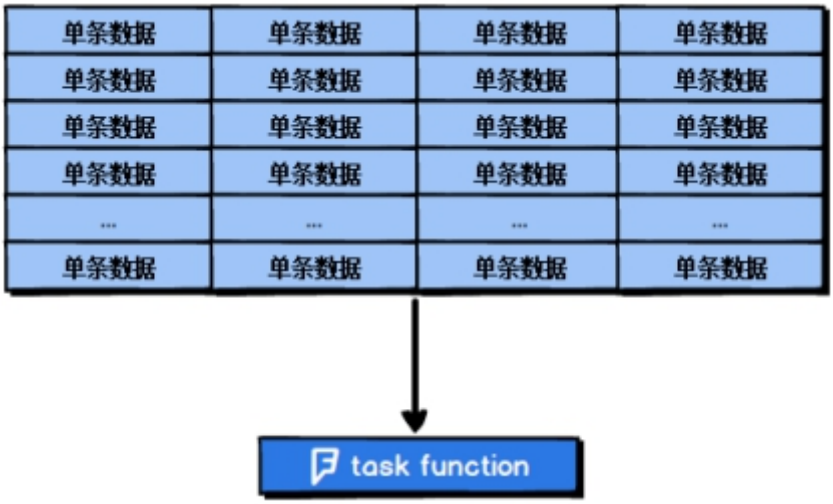
> 思考：如何提高意志力上限

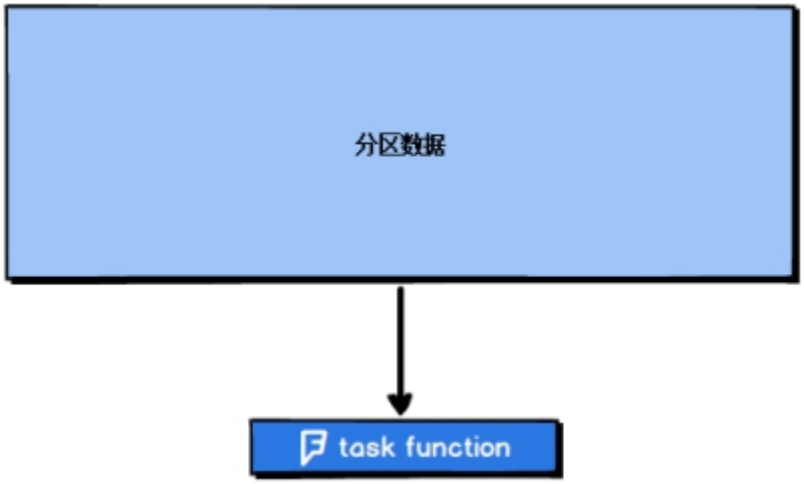
Spark之算子调优

调优一：mapPartitions

> 普通的map算子对RDD中的每一个元素进行操作，而mapPartitions算子对RDD中每一个分区进行操作。如果是普通的map算子，假设一个partition有1万条数据，那么map算子中的function要执行1万次，也就是对每个元素进行操作。



如果是mapPartition算子，由于一个task处理一个RDD的partition，那么一个task只会执行一次function，function一次接收所有的partition数据，效率比较高。



比如，当要把RDD中的所有数据通过JDBC写入数据，如果使用map算子，那么需要对RDD中的每一个元素都创建一个数据库连接，这样对资源的消耗很大，如果使用mapPartitions算子，那么针对**一个分区的数据，只需要建立一个数据库连接。**

mapPartitions算子也存在一些缺点：对于普通的map操作，一次处理一条数据，如果在处理了2000条数据后内存不足，那么可以将已经处理完的2000条数据从内存中垃圾回收掉；但是如果使用mapPartitions算子，但数据量非常大时，function一次处理一个分区的数据，***如果一旦内存不足，此时无法回收内存，就可能会OOM**，即内存溢出。

因此，mapPartitions算子适用于数据量不是特别大的时候，此时**使用mapPartitions算子对性能的提升效果还是不错的。**（当数据量很大的时候，一旦使用mapPartitions算子，就会直接OOM）

在项目中，应该首先**估算一下RDD的数据量**、每个partition的数据量，以及分配给每个Executor的内存资源，如果资源允许，可以考虑使用mapPartitions算子代替map。

调优二：foreachPartition优化数据库操作

在生产环境中，通常使用**foreachPartition算子来完成数据库的写**入，通过foreachPartition算子的特性，可以优化写数据库的性能。如果使用foreach算子完成数据库的操作，由于foreach算子是遍历RDD的每条数据，因此，每条数据都会建立一个数据库连接，这是对资源的极大浪费，因此，对于写数据库操作，我们应当使用foreachPartition算子。与mapPartitions算子非常相似，foreachPartition是将RDD的每个分区作为遍历对象，一次处理一个分区的数据，也就是说，如果涉及数据库的相关操作，一个分区的数据只需要创建一次数据库连接，如图2所示：

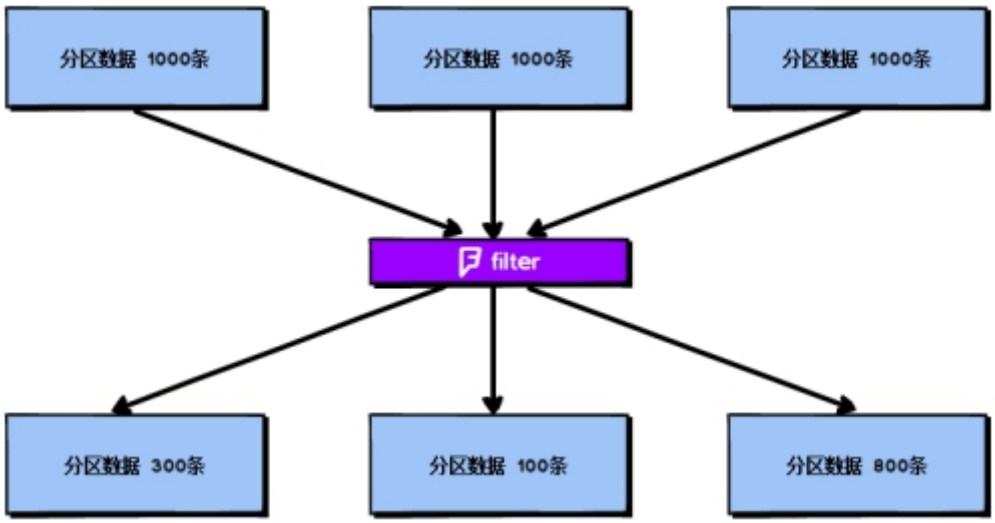
使用了foreachPartition算子后，可以获得以下的性能提升：

1. 对于我们写的function函数，一次处理一整个分区的数据；
2. 对于一个分区内的数据，创建唯一的数据库连接；
3. 只需要向数据库发送一次SQL语句和多组参数；

在生产环境中，全部都会使用foreachPartition算子完成数据库操作。foreachPartition算子存在一个问题，与mapPartitions算子类似，如果一个分区的数据量特别大，可能会造成OOM，即内存溢出。

调优三：filter与coalesce的配合使用

在Spark任务中我们经常会使用filter算子完成RDD中数据的过滤，在任务初始阶段，从各个分区中加载到的数据量是相近的，但是一旦进过filter过滤后，**每个分区的数据量有可能会存在较大差异**，如图所示：



根据图我们可以发现两个问题：

1. 每个partition的数据量变小了，如果还按照之前与partition相等的task个数去处理当前数据，有点浪费task的计算资源；

2. 每个partition的数据量不一样，会导致后面的每个task处理每个partition数据的时候，每个task要处理的数据量不同，这很有可能导致数据倾斜问题。

如图所示，第二个分区的数据过滤后只剩100条，而第三个分区的数据过滤后剩下800条，在相同的处理逻辑下，第二个分区对应的task处理的数据量与第三个分区对应的task处理的**数据量差距达到了8倍**，这也会导致运行速度可能存在数倍的差距，这也就是**数据倾斜**问题。

针对上述的两个问题，我们分别进行分析：

1. 针对第一个问题，既然分区的数据量变小了，我们希望可以对分区数据进行重新分配，比如将原来4个分区的数据转化到2个分区中，这样只需要用后面的两个task进行处理即可，避免了资源的浪费。

2. 针对第二个问题，解决方法和第一个问题的解决方法非常相似，对分区数据重新分配，让每个partition中的数据量差不多，这就避免了数据倾斜问题。

那么具体应该如何实现上面的解决思路？我们需要coalesce算子。

repartition与coalesce都可以用来进行重分区，其中repartition只是coalesce接口中shuffle为true的简易实现，coalesce默认情况下不进行shuffle，但是可以通过参数进行设置。

假设我们希望将原本的分区个数A通过重新分区变为B，那么有以下几种情况：

1. A > B（多数分区合并为少数分区）

① A与B相差值不大
此时使用coalesce即可，无需shuffle过程。

② A与B相差值很大
此时可以使用coalesce并且不启用shuffle过程，但是会导致合并过程性能低下，所以推荐设置coalesce的第二个参数为true，即启动shuffle过程。

2. A < B（少数分区分解为多数分区）

此时使用repartition即可，如果使用coalesce需要将shuffle设置为true，否则coalesce无效。

我们可以在filter操作之后，使用coalesce算子针对每个partition的数据量各不相同的情况，压缩partition的数量，而且让每个partition的数据量尽量均匀紧凑，以便于后面的task进行计算操作，在某种程度上能够在一定程度上提升性能。

注意：local模式是进程内模拟集群运行，已经对并行度和分区数量有了一定的内部优化，因此不用去设置并行度和分区数量。

调优四：repartition解决SparkSQL低并行度问题

问题场景

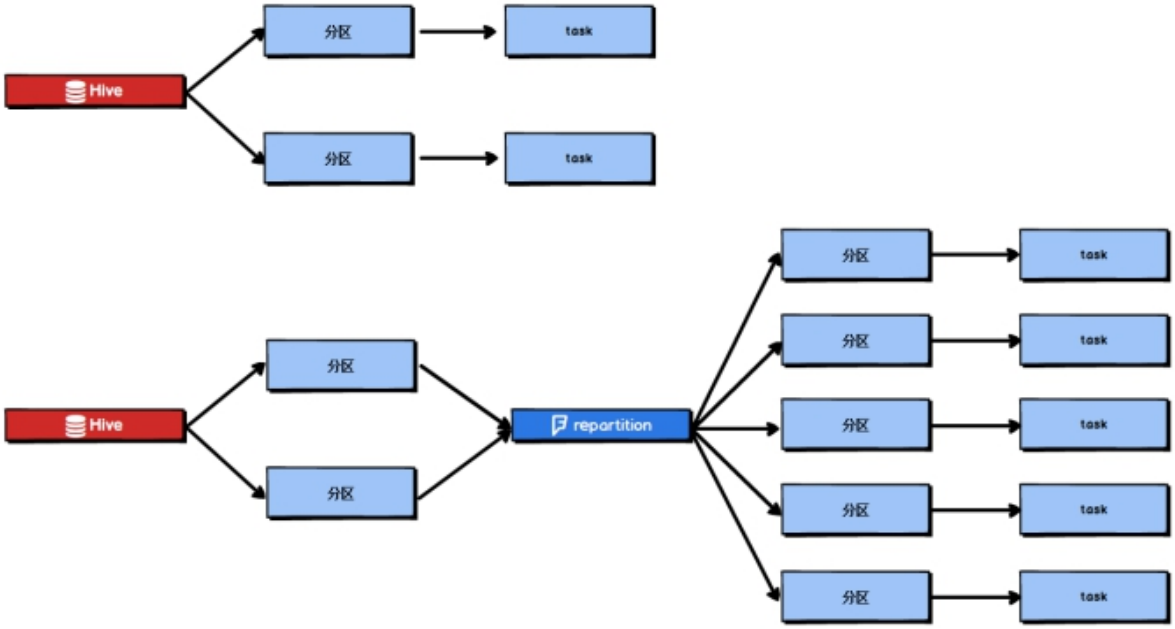
在前面的常规性能调优中我们讲解了并行度的调节策略，但是，**并行度的设置对于Spark SQL是不生效的**，用户设置的并行度只对于Spark SQL以外的所有Spark的stage生效。

Spark SQL的并行度不允许用户自己指定，Spark SQL自己会默认根据hive表对应的**HDFS文件的split个数**自动设置Spark SQL所在的那个stage的并行度，用户自己通spark.default.parallelism参数指定的并行度，只会在没Spark SQL的stage中生效。

由于Spark SQL所在stage的并行度无法手动设置，如果数据量较大，并且此stage中后续的transformation操作有着复杂的业务逻辑，而Spark SQL自动设置的task数量很少，这就意味着每个task要处理为数不少的数据量，然后还要执行非常复杂的处理逻辑，这就可能表现为第一个有Spark SQL的stage速度很慢，而后续的没有Spark SQL的stage运行速度非常快。

解决方案

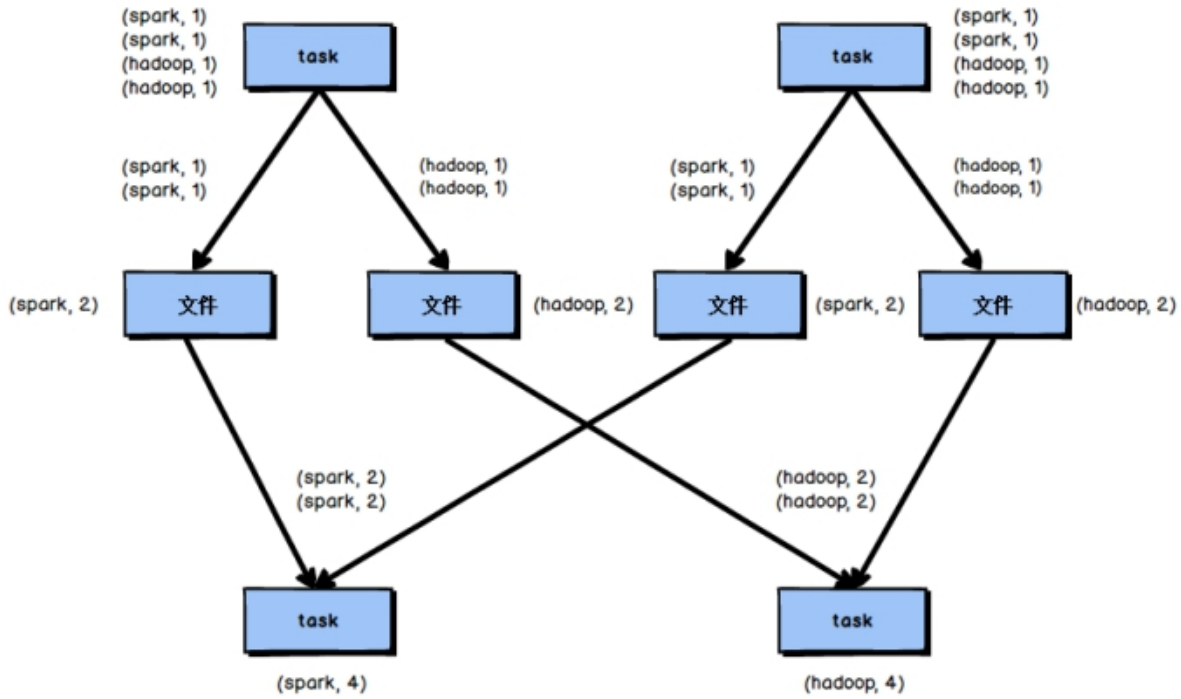
为了解决Spark SQL无法设置并行度和task数量的问题，我们可以使用repartition算子。



Spark SQL这一步的并行度和task数量肯定是没有办法去改变了，但是，**对于Spark SQL查询出来的RDD，立即使用repartition算子，去重新进行分区，这样可以重新分区为多个partition**，从repartition之后的RDD操作，由于不再设计Spark SQL，因此stage的并行度就会等于你手动设置的值，这样就避免了Spark SQL所在的stage只能用少量的task去处理大量数据并执行复杂的算法逻辑。使用repartition算子的前后对比如上图所示。

调优五：reduceByKey融合

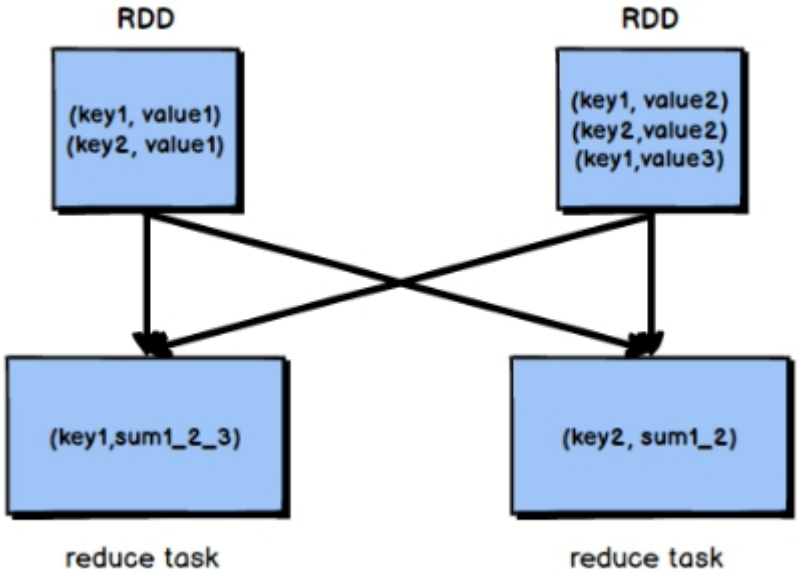
reduceByKey相较于普通的shuffle操作一个显著的特点就是会进行**map端的本地聚合**，map端会先对本地的数据进行combine操作，然后将数据写入给下个stage的每个task创建的文件中，也就是在map端，对每一个key对应的value，执行reduceByKey算子函数。reduceByKey算子的执行过程如图所示

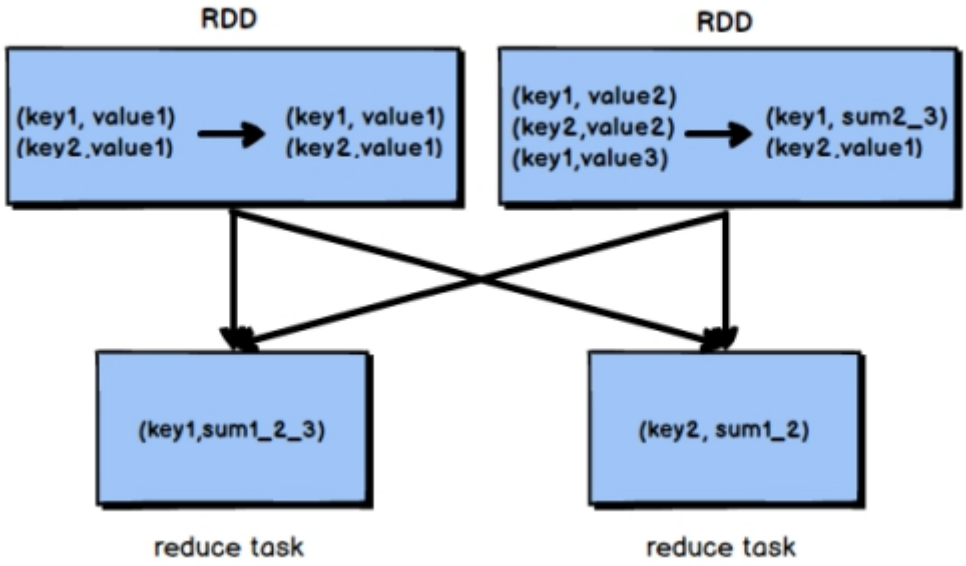


使用reduceByKey对性能的提升如下：

```
1.本地聚合后，在map端的数据量变少，减少了磁盘IO，也减少了对磁盘空间的占用；
2.本地聚合后，下一个stage拉取的数据量变少，减少了网络传输的数据量；
3.本地聚合后，在reduce端进行数据缓存的内存占用减少；
4.本地聚合后，在reduce端进行聚合的数据量减少。
```

基于reduceByKey的本地聚合特征，我们应该考虑使用**reduceByKey代替其他的shuffle算子**，例如groupByKey。reduceByKey与groupByKey的运行原理如图所示：





根据上图可知，groupByKey不会进行map端的聚合，**而是将所有map端的数据shuffle到reduce端**，然后在reduce端进行数据的聚合操作。**由于reduceByKey有map端聚合的特性，使得网络传输的数据量减小，因此效率要明显高于groupByKey。**