据，那么在调用cache或persist函数时就会碰到spark-1476这个异常。

下面列出的这些API会导致Shuffle操作，是数据倾斜可能发生的关键点所在

1. groupByKey
2. reduceByKey
3. aggregateByKey
4. sortByKey
5. join
6. cogroup
7. cartesian
8. coalesce
9. repartition
10. repartitionAndSortWithinPartitions

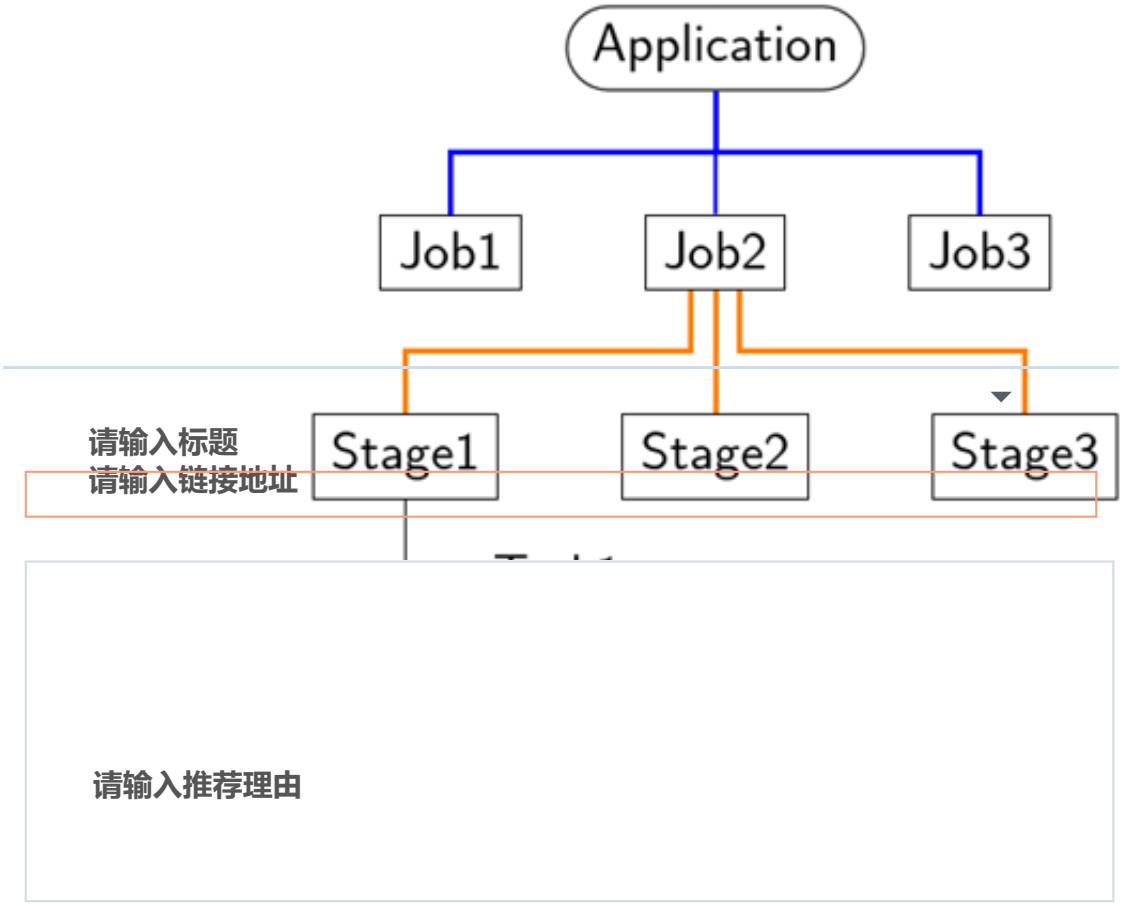

图1: Spark任务并发模型

```
  def rdd: RDD[T]
}

// TODO View bounds are deprecated, should use context bounds
// Might need to change ClassManifest for ClassTag in spark 1.0.0
case class DemoPairRDD[K <% Ordered[K] : ClassManifest, V: ClassManifest](
  rdd: RDD[(K, V)]) extends RDDWrapper[(K, V)] {
  // Here we use a single Long to try to ensure the sort is balanced,
  // but for really large dataset, we may want to consider
  // using a tuple of many Longs or even a GUID
  def sortByKeyGrouped(numPartitions: Int): RDD[(K, V)] =
    rdd.map(kv => ((kv._1, Random.nextLong()), kv._2)).sortByKey()
    .grouped(numPartitions).map(t => (t._1._1, t._2))
}

case class DemoRDD[T: ClassManifest](rdd: RDD[T]) extends RDDWrapper[T] {
  def grouped(size: Int): RDD[T] = {
    // TODO Version where withIndex is cached
    val withIndex = rdd.mapPartitions(_.zipWithIndex)

    val startValues =
      withIndex.mapPartitionsWithIndex((i, iter) =>
        Iterator((i, iter.toIterable.last))).toArray().toList
      .sortBy(_._1).map(_._2._2.toLong).scan(-1L)(_ + _).map(_ + 1L)

    withIndex.mapPartitionsWithIndex((i, iter) => iter.map {
      case (value, index) => (startValues(i) + index.toLong, value)
    })
    .partitionBy(new Partitioner {
      def numPartitions: Int = size
      def getPartition(key: Any): Int =
        (key.asInstanceOf[Long] * numPartitions.toLong / startValues.last).toInt
    })
    .map(_._2)
  }
}
```


定义隐式的转换

```
  implicit def toDemoRDD[T: ClassManifest](rdd: RDD[T]): DemoRDD[T] =
    new DemoRDD[T](rdd)
  implicit def toDemoPairRDD[K <% Ordered[K] : ClassManifest, V: ClassManifest](
    rdd: RDD[(K, V)]): DemoPairRDD[K, V] = DemoPairRDD(rdd)
  implicit def toRDD[T](rdd: RDDWrapper[T]): RDD[T] = rdd.rdd
}
```

在spark-shell中就可以使用了

```
import RDDConversions._

yourRdd.grouped(5)
```

第2章 减少网络通信开销

Spark的Shuffle过程非常消耗资源，Shuffle过程意味着在相应的计算节点，要先将计算结果存储到磁盘，后续的Stage需要将上一个Stage的结果再次读入。数据的写入和读取意味着Disk I/O操作，与内存操作相比，Disk I/O操作是非常低效的。

使用iostat来查看disk i/o的使用情况，disk i/o操作频繁一般会伴随着cpu load很高。

如果数据和计算节点都在同一台机器上，那么可以避免网络开销，否则还要加上相应的网络开销。 使用iftop来查看网络带宽使用情况，看哪几个节点之间有大量的网络传输。
图2是Spark节点间数据传输的示意图，Spark Task的计算函数是通过Akka通道由Driver发送到Executor上，而Shuffle的数据则是通过Netty网络接口来实现。由于Akka通道中参数spark.akka.framesize决定了能够传输消息的最大值，所以应该避免在Spark Task中引入超大的局部变量。

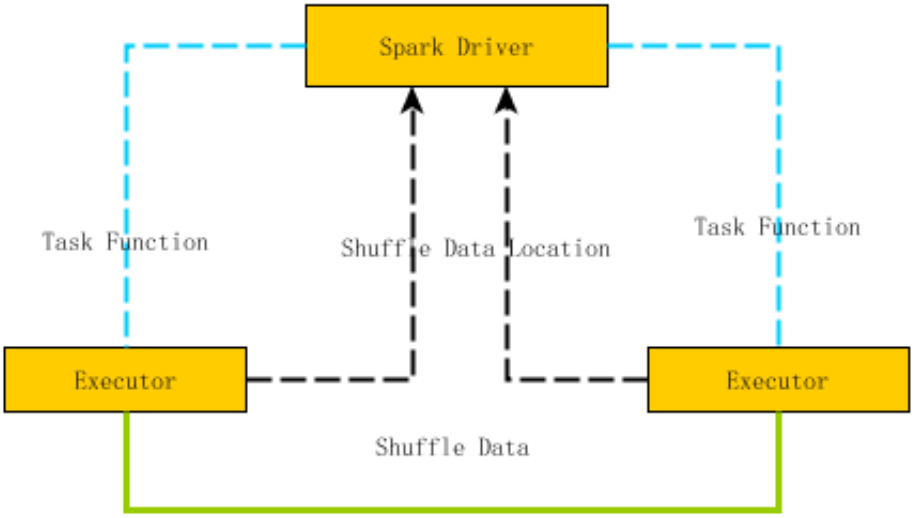

图2: Spark节点间的数据传输

第1节 选择合适的并发数

为了提高Spark应用程序的效率，尽可能的提升CPU的利用率。并发数应该是可用CPU物理核数的两倍。在这里，并发数过低，CPU得不到充分的利用，并发数过大，由于spark是每一个task都要分发到计算结点，所以任务启动的开销会上升。

并发数的修改，通过配置参数来改变spark.default.parallelism，如果是sql的话，可能通过修改spark.sql.shuffle.partitions来修改。

第1项 Repartition vs. Coalesce

repartition和coalesce都能实现数据分区的动态调整，但需要注意的是repartition会导致shuffle操作，而coalesce不会。

第2节 reduceByKey vs. groupBy

groupBy操作应该尽可能的避免，第一是有可能造成大量的网络开销，第二是可能导致OOM。以WordCount为例来演示reduceByKey和groupBy的差异

```
reduceByKey
      sc.textFile(“README.md”).map(l=>l.split(“,”)).map(w=>(w,1)).reduceByKey(_ + _)
```

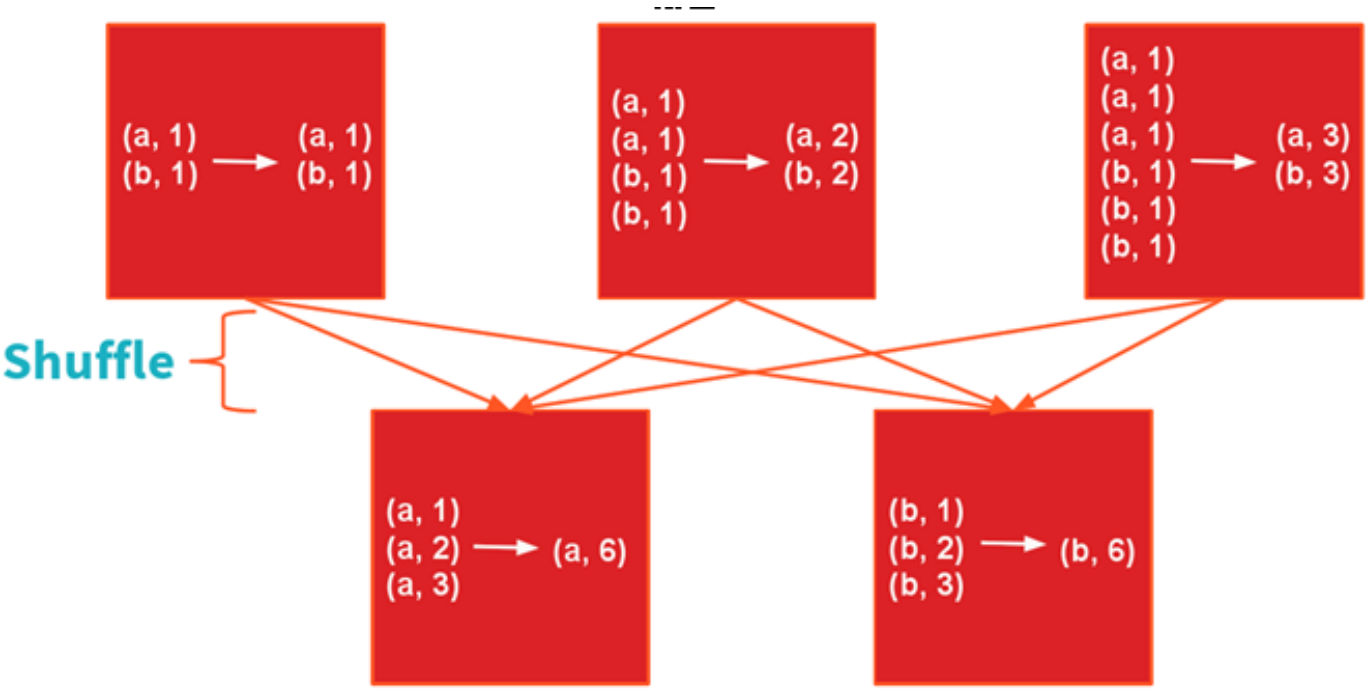

图3：reduceByKey的Shuffle过程

Shuffle过程如图2所示

```
groupByKey
        sc.textFile(“README.md”).map(l=>l.split(“,”)).map(w=>(w,1)).groupByKey.map(r=>(r._1,r._2.sum))
```

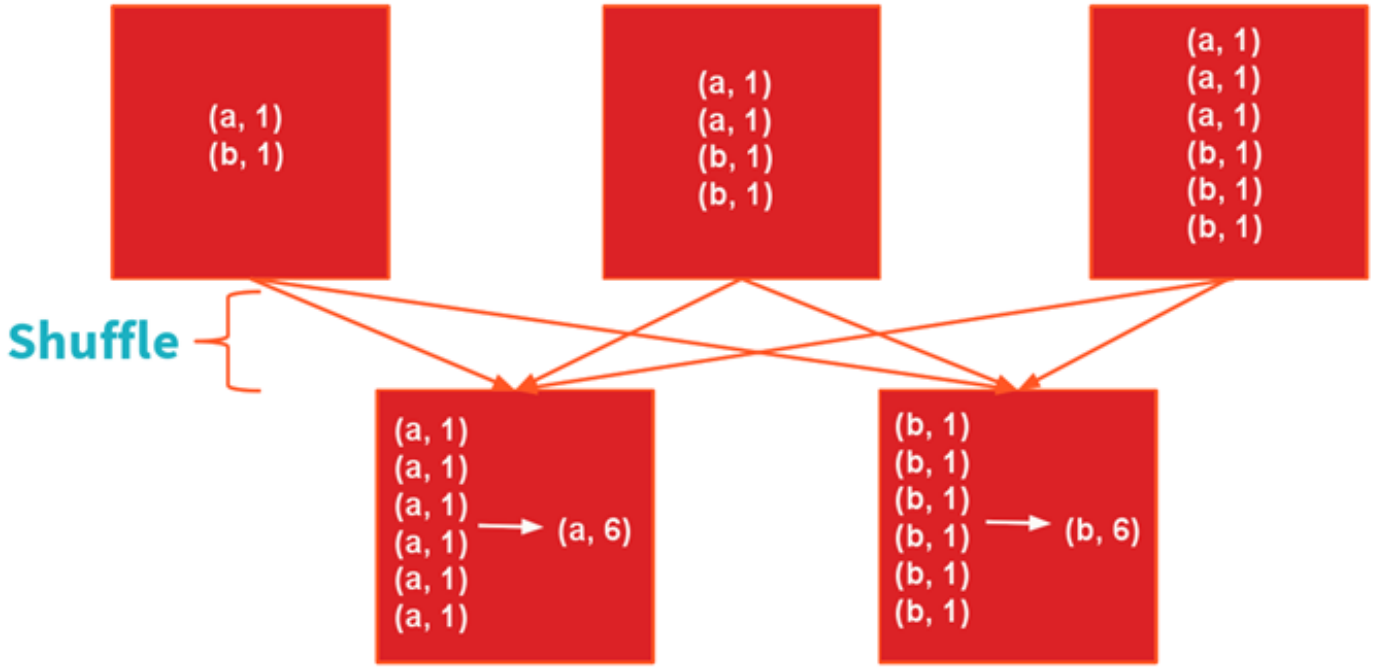

图4：groupByKey的Shuffle过程

建议: 尽可能使用reduceByKey, aggregateByKey, foldByKey和combineByKey
假设有一RDD如下所示，求每个key的均值

val data = sc.parallelize(List((0, 2.), (0, 4.), (1, 0.), (1, 10.), (1, 20.)))

方法一：reduceByKey

```
data.map(r=>(r._1, (r.2,1))).reduceByKey((a,b)=>(a._1 + b._1, a._2 + b._2)).ma
p(r=>(r._1,(r._2._1/r._2._2)).foreach(println)
```

方法二：combineByKey

```
data.combineByKey(value=>(value,1),
    (x:(Double, Int), value:Double)=> (x._1+value, x._2 + 1),
    (x:(Double,Int), y:(Double, Int))=>(x._1 + y._1, x._2 + y._2))
```

第3节 BroadcastHashJoin vs. ShuffleHashJoin

在Join过程中，经常会遇到大表和小表的join. 为了提高效率可以使用BroadcastHashJoin, 预先将小表的内容广播到各个Executor, 这样将避免针对小表的Shuffle过程，从而极大的提高运行效率。

其实BroadCastHashJoin核心就是利用了BroadCast函数，如果理解清楚broadcast的优点，就能比较好的明白BroadcastHashJoin的优势所在。

以下是一个简单使用broadcast的示例程序。

```
val lst = 1 to 100 toList
val exampleRDD = sc.makeRDD(1 to 20 toSeq, 2)
val broadcastLst = sc.broadcast(lst)
exampleRDD.filter(i=>broadcastLst.valuecontains(i)).collect.foreach(println)
```

第4节 map vs. mapPartitions

有时需要将计算结果存储到外部数据库，势必会建立到外部数据库的连接。应该尽可能的让更多的元素共享同一个数据连接而不是每一个元素的处理时都去建立数据库连接。
在这种情况下，mapPartitions和foreachPartitons将比map操作高效的多。

第5节 数据就地读取

移动计算的开销远远低于移动数据的开销。

Spark中每个Task都需要相应的输入数据，因此输入数据的位置对于Task的性能变得很重要。按照数据获取的速度来区分，由快到慢分别是：

1.PROCESS_LOCAL
2.NODE_LOCAL
3.RACK_LOCAL

Spark在Task执行的时候会尽优先考虑最快的数据获取方式，如果想尽可能的在更多的机器上启动Task，那么可以通过调低spark.locality.wait的值来实现, 默认值是3s。

除了HDFS，Spark能够支持的数据源越来越多，如Cassandra, HBase,MongoDB等知名的NoSQL数据库，随着Elasticsearch的日渐兴起，spark和elasticsearch组合起来提供高速的查询解决方案也成为一种有益的尝试。

上述提到的外部数据源面临的一个相同问题就是如何让spark快速读取其中的数据， 尽可能的将计算结点和数据结点部署在一起是达到该目标的基本方法，比如在部署Hadoop集群的时候，可以将HDFS的DataNode和Spark Worker共享一台机器。

以cassandra为例,如果Spark的部署和Cassandra的机器有部分重叠，那么在读取Cassandra中数据的时候，通过调低spark.locality.wait就可以在没有部署Cassandra的机器上启动Spark Task。

对于Cassandra, 可以在部署Cassandra的机器上部署Spark Worker，需要注意的是Cassandra的compaction操作会极大的消耗CPU，因此在为Spark Worker配置CPU核数时，需要将这些因素综合在一起进行考虑。

这一部分的代码逻辑可以参考源码TaskSetManager::addPendingTask

```
private def addPendingTask(index: Int, readding: Boolean = false) {
  // Utility method that adds `index` to a list only if readding=false or it's no
t already there
  def addTo(list: ArrayBuffer[Int]) {
    if (!readding || !list.contains(index)) {
      list += index
    }
  }

  for (loc <- tasks(index).preferredLocations) {
    loc match {
      case e: ExecutorCacheTaskLocation =>
        addTo(pendingTasksForExecutor.getOrElseUpdate(e.executorId, new ArrayBuff
er))
      case e: HDFSCacheTaskLocation => {
        val exe = sched.getExecutorsAliveOnHost(loc.host)
        exe match {
          case Some(set) => {
            for (e <- set) {
              addTo(pendingTasksForExecutor.getOrElseUpdate(e, new ArrayBuffer))
            }
            logInfo(s"Pending task $index has a cached location at ${e.host} " +
              ", where there are executors " + set.mkString(","))
          }
           case None => logDebug(s"Pending task $index has a cached location at
${e.host} " +
              ", but there are no executors alive there.")
        }
      }
      case _ => Unit
    }
    addTo(pendingTasksForHost.getOrElseUpdate(loc.host, new ArrayBuffer))
    for (rack <- sched.getRackForHost(loc.host)) {
      addTo(pendingTasksForRack.getOrElseUpdate(rack, new ArrayBuffer))
    }
  }

  if (tasks(index).preferredLocations == Nil) {
    addTo(pendingTasksWithNoPrefs)
  }

  if (!readding) {
    allPendingTasks += index  // No point scanning this whole list to find the ol
d task there
  }
}
```

如果准备让spark支持新的存储源，进而开发相应的RDD，与位置相关的部分就是自定义getPreferredLocations函数，以elasticsearch-hadoop中的EsRDD为例，其代码实现如下。

```
override def getPreferredLocations(split: Partition): Seq[String] = {
  val esSplit = split.asInstanceOf[EsPartition]
  val ip = esSplit.esPartition.nodeIp
  if (ip != null) Seq(ip) else Nil
}
```

第6节 序列化

使用好的序列化算法能够提高运行速度，同时能够减少内存的使用。

Spark在Shuffle的时候要将数据先存储到磁盘中，存储的内容是经过序列化的。序列化的过程牵涉到两大基本考虑的因素，一是序列化的速度，二是序列化后内容所占用的大小。

kryoSerializer与默认的javaSerializer相比，在序列化速度和序列化结果的大小方面都具有极大的优势。所以建议在应用程序配置中使用KryoSerializer.

```
spark.serializer  org.apache.spark.serializer.KryoSerializer
```

默认的cache没有对缓存的对象进行序列化，使用的StorageLevel是MEMORY_ONLY,这意味着要占用比较大的内存。可以通过指定persist中的参数来对缓存内容进行序列化。

```
exampleRDD.persist(MEMORY_ONLY_SER)
```

需要特别指出的是persist函数是等到job执行的时候才会将数据缓存起来，属于延迟执行; 而unpersist函数则是立即执行，缓存会被立即清除。

> 作者简介：许鹏， 《Apache Spark源码剖析》作者，关注于大数据实时搜索和实时流数据处理，对elasticsearch, storm及drools多有研究，现就职于携程。

(http://geek.csdn.net/user/publishlist/karamos)

karamos (http://geek.csdn.net/user/publishlist/karamos)

发布于 Spark (http://geek.csdn.net/forum/48) 7小时前

评论

已有0条评论 最新