作业1

09_茜

August 2019

Contents

1 Spark+Zeppelin安装步骤

由于zeppelin依赖Java, Spark, Hadoop

- brew install scala
- brew install apache-spark
- brew install apache-zeppelin
- zeppelin-daemon.sh start
- 访问http://localhost:8080

2 在Zeppelin上运行WordCount程序

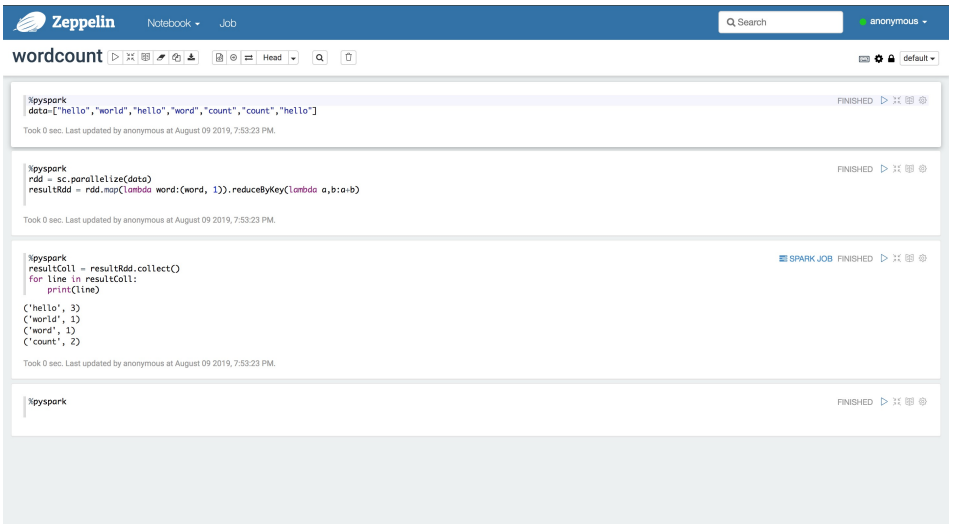


Figure 1: WordCount

3 MapReduce和Spark有哪些区别，请使用具体例子说明

3.1 MapReduce运行机制

比如现在map task有如下数据，目的是求出每年对应的最大值:
(2000, 1)
(2000, 2)
(2000, 3)
(1999, 1)
(1999, 2)
(1999, 3)

图中buffer in memory状态如下: 以上数据分别分布在内存的两个地方:
内存A:
(2000, 1)
(1999, 2)
(2000, 2)
内存B:
(1999, 1)
(2000, 3)
(1999, 3)

Partition, sort and spill to disk状态如下:
内存A:
(1999, 2)

(2000, 1)
(2000, 2)

内存B:
(1999, 1)
(1999, 3)

(2000, 3)

如果有combiner函数(根据键求最大值)，那么在排序后的数据集上运行，即得到:
内存A:
(1999, 2)
(2000, 2)

内存B:
(1999, 3)

(2000, 3)

Merge on Disk会发生如下过程:
内存1:
(1999, 2)
(1999, 3)

内存2:
(2000, 2)
(2000, 3)

merge过程:

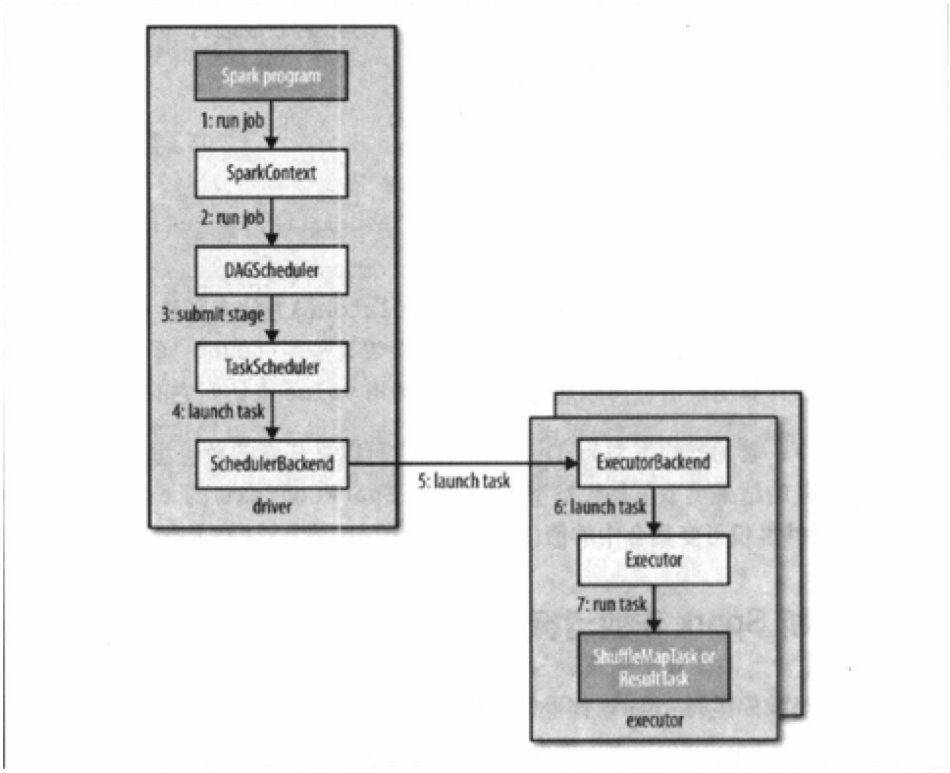
不变，因为我值设定了一个map task

经过reduce后(和combiner相同，都是根据键求最大值)
内存1:
(1999, 3)

内存2:
(2000, 3)

3.2 Spark运行机制

应用>作业>有向无环图>阶段>任务->并行运行在RDD上



mechanism.jpg

Figure 2: Spark mechanism

最重要的是上图的DAG(有向无环图) DAG(有向无环图)的构建: 在阶段中运行的任务类型:shuffle map任务和result任务

- shuffle map任务就像是MapReduce中shuffle的map端部分。每个shuffle map任务在一个RDD分区上进行计算，并根据分区函数把输出写入一组新的分区中，以允许在后面的阶段中取用
- result任务运行在最终阶段，并将结果返回给用户程序。每个result任务在它自己的RDD分区上运行计算，然后把结果发送回driver，再由driver将每个分区的计算结果汇集成最终结果
- 比较复杂的作业要涉及到分组操作，并且要求一个或多个shuffle阶段。例如，该作业用于为存储在inputPath目录下的文本文件计算词频统计分布图(每行文本只有一个单词) val hist: Map[Int, Long] = sc.textFile(inputPath) .map(word =¿ (word.toLowerCase(), 1)) .reduceByKey((a, b) =>a + b)

.map(_.swap)
.countByKey()

前两个转换map()和recudeByKey()，用于计算每个单词出现的频率。 第三个转换是map()，它交换每个键值对中的键和值，从而得到(count, word)对。最后是countByKey()动作，它返回的是每个计数对应的单词量(即词频分布) 通常，每个阶段的RDD都要在DAG中显示

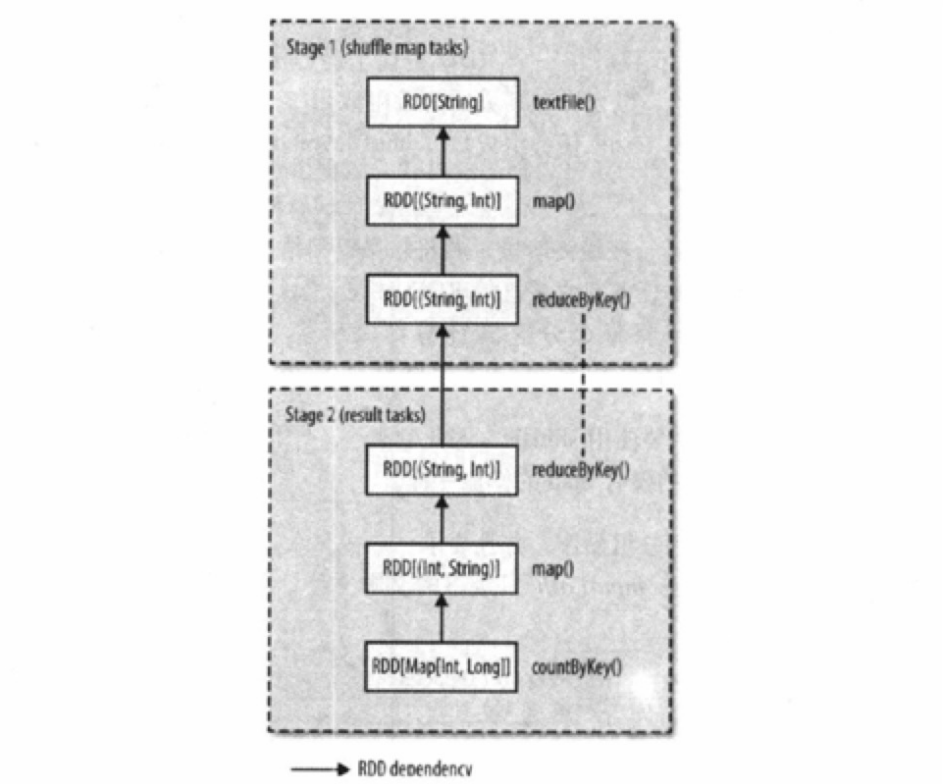


Figure 3: RDD

注意，reduceByKey()跨越了两个阶段，这是因为它是使用shuffle实现的，并且就像MapReduce一样，reduce函数一边在map作为combiner运行阶段，一边又在reduce端又作为reduce运行。它与MapReduce相似的另一个地方是:Spark的shuffle实现将其输出写入本地磁盘上的分区文件中(即使对内存中的RDD也一样)并且这些文件将由下一阶段的RDD读取

4 RDD的本质

RDD之所以称之抽象数据集，是因为RDD存放的不是实际数据集，而是数据转化的步骤规则，((简称DAG)

DAG真正被触发执行实际的数据处理，需要一个导火索，该导火索就是Action计算子，才会被真正唤醒。

弹性分布式数据集(RDD)是所有spark程序的核心

4.1 RDD的创建有三种方法

- 来自一个内存中的对象集合(也称为并行化一个集合)
- 使用外部存储器(例如HDFS)的数据集
- 对现有的RDD进行转换

第一种方法适用于对少量的输入进行并行的CPU密集型计算 对于第二种方法: val text: RDD(String) = sc.textFile(inputPath) Spark内部使用了旧的MapReduceAPI的TextInputFormat(TextInputFormat是默认的InputFormat。每条记录是一行输入。Key是LongWritable类型，存储该行在整个文件中的字节偏移量（不是行数），值是这行的内容，为一个Text对象)读取文件，这意味着它的分割行为与Hadoop中的一致。因此，在使用HDFS的情况下，每个HDFS块对应于一个Spark分区 也可以通过返回一个字符串给RDD来把文本文件作为一个完整的文件对待。(类似于之前的避免分割)。其中第一个字符串是文件路径，第二个字符串是文件内容。由于每个文件都要被加载到内存中，因此这种方法只适合小文件

Spark也可以处理文本文件以外的其他格式。例如，读取顺序文件:
sc.sequenceFile[IntWritable,Text](inputPath)

4.2 转换和动作

动作的效果是立竿见影的，但转换不是，转换是惰性的，因为在对RDD执行一个动作之前都不会为该RDD的任何转换动作采取实际行动 例如:
val text = sc.textFile(inputPath)
val lower: RDD[String] = text.map(_.toLowerCase())
lower.foreach(println(_))

map()方法是一种转换操作，它在Spark内部被表示为可以在稍后某个时刻对输入RDD(文本)的每个元素调用一个函数(toLowerCase())。在调用foreach()方法(这是一个动作)之前，该函数(toLowerCase)实际上并没有被调用。事实上，Spark在结果即将被写入控制台之前，才会运行一个作业来读取输入文件并对其中每一行的文本调用toLowaerCase要判断一个操作是转换还是动作，

我们可以观察其返回类型:如果返回类型是RDD，那么它是一个转换，否则就是一个动作

4.3 持久化

可以用下述命令把年份/温度对构成的中间数据缓存在内存中 scala¿ tuples.cache() res1: tuples.type = MappedRDD[4] at map at ¡console¿: 18 调用cache()并不会立即缓存RDD，相反，它用一个标志来对该RDD进行标记，以指示该RDD应当在作业运行时被缓存。因此，让我们先强制运行一个作业:
scala>tuples.reduceByKey((a, b) =>Math.max(a, b).foreach(println(_)))
INFO BlockManagerInfo: Added rdd_4_0 in memory on 192.168.1.90:64640
INFO blockManagerInfo: Added rdd_4_1 in memory on 192.168.1.90:64640
(1950, 22)
(1949, 111)

来自BlockManagerInfo的日志表明RDD分区已作为作业运行的一部分保存在内存中。日志显示RDD的编号为4(这个编号在调用cache()方法后显示在控制台上)，它有两个分区，分别被编号为0和1.如果我们对缓存的数据集运行另一个作业，将会看到从内存中加载该RDD，这次我们要计算最低温度:
scala>tuples.reduceByKey((a, b) =>Math.min(a, b).foreach(println(_)))
INFO BlockManager: Found block rdd_4_0 locally
INFO BlockManager: Found block rdd_4_1 locally
(1949, 78)
(1950, -11)

相比较而言，MapReduce在执行另一个计算时(两个MapReduce间的)必须从磁盘中重新载入数据集，即使它可以使用中间数据集作为输入，也始终无法摆脱必须从磁盘加载的事实，这必然会影响其执行速度。Spark可以在夸集群的内存中缓存数据集，这也就意味着对数据集所做的任何计算都非常快

以迭代算法为例:

- Spark:上一次迭代计算的结果可以缓存在内存中，以用作下一次迭代的输入
- MapReduce:每次迭代都要作为单个MapReduce作业来运行，因此每次迭代的结果必须写入磁盘，然后在下一次迭代时从磁盘中读回

注意:被缓存的RDD只能由同一应用的作业来读取，如果要在应用之间共享数据，则必须在第一个应用中使用savaAs*()(例如savaAsTextFile(), savaAsHadoopFile()等)方法将其写入外部存储器，然后在第二个应用中使用SparkContext的相应方法(如textfile(), hadoopFile()等)进行加载。同理，当应用终止时，它缓存的所有RDD都将被销毁，除非这些RDD已被显式保存，否则无法再次访问对于交互式探索查询，以及某些迭代算法非常有用