

通过分区(Partitioning)提高Spark的运行性能

在Sortable公司，很多数据处理的工作都是使用Spark完成的。在使用Spark的过程中他们发现了一个能够提高Spark job性能的一个技巧，也就是修改数据的分区数，本文将举个例子并详细地介绍如何做到的。

查找质数

比如我们需要从2到2000000之间寻找所有的质数。我们很自然地会想到先找到所有的非质数，剩下的所有数字就是我们要找的质数。

我们首先遍历2到2000000之间的每个数，然后找到这些数的所有小于或等于2000000的倍数，在计算的结果中可能会有许多重复的数据（比如6同时是2和3的倍数）但是这并没有啥影响。

我们在Spark shell中计算：

Welcome to

```

  ____
 /  _/  _  _  _/  /  _
 _W W/  _ W/  _ ` /  /  ' /
 /  _/  . _/W _ / / / /W _W version 1.6.1
 /  _/

```

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_45)

Type in expressions to have them evaluated.

Type :help for more information.

Spark context available as sc.

SQL context available as sqlContext.

```
scala> val n = 2000000
```

```
n: Int = 2000000
```

```
scala> val composite = sc.parallelize(2 to n, 8).map(x => (x, (2 to (n / x))))
.flatMap(kv => kv._2.map(_ * kv._1))
```

```
composite: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[2] at flatMap at <console>:29
```

```
scala>
```

```
scala> val prime = sc.parallelize(2 to n, 8).subtract(composite)
```

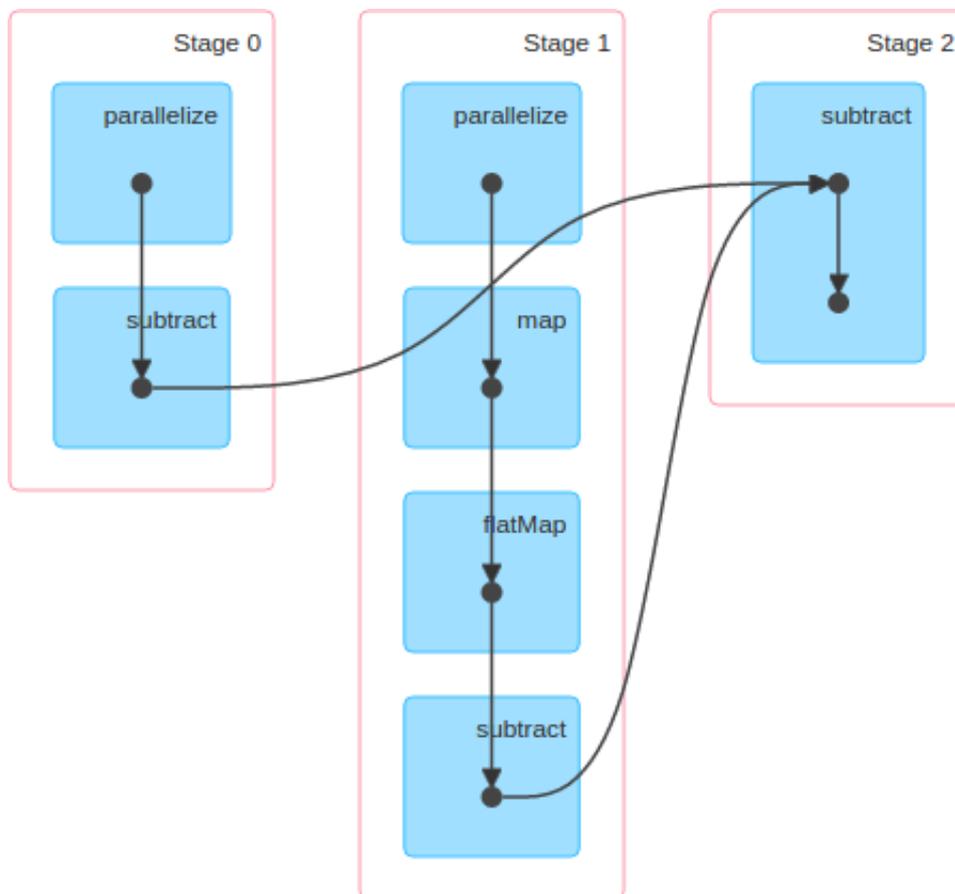
```
prime: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[7] at subtract at &lt;console&gt;;31
```

```
scala> prime.collect()
```

```
res0: Array[Int] = Array(563249, 17, 281609, 840761, 1126513, 1958993, 840713, 1959017, 41,
```

281641, 1681513, 1126441, 73, 1126457, 89, 840817, 97, 1408009, 113, 137, 1408241, 563377, 1126649, 281737, 281777, 840841, 1408217, 1681649, 281761, 1408201, 1959161, 1408177, 840929, 563449, 1126561, 193, 1126577, 1126537, 1959073, 563417, 233, 281849, 1126553, 563401, 281833, 241, 563489, 281, 281857, 257, 1959241, 313, 841081, 337, 1408289, 563561, 281921, 353, 1681721, 409, 281993, 401, 1126897, 282001, 1126889, 1959361, 1681873, 563593, 433, 841097, 1959401, 1408417, 1959313, 1681817, 457, 841193, 449, 563657, 282089, 282097, 1408409, 1408601, 1959521, 1682017, 841241, 1408577, 569, 1408633, 521, 841273, 1127033, 841289, 617, 1408529, 1959457, 563777, 841297, 1959473, 577, 593, 563809, 601,...

答案看起来是可靠的，但是我们来看看这个程序的性能。如果我们到Spark UI里面看的话可以发现Spark在整个计算过程中使用了3个stages，下图就是UI中这个计算过程的DAG(Directed Acyclic Graph)可视化图，其中展示了DAG图中不同的RDD计算。



在Spark中，只要job需要在分区之间进行数据交互，那么一个新的stage将会产生（如果使用Spark术语的话，分区之间的数据交互其实就是shuffle）。Spark stage中每个分区将会起一个task进行计算，而这些task负责将这个RDD分区的数据转化(transform)成另外一个RDD分区的数据。我们简单地看下Stage 0的task运行情况：

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Write Time	Shuffle Write Size / Records	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/05/05 14:42:29	0.5 s	34 ms	12 ms	1008.3 KB / 249999	
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/05/05 14:42:29	0.6 s	34 ms	9 ms	1008.9 KB / 250000	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/05/05 14:42:29	0.5 s	34 ms	12 ms	1008.8 KB / 250000	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/05/05 14:42:29	0.5 s	34 ms	15 ms	1008.6 KB / 250000	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/05/05 14:42:29	0.5 s	34 ms	10 ms	1008.9 KB / 250000	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/05/05 14:42:29	0.6 s	34 ms	14 ms	1008.3 KB / 250000	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/05/05 14:42:29	0.5 s	34 ms	13 ms	1008.9 KB / 250000	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/05/05 14:42:29	0.5 s	34 ms	12 ms	1008.6 KB / 250000	

上图中我们对Duration和Shuffle Write Size / Records两列非常感兴趣。sc.parallelize(2 to n, 8)已经生成了1999999 records，而这写记录均匀地分布到8个分区里面；每个task的计算几乎花费了相同的时间，所以这个stage是没问题的。

Stage 1是比较重要的stage，因为它运行了map和flatMap transformation，我们来看看它的运行情况：

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Write Time	Shuffle Write Size / Records	Errors
0	8	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/05/05 14:42:30	14 s	0.2 s	0.2 s	106.3 MB / 23640584	
1	9	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/05/05 14:42:30	1 s	0.1 s	22 ms	4.6 MB / 1019047	
2	10	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/05/05 14:42:30	0.8 s	0.1 s	5 ms	1683.9 KB / 416666	
3	11	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/05/05 14:42:30	0.6 s	0.1 s	4 ms	1008.9 KB / 250000	
4	12	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/05/05 14:42:30	0.4 s	96 ms		0.0 B / 0	
5	13	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/05/05 14:42:30	0.4 s	96 ms		0.0 B / 0	
6	14	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/05/05 14:42:30	0.3 s	96 ms		0.0 B / 0	
7	15	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/05/05 14:42:30	0.2 s	90 ms		0.0 B / 0	

从上图可以看出，这个stage运行的并不好，因为工作负载并没有均衡到所有的task中！93%的数据集中在一个task中，而这个task的计算花费了14s；另外一个比较慢的task花费了1s。然而我们提供了8个core用于计算，而其中的7个core在这13s内都在等待这个stage的完成。这对资源的利用非常不高效。



微信扫一扫，加关注

即可及时了解Spark、Hadoop或者Hbase等相关的文章

欢迎关注微信公共帐号: iteblog_hadoop

过往记忆博客 (<http://www.iteblog.com>)
专注于Hadoop、Spark、Flume、Hbase等技术的博客，欢迎关注。

Hadoop、Hive、Hbase、Flume等交流群：138615359和149892483

如果想及时了

解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

为什么会出现这种情况？

当我们运行`sc.parallelize(2 to n, 8)`语句的时候，Spark使用分区机制将数据很好地分成8个组。它最有可能使用的是`range partitioner`，也就是说2-250000被分到第一个分区；250001-500000分到第二个分区等等。然而我们的`map`函数将这些数转成`(key,value)`pairs，而`value`里面的数据大小变化很大（`key`比较小的时候，`value`的值就比较多，从而也比较大）。每个`value`都是一个`list`，里面存放着我们需要乘上`key`并小于2000000的倍数值，有一半以上的键值对（所有`key`大于1000000）的`value`是空的；而`key`等于2对应的`value`是最多的，包含了所有从2到1000000的数据！这就是为什么第一个分区拥有几乎所有的数据，它的计算花费了最多的时间；而最后四个分区几乎没有数据！

如何解决

我们可以将数据重新分区。通过对RDD调用`.repartition(numPartitions)`函数将会使Spark触发`shuffle`并且将数据分布到我们指定的分区数中，所以让我们尝试将这个加入到我们的代码中。

我们除了在`.map`和`.flatMap`函数之间加上`.repartition(8)`之外，其他的代码并不改变。我们的RDD现在同样拥有8个分区，但是现在的数据将会在这些分区重新分布，修改后的代码如下：

```
/**  
 * User: 过往记忆  
 * Date: 2016年6月24日
```

* Time: 下午21:16

* bolg: https://www.iteblog.com

* 本文地址 : https://www.iteblog.com/archives/1695.html

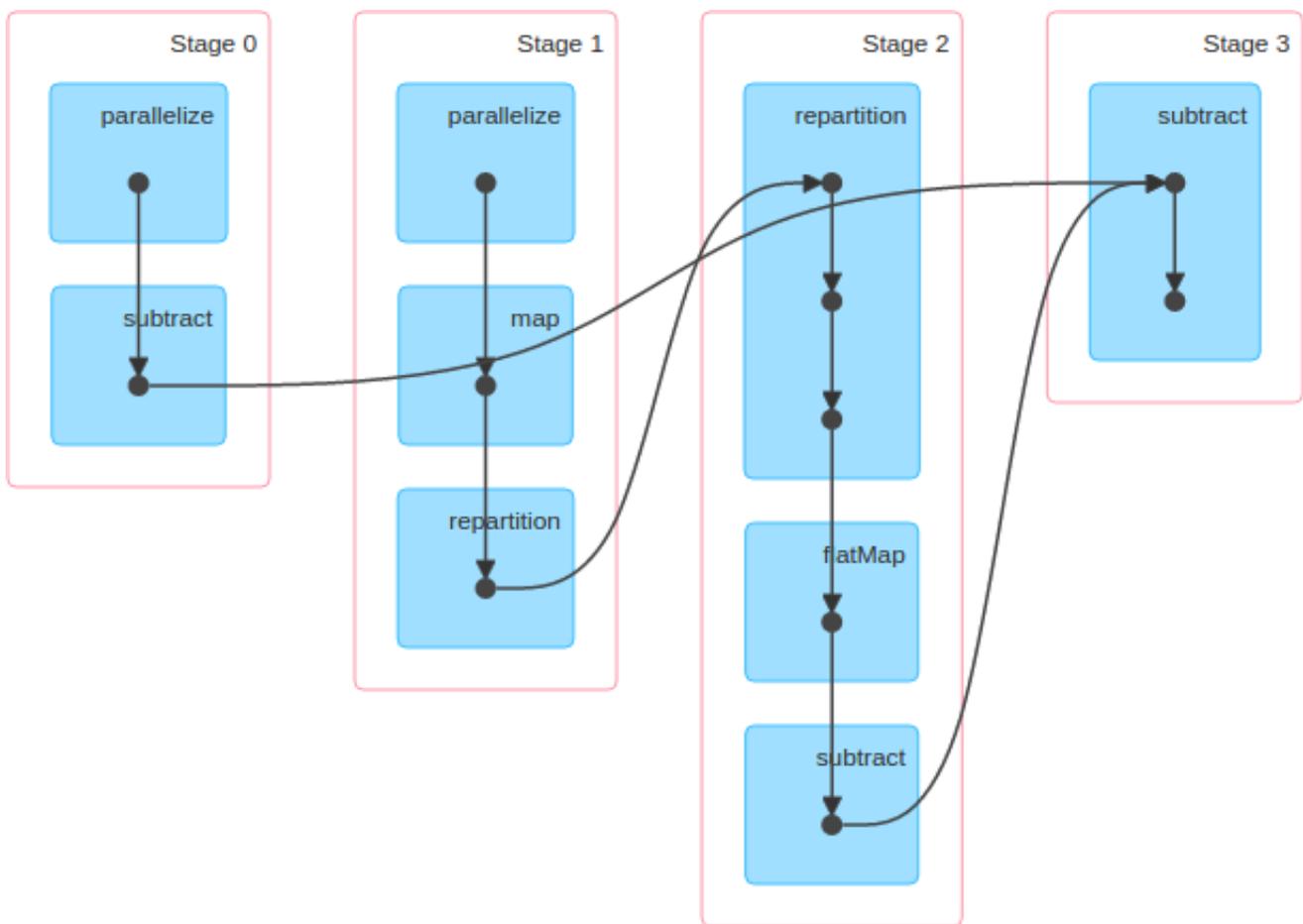
* 过往记忆博客, 专注于hadoop、hive、spark、shark、flume的技术博客, 大量的干货

* 过往记忆博客微信公共帐号 : iteblog_hadoop

*/

```
val composite = sc.parallelize(2 to n, 8).map(x => (x, (2 to (n / x))))).repartition(8).flatMap(kv => k  
v._2.map(_ * kv._1))
```

新的DAG可视化图看起来比之前更加复杂, 因为repartition操作会有shuffle操作, 所有增加了一个stage。



Stage 0和之前一样, 新的 Stage 1看起来和 Stage 0也很类似, 每个task大约都处理250000条记录, 并且花费1s的时间。 Stage 2是比较重要的stage, 下面是其截图:

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Shuffle Read Size / Records	Write Time	Shuffle Write Size / Records	Errors
0	16	0	SUCCESS	NODE_LOCAL	driver / localhost	2016/05/05 14:50:17	5 s	0.2 s	2.7 MB / 250000	42 ms	14.2 MB / 3242491	
1	17	0	SUCCESS	NODE_LOCAL	driver / localhost	2016/05/05 14:50:17	5 s	0.2 s	2.7 MB / 250000	44 ms	13.9 MB / 3114793	
2	18	0	SUCCESS	NODE_LOCAL	driver / localhost	2016/05/05 14:50:17	5 s	0.2 s	2.7 MB / 250000	36 ms	13.5 MB / 3023100	
3	19	0	SUCCESS	NODE_LOCAL	driver / localhost	2016/05/05 14:50:17	5 s	0.2 s	2.7 MB / 250000	38 ms	13.3 MB / 2952632	
4	20	0	SUCCESS	NODE_LOCAL	driver / localhost	2016/05/05 14:50:17	5 s	0.2 s	2.7 MB / 250000	56 ms	12.7 MB / 2895958	
5	21	0	SUCCESS	NODE_LOCAL	driver / localhost	2016/05/05 14:50:17	4 s	0.2 s	2.7 MB / 249999	53 ms	12.9 MB / 2848739	
6	22	0	SUCCESS	NODE_LOCAL	driver / localhost	2016/05/05 14:50:17	5 s	0.2 s	2.7 MB / 250000	41 ms	16.6 MB / 3808486	
7	23	0	SUCCESS	NODE_LOCAL	driver / localhost	2016/05/05 14:50:17	5 s	0.2 s	2.7 MB / 250000	36 ms	15.2 MB / 3440098	

从上图可以看出，现在的Stage 2比之前旧的Stage 1性能要好很多，这次Stage我们处理的数据和之前旧的Stage 1同样多，但是这次每个task花费的时间大概为5s，而且每个core得到了高效地使用。

两个版本的代码最后一个Stage大概都运行了6s，所以第一个版本的代码运行了大约 $0.5 + 14 + 6 = \sim 21s$ ；而对数据进行重新分布之后，这次运行的时间大约为 $0.5 + 1 + 5 + 6 = \sim 13s$ 。虽然说修改后的代码需要做一些额外的计算(重新分布数据)，但是这个修改却减少了总的运行时间，因为它使得我们可以更加高效地使用我们的资源。

当然，如果你的目标是寻找质数，有比这里介绍的更加高效的算法。但是本文仅仅是用来介绍考虑Spark数据的分布是多么地重要。增加.repartition函数将会增加Spark总体的工作，但好处可以显著大于成本

本文翻译自：[Improving Spark Performance With Partitioning](#)

本博客文章除特别声明，全部都是原创！
原创文章版权归过往记忆大数据（过往记忆）所有，未经许可不得转载。
本文链接: 【】（）