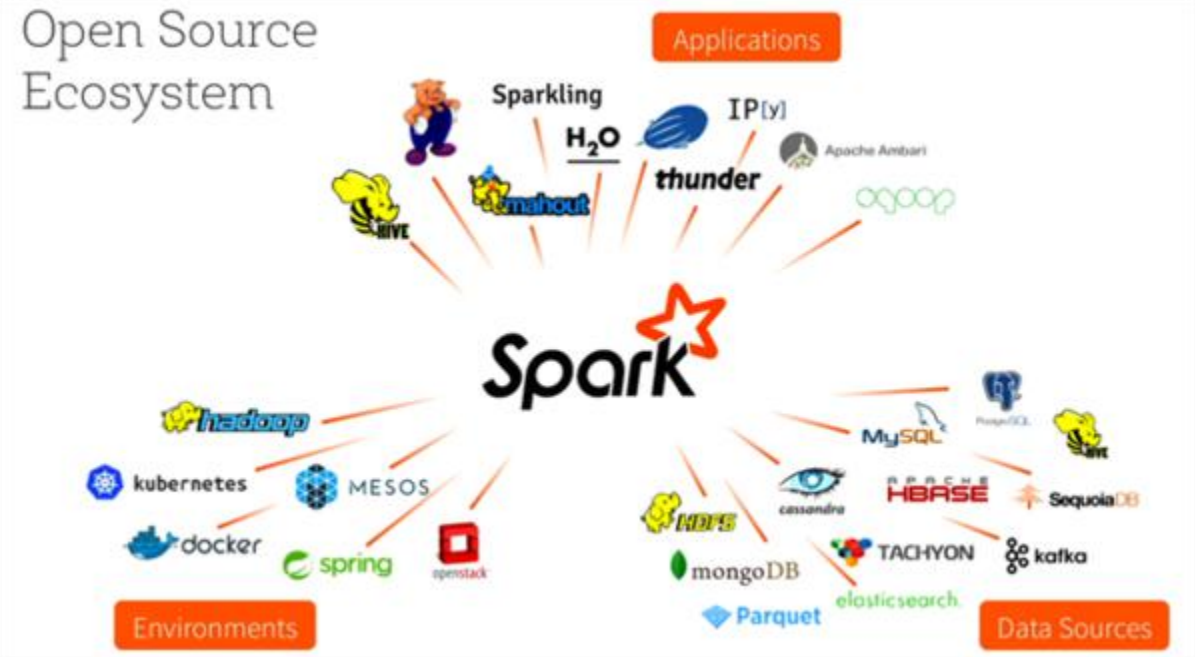
第七章 基于Spark的大数据挖掘



2018-5-14

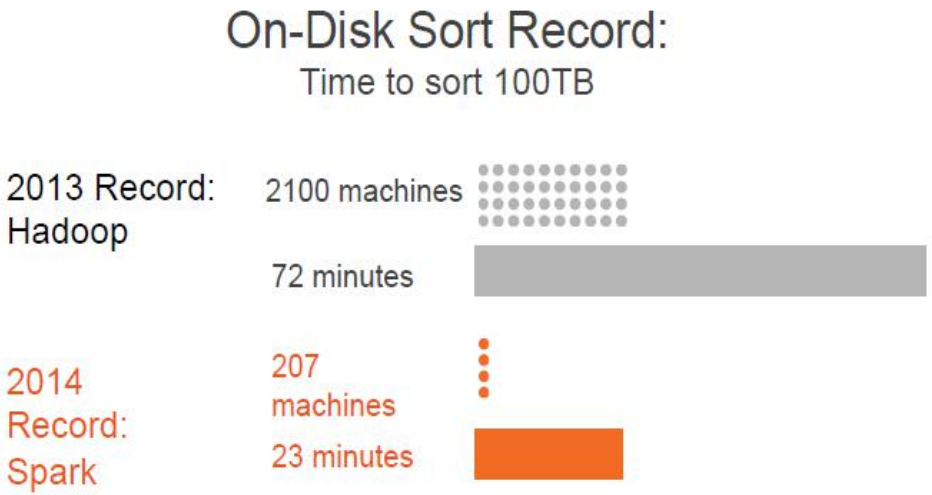
目录

7.1 Spark简介

Spark于2009年诞生于加州大学伯克利分校的AMPLab，最开始它只是伯克利的一个研究生项目。2010年，Spark正式开源，2013年得到Apache基金项目的青睐，并且于2014年成为Apache基金的顶级项目。

Spark简介

2014年11月，Spark在Daytona Gray Sort 100TB Benchmark竞赛中打破了由Hadoop MapReduce保持的排序记录，Spark利用1/10的节点数，把100TB数据的排序时间从72分钟提高到了23分钟。而且Spark只用了其十分之一的节点。



Spark用更少的节点在更短的时间里完成了100T数据的排序

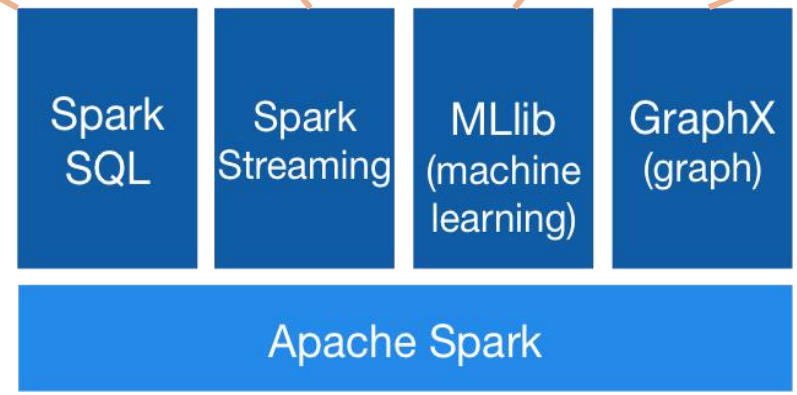
Spark架构

Spark SQL：Spark用来操作结构化数据的程序包，它支持多种数据源：包括HDFS数据、Hive表、MySQL等。

Spark Streaming：对实时数据进行流式计算的组件，支持与Spark core同级别的容错性，吞吐量和可伸缩性。

Spark MLlib：提供多种机器学习算法，包括分类、聚类、回归、协同过滤等，还提供了模型评估等额外功能。

Spark GraphX：用来操作图的程序库，用于并行的图计算，扩展了Spark的RDD API，包含了一些常用图算法。



Spark特点

速度快： 基于分布式的内存计算使得其计算速度比Hadoop快一到两个数量级；

易用性： 支持Java、Scala、Python以及R语言，提供超过80余种高级操作，而且支持Python和Scala以及R的shell编程；

普适性： Spark的目标是提供一栈式大数据处理平台，尽可能多地支持多种数据处理应用，如已经结合SQL、streaming、机器学习以及图计算等复杂的数据分析模块；

多平台支持： 可以在Hadoop YARN、Apache Mesos等分布式管理平台运行Spark，而且支持多种数据源：HDFS、Cassandra、Hbase、Hive等。

Spark WordCount 示例

Scala是一种面向对象和面向函数式的编程，这一特性使得Spark基于Scala的编程方便而快捷，我们通过简单的WordCount单词计数程序来对比一下Spark下Scala编写的程序与Hadoop下Java的WordCount程序的对比：

MapReduce写的单词计数需要分别编写Map函数和Reduce函数，而Spark下只需在同一行中调用几个高级操作即可完成单词计数并且将结果打印出来，开发者也不必在main函数中考虑到Job的调度，方便而快捷。

```
public class WordCount {

  public static class Map extends MapReduceBase implements
      Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        output.collect(word, one);
      }
    }
  }

  public static class Reduce extends MapReduceBase implements
      Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
      int sum = 0;
      while (values.hasNext()) {
        sum += values.next().get();
      }
      output.collect(key, new IntWritable(sum));
    }
  }
```

MapReduce版本的WordCount

Spark下用scala写的WordCount：

```
//获取文件
val lines = sc.textFile(“/path/to/file”)
//进行单词计数
lines.flatMap(_.split(" ")).map((_, 1)).reduceByKey(_+_).collect().foreach(println)
```

基于Spark的大数据挖掘

7.1 Spark简介
7.2 弹性分布式数据集—— RDD
7.3 Spark Streaming
7.4 大规模图计算与挖掘
7.5 Spark机器学习

2018-5-14

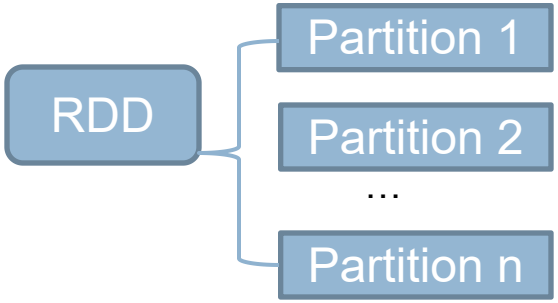
弹性分布式数据集——RDD

RDD是Spark对数据的核心抽象，其实质是分布式元素集合。在Spark中，所有对数据的操作不外乎创建、转化已有RDD以及调用RDD操作进行求值。而Spark会自动将RDD中的数据分发到集群，并将操作并行化执行。

分布式->分布于多台机器，并行计算

弹性->计算过程可在磁盘与内存间交换

在分布式系统中，一般每个RDD都会被分为多个分区（partition），这些分区运行在不同的节点上



RDD的创建

创建RDD——主要有两种方法：

（1）读取外部数据

调用SparkContext.textFile方法从文件创建RDD：

```
val lines = sc.textFile(”/path/to/file”)
```

（2）在驱动器程序中对集合进行并行化

调用SparkContext.parallelize方法：

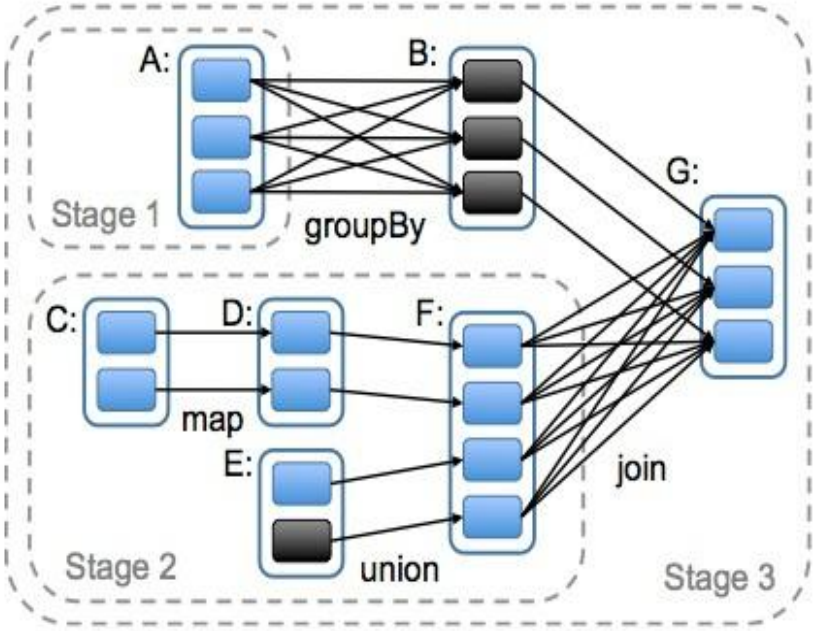
```
val rdd = sc.parallelize(1 to 10)
```

rdd包含了1~10这些数字

还可以用makeRDD、hadoopFile、hadoopRDD等方法来创建RDD,以及基于现有RDD产生新的RDD。这些方法可以视为以上两种方法的变种。

RDD之间的关系

- 在整个流程中，A、B、C、D、E、F、G分别对应一个RDD。
- 相邻的RDD之间存在父子关系，例如：B是A的儿子，A是B的父亲，B和F都是G的父亲。
- 在父子RDD之间，数据的转换与传输有很大不同，例如：A和B之间执行groupby操作，A中不同partition上的数据可能会在B中的任一Partition之中；C和D之间执行map操作，C中各个partition的数据不会随意跑到D中的任一partition中去。



一些RDD操作的依赖性质

RDD的依赖性质——宽与窄

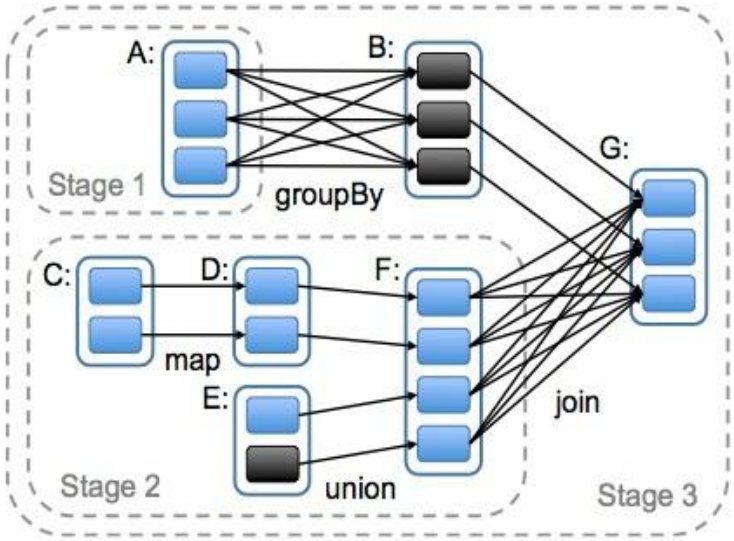
在Spark中，RDD有两种依赖形式：宽依赖(Wide dependency)与窄依赖(Narrow dependency)

窄依赖：是指父RDD的每一个分区最多被一个子RDD的分区所用，表现为一个父RDD的分区对应于一个子RDD的分区或多个父RDD的分区对应于一个子RDD的分区，也就是说一个父RDD的一个分区不可能对应一个子RDD的多个分区。

宽依赖：是指子RDD的分区依赖于父RDD的多个分区或所有分区，也就是说存在一个父RDD的一个分区对应一个子RDD的多个分区。

窄依赖的RDD操作处理起来比较简单，更容易进行基于数据划分的并行化设计；宽依赖的RDD操作处理起来比较复杂，无法通过数据划分进行并行化设计。

RDD宽依赖与窄依赖



一些RDD操作的依赖性质

RDD(A)到RDD(B)的groupBy操作为宽依赖操作，需要对RDD的所有分区进行数据混洗；

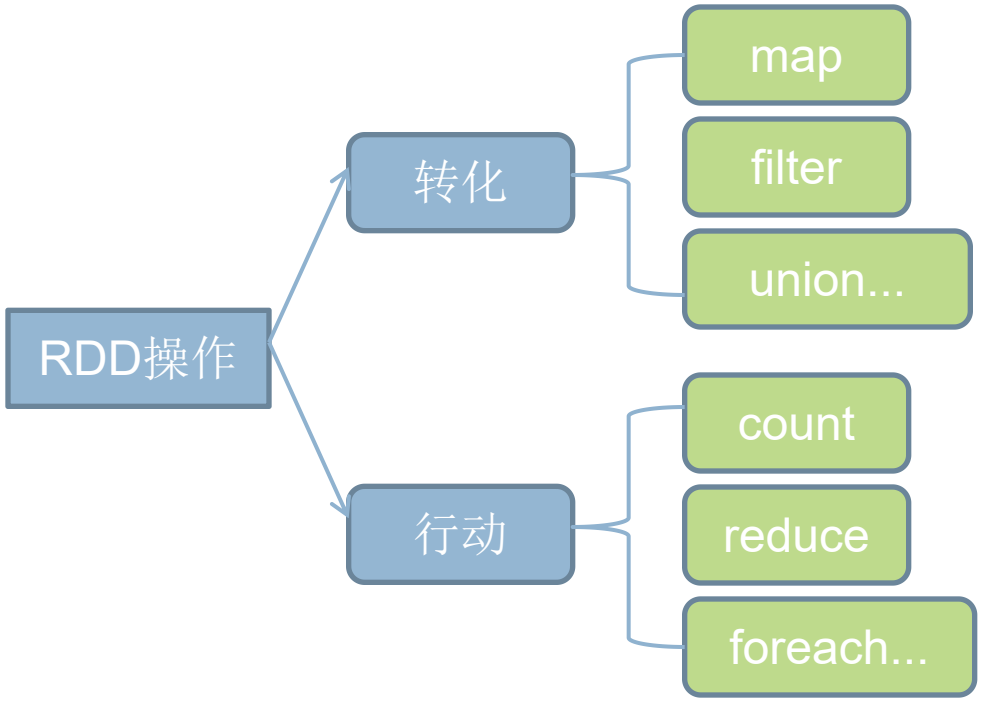
RDD(C)到RDD(D)的map操作是窄依赖，不需要进行数据混洗；

RDD(D)、RDD(E)进行union操作转化为RDD(F)是窄依赖；

RDD(B)、RDD(F)到RDD(G)的join操作是宽依赖

涉及到宽依赖的RDD操作一般都需要进行数据混洗，即将各个分区的数据进行计算或者重新分布，各节点间需要进行网络通信来进行数据传输。

RDD操作



返回一个新的RDD。转化出来的RDD是惰性求值的，只有在行动操作中用到时才会被计算。

返回其他数据类型。对RDD进行实际的操作，把最终求得的结果返回驱动器程序或者写入存储系统。

RDD的转化操作

- map(func) :返回一个新的分布式数据集，由每个原元素经过func函数转换后组成
- filter(func) : 返回一个新的数据集，由经过func函数后返回值为true的原元素组成
- union(otherDataset) : 返回一个新的数据集，由原数据集和参数联合而成

如果你的机器中搭建了Spark开发环境，即可以在终端输入spark-shell或者sparkpy来运行scala或者python版本的shell:

```
scala> val a = sc.parallelize(1 to 9, 3)          //创建一个RDD
scala> val b = a.map(x => x*2)                    //返回一个新的RDD
scala> a.collect                                  //其中每个数是原来的2倍
res10: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
scala> b.collect
res11: Array[Int] = Array(2, 4, 6, 8, 10, 12, 14, 16, 18)
```

RDD的行动操作

- reduce(func)：通过函数func聚集数据集中的所有元素。Func函数接受2个参数，返回一个值。这个函数必须是关联性的，确保可以被正确的并发执行；
- collect()：在Driver的程序中，以数组的形式，返回数据集的所有元素。这通常会在使用filter或者其它操作后，返回一个足够小的数据子集再使用，直接将整个RDD集Collect返回，很可能会让Driver程序out of memory；
- count()：返回数据集的元素个数；

reduceByKey也是行动操作：

```
scala> var a = sc.parallelize(List((1,2),(3,4),(3,6)))      //创建一个RDD
scala> a.reduceByKey((x,y) => x + y)                        //按照键值对进行加法运算
scala> a.collect                                            //收集结果（类似于单词计数）
res7: Array[(Int, Int)] = Array((1,2), (3,10))
```

键值对RDD——pair RDD

- 键值对是Spark许多操作所需要的常见数据类型，通常用来进行聚合计算，我们一般先通过一些初始的ETL（抽取、转化、装载）操作来将其转化为键值对形式；
- 另外，键值对操作还涉及到RDD在各节点上分布情况的高级特性——分区。用户可以通过分区技术把经常访问的数据放到一个节点上，从而减少通信开销，以提升性能。
- Spark为键值对类型的RDD提供了一些专有操作，键值对RDD称为pair RDD。 例如，reduceByKey方法可以对每个键值进行归约；join方法将两个RDD中键相同的元素组合到一起，形成一个RDD。

- 下面语句将创建一个pair RDD，每行的第一个单词作为键，每行本身是值。

```
scala> val pairs = lines.map(x => (x.split(“ ”)(0), x))      //创建一个pair RDD
```

Pair RDD的转化操作

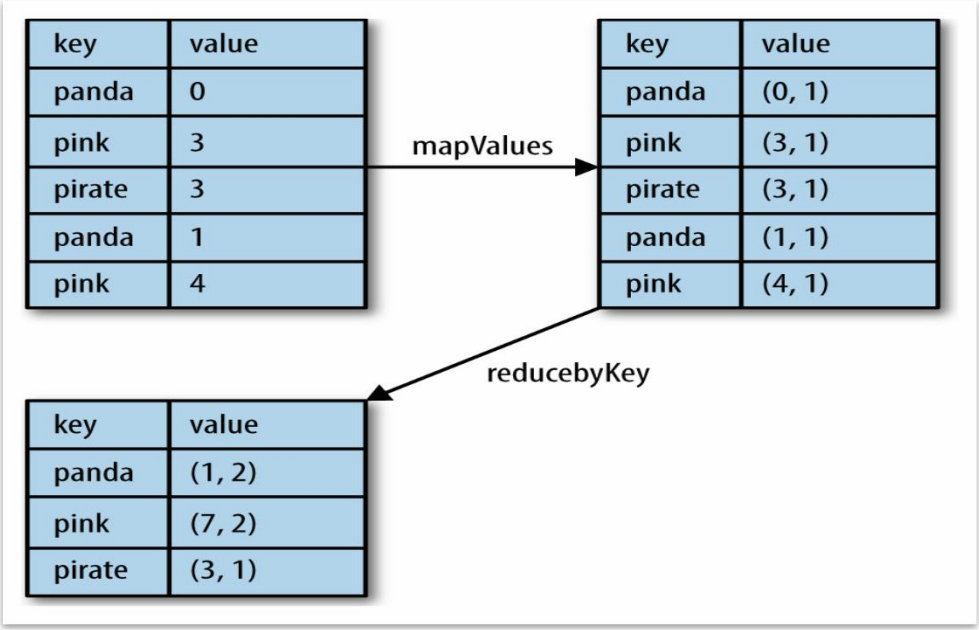
Pair RDD的转化操作（以键值对集{(1,2),(3,4)(3,6)}为例）			
函数名	目的	示例	结果
reduceByKey(func)	合并具有相同键的值	rdd.reduceByKey((x, y) => x + y)	{(1, 2), (3, 10)}
groupByKey()	对具有相同键的值进行分组	rdd.groupByKey()	{(1, [2]), (3, [4, 6])}
mapValues(func)	对pairRDD中每个值应用一个函数而不改变键	rdd.mapValues(x => x+1)	{(1, 3), (3, 5), (3, 7)}
keys()	返回一个仅包含键的RDD	rdd.keys()	{1, 3, 3}
sortByKey()	返回一个根据键排序的RDD	rdd.sortByKey()	{(1, 2), (3, 4), (3, 6)}

Pair RDD的转化操作

聚集：当数据以键值对出现时，针对相同键进行统计很常见。

如：可以使用reduceByKey()和mapValues()来计算每个键对的对应值的平均值：

```
rdd.mapValues(x => (x, 1)).reduceByKey((x, y) => (x._1 + y._1, x._2 + y._2))
```



←计算sum、Count

Pair RDD的行动操作

Pair RDD的行动操作（以键值对集合{(1,2),(3,4)(3,6)}为例）			
函数名	目的	示例	结果
countByKey()	对每个键值对应的元素分别计数	rdd.countByKey()	{(1, 1), (3, 2)}
collectAsMap()	将结果以映射表的形式返回以便查询	rdd.collectAsMap()	Map{(1, 2), (3, 4), (3, 6)}
lookup(key)	返回给定键的所有值	rdd.lookup(3)	[4, 6]

和转化操作一样，所有基础RDD支持的行动操作在pair RDD上也都是可用的，以上是pair RDD额外提供的几个行动操作。

另外，pair RDD还可用于数据分区：因为在分布式程序中，通信的代价是远大于磁盘I/O的，因此控制数据分布以获得最少的网络传输可以极大地提高整体性能。

基于Spark的大数据挖掘

7.1 Spark简介
7.2 弹性分布式数据集—— RDD
7.3 Spark Streaming
7.4 大规模图计算与挖掘
7.5 Spark机器学习

Spark Streaming—流数据处理

- 许多应用需要即时处理收到的数据，例如对飞机、汽车等轨迹的追踪，对实时页面访问的统计，对金融信息的实时处理等等。Spark Streaming就是为这些实时数据应用而设计的模型。它允许用户使用一套和批处理非常接近的API来编写流式计算应用，可以在其中大量重用批处理应用的技术或者代码。
- 与Spark基于RDD的概念相似，Spark Streaming使用离散化流作为抽象数据表示，叫做DStream。

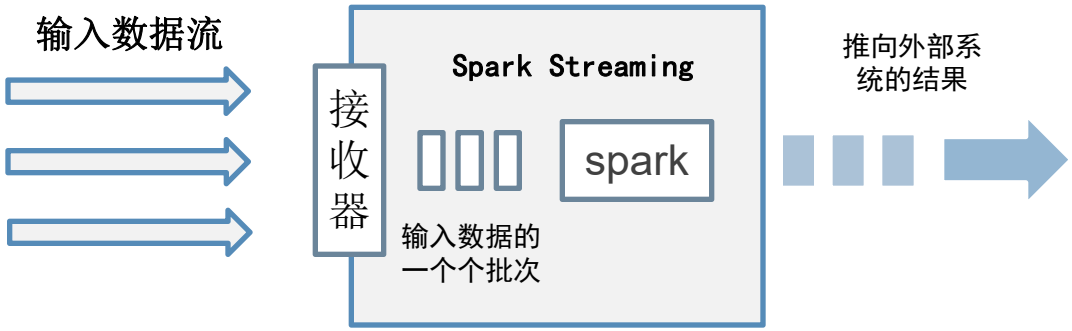
Spark Streaming—流数据处理

- DStream可以从各种输入源创建：Flume、Kafka或者HDFS等。
- 创建出来的DStream支持两种操作：转化和输出。转化操作生成新的DStream，输出操作把数据写入内部系统。一个Dstream就是一系列的RDD（每次输入的一个Batch就是一个RDD）。

本节围绕Spark Streaming的架构、转化操作、输出操作、以及检查点机制来讲解

Spark Streaming—架构

- Spark Streaming使用“微批次”的架构把流式计算当做一系列连续的小规模批处理来对待，将输入数据按照均匀的时间间隔创建成一个个批次，批次间隔从500毫秒到几秒不等。
- 每个批次形成一个RDD，以Spark作业的方式处理并生成其他的RDD，处理结果可以以批处理的方式传给外部系统。



Streaming—DStream

- Spark Streaming的编程抽象是离散化流，也就是Dstream，它是一个RDD序列，每个RDD序列代表一个时间片内的数据（即一个batch）

Dstream是一个持续的RDD序列

- 可以从外部数据源创建Dstream，也可以对其他Dstream应用转化操作产生新的Dstream，Dstream也支持很多前面所提及的RDD支持的转化操作，此外，Dstream还支持“有状态”转化操作，用来聚合不同时间片段的数据。

Streaming—DStream例子

- 我们从StreamingContext开始，从底层创建出SparkContext来处理数据。构造函数还包含处理批次数据的批次间隔。
- 这里，我们用Scala构建流式筛选，打印出日志文件中包含“error”的行：

```
//从SparkConf创建StreamingContext并且制定处理批次间隔为1秒
val ssc = new StreamingContext(conf, Seconds(1))
//连接本机9999端口后，使用接收到的数据创建Dstream（假设该端口有
//数据不断发送过来）
val lines = ssc.socketTextStream(“localhost”, 9999)
//从Dstream中筛选出包含字符串“error”的行
val errorLines = lines.filter(_.contains(“error”))
//打印包含“error”的行
errorLines.print()
```

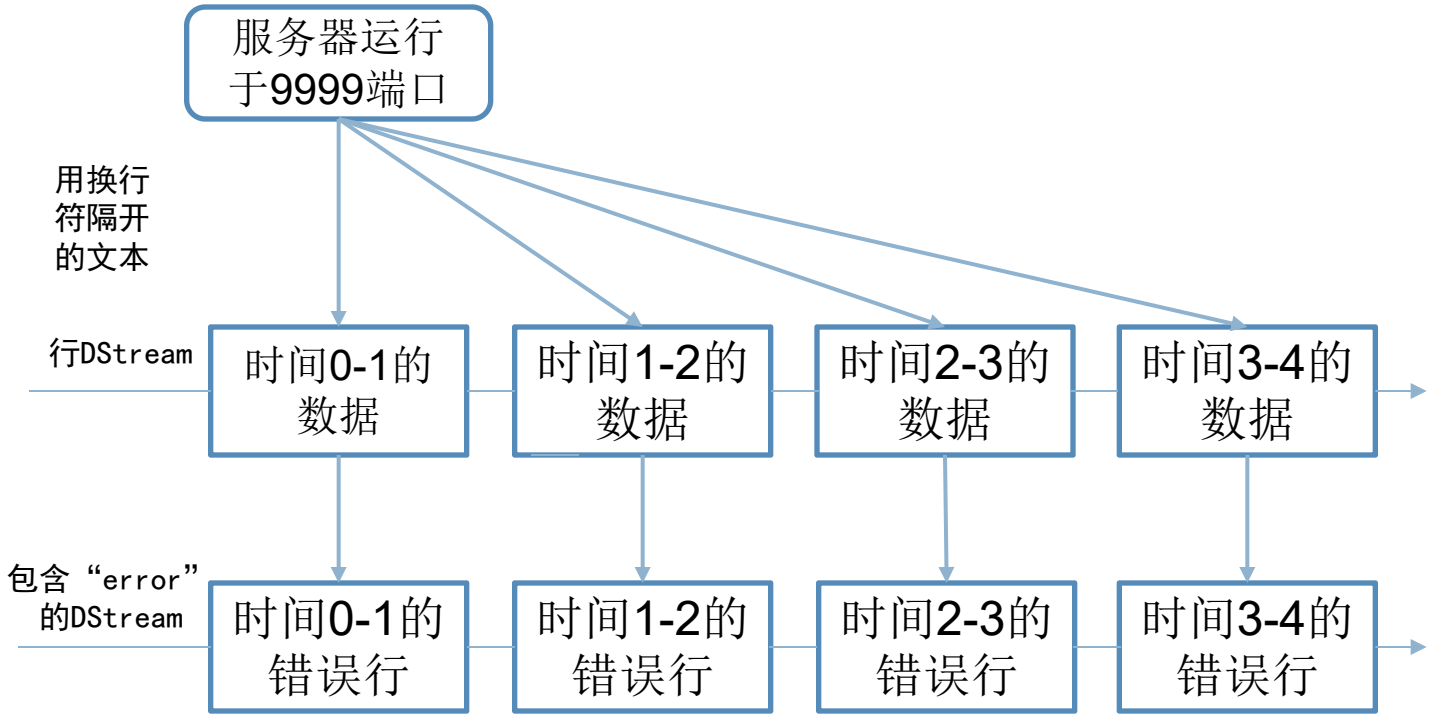
Streaming—DStream例子

- 到这里，我们只是设定好了要进行的计算，系统接收到数据的时候计算就会开始。但是要接收数据，必须显式调用StreamingContext的start()方法，这样Spark Streaming才会把作业不断交给下面的SparkContext去调度执行；
- 执行的过程会在另一个进程中进行，所以需要调用awaitTermination()来等待计算完成，防止应用退出：

```
//启动流计算环境StreamingContext并且等待它完成
ssc.start()
//等待作业完成
awaitTermination()
```

Streaming—DStream例子

- 程序运行过程中的转化关系如下图：



Streaming—转化操作

- Dstream的转化操作分为无状态和有状态的转化：
- 无状态的转化：
  - 无状态的转化操作中的每个批次不依赖于之前的批次数据。它将简单的RDD转化操作应用到每个批次，也就是转化Dstream中的每个RDD。

Dstream的无状态转化操作（部分）			
函数名	目的	示例	用户自定义函数签名
map()	对Dstream每个元素应用函数，返回新元素组成的Dstream	ds.map(x => x+1)	f: (T) -> U
flatMap()	对Dstream每个元素应用函数，返回各元素输出的迭代器组成的DStream	ds.flatMap(x => x.split(“ ”))	f: T -> Iterable[U]
repartition()	改变Dstream分区数	ds.repartition(10)	f: N/A

  - 此外还有filter()，reduceByKey()，groupByKey()等无状态转化操作。

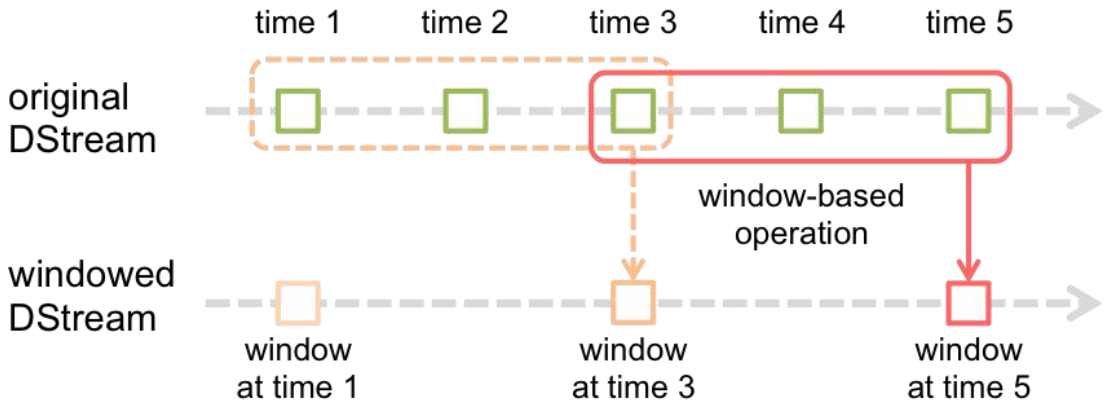
Streaming—转化操作

- 有状态转化操作：

  - 无状态的转化看起来是应用到整个流上，而实际上则是分别应用到Dstream中的每个RDD(batch)上，如reduceByKey会归约每个时间区间中的数据，不会归约不同时间区间的数据；
  - 这就需要我们用到有状态转化操作：即前一批次的数据也被应用到新批次的计算中。主要的应用是滑动窗口，还有updateStateByKey()，前者以一个时间阶段作为滑动窗口进行计算，后者用来跟踪每个键的状态变化；
  - 有状态的转化操作需要我们在StreamingContext中打开检查点机制来确保容错性，需要传递一个目录作为参数给ssc.checkpoint()来打开它，如ssc.checkpoint(“hdfs://…”)。

Streaming基于窗口的转化操作

- 基于窗口的操作会在一个比StreamingContext的批次间隔更长的时间范围内整合多个批次的结果计算出整个窗口的结果。
- 所有基于窗口的操作都需要有两个参数：
  - 窗口大小（时长）：窗口的持续时间
  - 滑动步长：每一次计算的时间间隔
- 需要注意的是：窗口大小和滑动步长必须是批次的整数倍！



一个大小为3，滑动步长为2的窗口

Streaming基于窗口的转化操作

- 一些常见的窗口操作如下，其中都以*windowLength*和*slideInterval*为基础：

Transformation	Meaning
window(*windowLength*, *slideInterval*)	返回一个新的Dstream来表示所请求的窗口操作结果数据
countByWindow(*windowLength*,*slideInterval*)	返回滑动窗口中的元素数据数目
reduceByWindow(*func*, *windowLength*,*slideInterval*)	通过func函数聚合窗口中的元素来返回一个新的单个元素流，func函数必须关联以便于并行计算
reduceByKeyAndWindow(*func*,*windowLength*, *slideInterval*, [*numTasks*])	应用到一个(K,V)对组成的DStream上，返回一个由(K,V)对组成的新的DStream。每一个key的值均由给定的reduce函数聚集起来。注意：在默认情况下，这个算子利用了Spark默认的并发任务数去分组。你可以用numTasks参数设置不同的任务数

Streaming上的输出操作

- 输出操作指定了对流数据经过转化操作得到的数据需要执行的操作：比如将结果输出到外部文件或者打印到屏幕；

输出操作	含义
print()	在DStream的每个批数据中打印前10条元素（默认但是可以自定义值），这个操作在开发和调试中都非常有用。在Python API中调用print()。
saveAsHadoopFiles(prefix, [suffix])	保存DStream的内容为一个hadoop文件。每一个批间隔的文件的文件名基于prefix和suffix生成。
saveAsTextFiles(prefix, [suffix])	保存DStream的内容为一个文本文件。每一个批间隔的文件的文件名基于prefix和suffix生成。
foreachRDD(func)	在从流中生成的每个RDD上应用函数func的最通用的输出操作。这个函数推送每个RDD的数据到外部系统，例如保存RDD到文件或者通过网络写到数据库中。需要注意的是，func函数在驱动程序中执行，并且通常都有RDD action在里面推动RDD流的计算。

Streaming上的输出操作

- 需要注意的两点：
- 输出操作通过懒执行的方式操作DStreams，正如RDD actions通过懒执行的方式操作RDD。具体地看，RDD actions和DStreams输出操作接收数据的处理。因此，如果你的应用程序没有任何输出操作或者将dstream.foreachRDD()用于输出操作，但是没有任何RDD action操作在dstream.foreachRDD()里面，那么什么也不会执行，系统仅仅会接收输入，然后丢弃它们。
- 默认情况下，DStreams输出操作是分时执行的，它们按照应用程序的定义顺序按序执行。

Streaming—24/7不间断运行

- Spark Streaming的一大优势在于它提供了强大的容错性保障，只要输入数据存储在可靠的系统，它就可以根据输入计算出正确的结果，提供“精确一次”执行的语义（就好像所有的数据都是在没有任何节点故障的情况下处理的一样），即使是工作节点或者驱动器程序发生失败。
- 要不间断运行Spark Streaming应用，需要一些特别的配置。第一步就是配置好如HDFS或者Amazon S3等可靠存储系统中的检查点机制。不仅如此，我们还需要考虑驱动器程序的容错性（需要特别的配置代码）以及对不可靠输入源的处理。

Streaming—检查点机制

- 检查点机制主要用于保障容错性，将流上数据阶段性存储到可靠存储系统以便恢复时使用，检查点机制主要为以下两个目的服务：
  - 控制发生失败时需要重算的状态数：检查点机制可以控制在重算状态过程中回溯多远；
  - 提供驱动器程序容错：如果计算过程中驱动器程序崩溃了，你可以重启驱动器程序并且让驱动器程序从检查点恢复，这样Spark Streaming可以读取之前程序运行的进度，并且从断点继续运行。

- 另外，Spark Streaming提供了一个特殊的用户界面，可以让开发者查看应用在干什么，这个界面在Spark的常规界面(http://:4040)的Streaming标签页。该界面展示了批处理和接收器的统计信息。

Streaming—性能参考

- 批次和窗口大小
  - 最常见的问题是可使用的最小批次间隔是多少，Spark Streaming中，500毫秒已被证实为对许多应用而言是比较好的最小批次大小；一般从比较大的批次间隔（十几秒）开始，不断更新批次大小直到找到最合适的处理间隔。对于滑动窗口而言，当计算代价巨大并成为系统瓶颈的时候，可以考虑增加滑动步长；
- 并行度：通过提高并行度减少处理时间
  - 增加接收器数目
  - 将收到的数据显示重新分区
  - 提高聚合计算的并行度
- 垃圾回收和内存使用
  - Java的垃圾回收机制（GC）也可能引起问题，打开java的清除收集器来减少GC引起的不可预测的长暂停；
  - 使用序化的格式缓存RDD也可以减轻GC的压力；
  - Spark也允许我们控制缓存的RDD以何种方式移除(默认是LRU)，可以通过设置spark.cleaner.ttl来显示控制缓存策略。

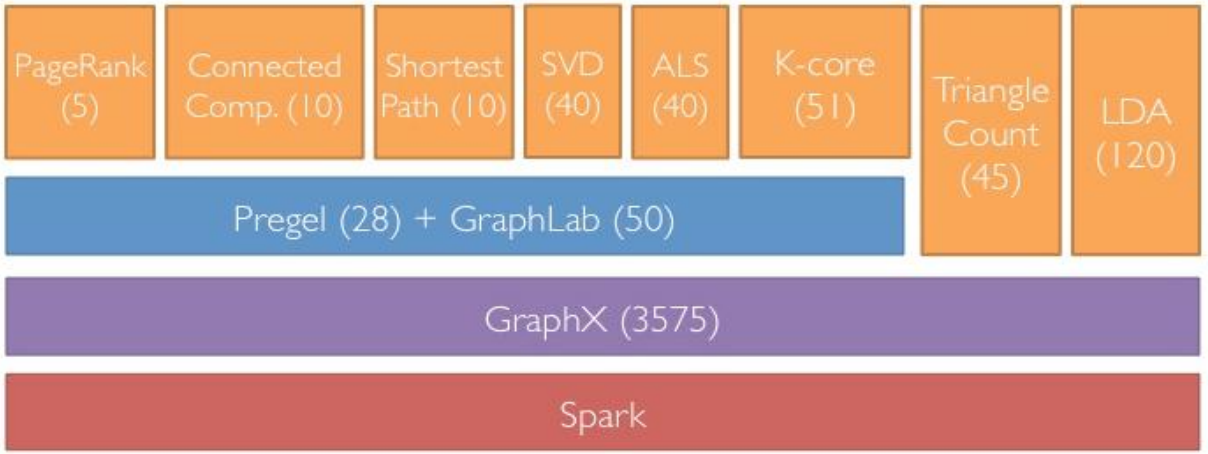
基于Spark的大数据挖掘

7.1 Spark简介
7.2 弹性分布式数据集—— RDD
7.3 Spark Streaming
7.4 大规模图计算与挖掘
7.5 Spark机器学习

2018-5-14

Spark GraphX

- GraphX是Spark中用来操作图的程序库，用于并行的图计算，扩展了Spark的RDD API，包含了一些常用图算法。
- 为了更好地支撑图计算，GraphX提供了一组基本的图操作，如：subgraph、joinVertics、aggregateMessage操作，以及优化了的Pregel（一个专门做图计算的框架）API。

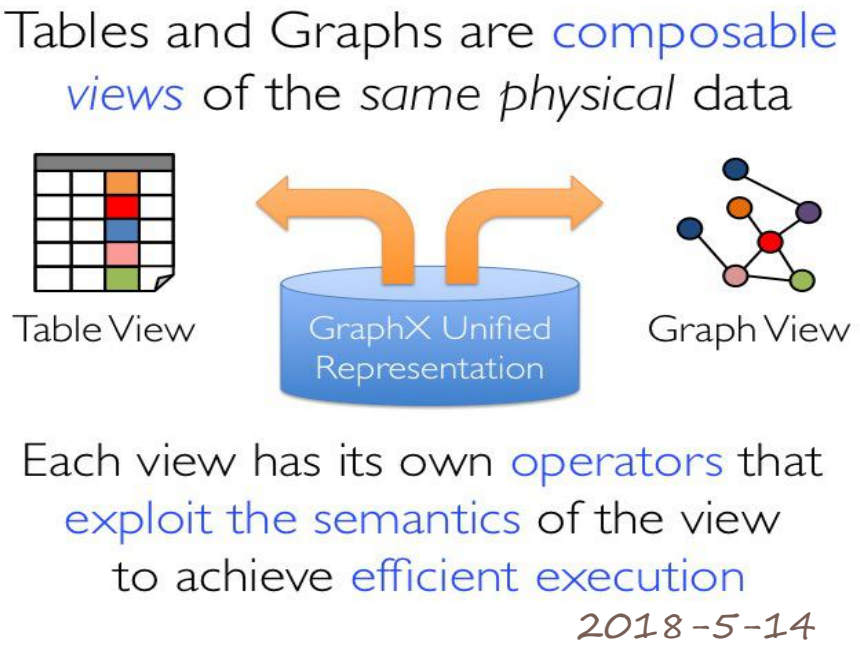


GraphX各部分的代码量

GraphX：两种视图

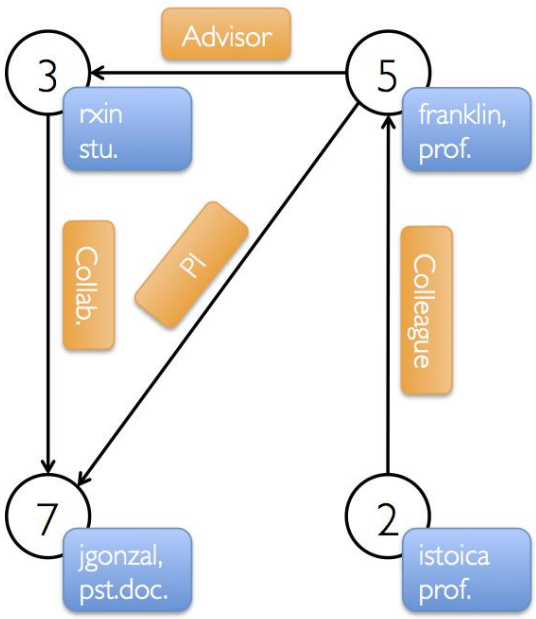
- GraphX通过引入Resilient Distributed Property Graph（一种点和边都带属性的有向多图）扩展了Spark RDD这种抽象数据结构，这种Property Graph拥有两种Table和Graph两种视图（及视图对应的一套API），而只有一份物理存储。

每种个视图都有各自的一套操作，并且通过视图语义来有效执行



GraphX：两种视图

- **Table**视图将图看成Vertex Property Table和Edge Property Table等的组合，这些Table继承了Spark RDD的API(fiter,map等)。



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

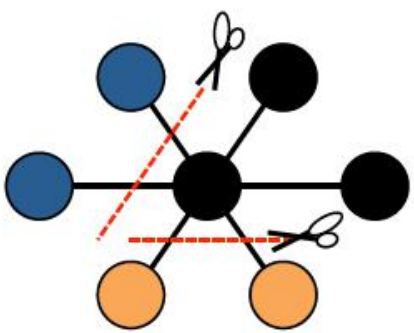
SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

GraphX：两种视图

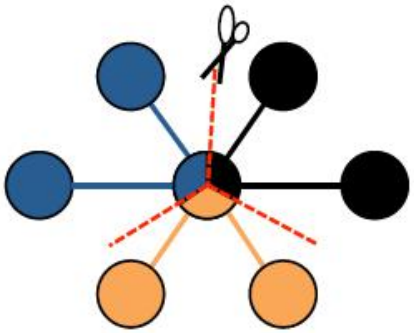
- Graph视图上包括reverse/subgraph/mapV(E)/joinV(E)/mrTriplets等操作：
  - GraphX上的一些操作

GraphX：边分割和点分割

- 大型的图存储总体上有边分割和点分割两种方式：
- 边分割（Edge-Cut）：每个顶点都存储一次，但有的边会被打断分到两台机器上。这样做的好处是节省存储空间；坏处是对图进行基于边的计算时，对于一条两个顶点被分到不同机器上的边来说，要跨机器通信传输数据，内网通信流量大。
- 点分割（Vertex-Cut）：每条边只存储一次，都只会出现在一台机器上。邻居多的点会被复制到多台机器上，增加了存储开销，同时会引发数据同步问题。好处是可以大幅减少内网通信量，因此比边分割更受欢迎一些。



(a) Edge-Cut

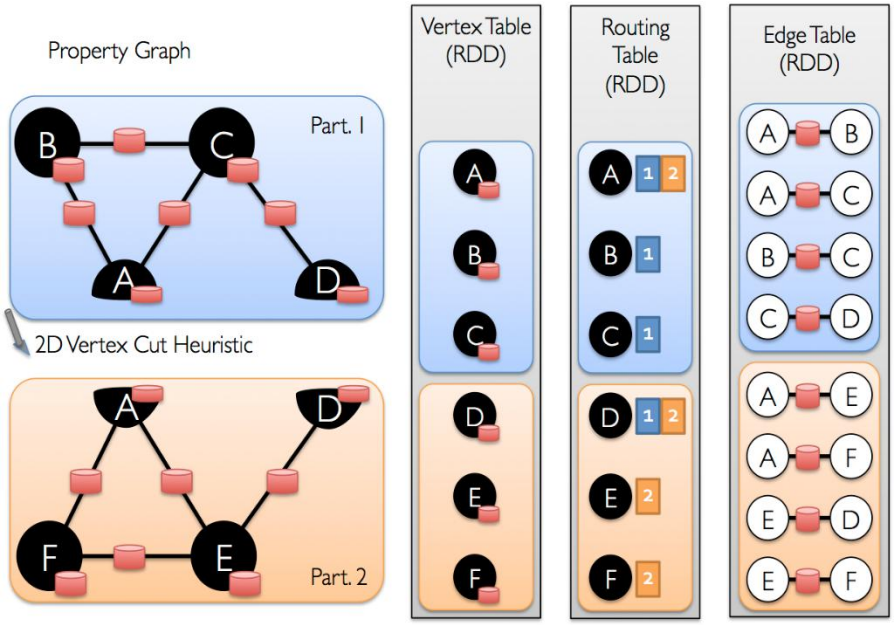


(b) Vertex-Cut

GraphX：存储模式

- Graphx借鉴PowerGraph，使用的是Vertex-Cut(点分割)方式存储图，用三个RDD存储图数据信息：

  - VertexTable(id, data)：id为Vertex id，data为Edge data
  - EdgeTable(pid, src, dst, data)：pid为Partion id，src为原定点id，dst为目的顶点id
  - RoutingTable(id, pid)：id为Vertex id，pid为Partition id



GraphX：PageRank

以下是GraphX用GraphX实现PageRank的代码：

```
// 导入图的边
val graph = GraphLoader.edgeListFile(sc,
"graphx/data/followers.txt")
// 运行PageRank
val ranks = graph.pageRank(0.0001).vertices
// 获取用户名（ID）
val users = sc.textFile("graphx/data/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
// 通过用户名对rank值进行join操作然后返回用户名及其rank值
val ranksByUsername = users.join(ranks).map {
  case (id, (username, rank)) => (username, rank)
}
// 打印结果
println(ranksByUsername.collect().mkString("\n"))
```

基于Spark的大数据挖掘

7.1 Spark简介
7.2 弹性分布式数据集—— RDD
7.3 Spark Streaming
7.4 大规模图计算与挖掘
7.5 Spark机器学习

2018-5-14

Spark MLlib

- MLlib是Spark提供机器学习函数的库，它是专门为集群上并行运行的情况而设计的。MLlib中包含很多的机器学习算法，可以在Spark支持的Scala、Python以及Java编程语言中使用。

- 机器学习本身是一个很大的话题，可以用几个学期的课来进行一些算法的讲解，因此，本小节注重于如何在Spark中使用机器学习，而不去介绍机器学习本身。

Spark MLlib

- MLlib的设计理念是将数据以RDD形式表示，然后在分布式数据集上调用各种算法。它还引入了一些数据类型：比如点和向量。归根结底，MLlib就是RDD上的一系列可供调用的函数集合。
- 机器学习算法尝试根据**训练数据**使得表示算法行为的数学目标最大化，并且以此来进行预测或者做出决定。机器学习问题分为：**分类、回归、聚类**，每一种都有不一样的目标：
  - Spark将机器学习算法都分成了两个模块：
  - 训练模块：通过训练样本输出模型参数
  - 预测模块：利用模型参数初始化，预测测试样本，输出与测值
- 所有的机器学习算法都要定义每个数据的**特征集**，也就是传给学习函数的值（如用户观看电影的类型、地点等）。大多数算法都是为**数值特征**定义的，因此**提取特征并转化为特征向量**是机器学习中非常重要的一步。

MLlib的数据类型

- MLlib包含一些特有的数据类型，他们位于org.apache.spark.mllib包内
- 主要的几个数据类型如下：
  - Vector：一个数学向量。MLlib既支持稠密向量也支持系数向量，稠密向量是将向量的每一位都存储下来，而稀疏向量只存储非零位以节省空间；
  - LabeledPoint：在诸如分类或者监督学习的算法中， LabeledPoint用来表示带标签的数据点，其包含一个特征向量与一个标签；
  - Rating：用户对一个产品的评分，用于产品推荐；
  - 各种Model类：每个model都是算法训练的结果，一般都有一个predict()方法来对新的数据点或者数据点组成的RDD应用该模型进行预测。
- 大多数的算法直接操作由Vector、LabeledPoint或者Rating对象组成的RDD，用户可以读取外部数据然后经过map()操作将数据对象转化为MLlib数据类型。

MLlib：向量（Vector）操作

- Vector是MLlib中最常见的数据类型，其使用有一些需要注意的地方：

  - （1）区分稠密向量与稀疏向量（稠密向量会把所有维度的值存在浮点数组中，稀疏向量只把各维度非零值存储下来），当非零元素占比较少时，我们通常使用稀疏向量，这样一来有利于减轻内存的压力，也可以提高计算速度；

  - （2）在Python中创建向量与Scala和Java中有些差别：Java和Scala中，Vector类是位数据表示服务的，没有在API中提供向量的加减法操作，这主要是为了让MLlib保持在较小规模，为了实现一套完整的线性代数库则超出了本工程的范围，开发者可以接种第三方库来实现向量的一些操作。。而在Python中则可对稠密向量使用NumPy进行这些数学操作

MLlib：用scala创建向量

- 下面举个例子来说明向量的创建过程

```
import org.apache.spark.mllib.linalg.Vectors

//创建稠密向量<1.0, 2.0, 3.0>；Vector.dense接受一个数组或者一串值
val denseVec1 = Vectors.dense(1.0, 2.0, 3.0)
val denseVec2 = Vectors.dense(Array(1.0, 2.0, 3.0))

//创建稀疏向量<1.0, 0.0, 2.0, 0.0>，该方法只接收向量的维度(4)以及非零
//向量的位置及其对应值,先存维度，第一个数组存位置，第二个数组存数值
val sparseVec = Vectors.sparse(4, Array(0, 2), Array(1.0, 2.0))
```

MLlib：基础统计分析

- 基础统计分析是进行机器学习的必不可少的步骤，能够辅助用户进一步地进行数据挖掘，**MLlib**包括了统计汇总、相关性分析、分层抽样、假设检验以及随机数生成等基础的统计分析功能。这些功能在进行学术实验和数据分析具有重要的作用。

- **1、统计汇总**
- 对于RDD[Vector]格式的数据，我们可以使用Statistic类中的colStats()方法来进行数据的列统计汇总。colStats()方法返回一MultivariateStatisticalSummary的实例，它包含了列向量的一些信息，包括其最大最小值以及均值、方差、非零元素数量等。

MLlib：基础统计分析

- 例7-14：列的统计汇总
  - import org.apache.spark.mllib.linalg.Vectors
  - import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Statistics}
  - val observations: RDD[Vector] = … //一个[Vector]的RDD
  - // 计算列的统计汇总
  - val summary: MultivariateStatisticalSummary = Statistics.colStats(observations)
  - println(summary.mean) // 计算每列的均值
  - println(summary.variance) // 计算列方差
  - println(summary.numNonzeros) // 计算每列非零数

MLlib：基础统计分析

- **2、相关性分析**
- 在进行统计分析的时候，我们经常会对两组或者多组数据进行相关性的分析。在Spark MLlib中，提供了对多系列数据之间两两相关度的计算的支持，算法灵活易用。目前支持的算法为Perarson相关和Spearman相关。

MLlib：基础统计分析

- 例7-15：相关性分析
  - import org.apache.spark.mllib.linalg._
  - import org.apache.spark.mllib.stat.Statistics
  - import org.apache.spark.rdd.RDD
  - val seriesX: RDD[Double] = … // 序列X
  - val seriesY: RDD[Double] = … // 序列Y（和序列X有相同的分区数和基数）
  - // 默认使用Perarson相关算法
  - val correlation: Double = Statistics.corr(seriesX, seriesY, "pearson")
  - val data: RDD[Vector] = … // 这里的Vector是行向量
  - val correlMatrix: Matrix = Statistics.corr(data, "pearson")
  - println(correlMatrix.toString)

MLlib：基础统计分析

- **3、随机数生成**
- 随机数生成在随机算法、原型设计和性能测试方面很有用处，在很多数据科学的实验中，我们往往都需要生成大量的随机数来验证算法。MLlib支持按照指定的分布来生成随机数。
- RandomRDD提供工厂方法生成随机的double RDD或者Vector RDD。

MLlib：基础统计分析

- 例7-16：随机数生成
  - import org.apache.spark.SparkContext
  - import org.apache.spark.mllib.random.RandomRDDs._
  - val sc: SparkContext = ...
  - // 生成100万个Double类型的数值，服从正态分布N(0 , 1)，并均匀分布在10个分区中
  - val u = normalRDD(sc, 1000000L, 10)
  - // 将刚刚得到的分布转化为服从N(1, 4) 分布的Double RDD.
  - val v = u.map(x => 1.0 + 2.0 * x)

MLlib：基础统计分析

- **4、分层抽样**
- 随机数生成在随机算法、原型设计和性能测试方面很有用处，在很多数据科学的实验中，我们往往都需要生成大量的随机数来验证算法。MLlib支持按照指定的分布来生成随机数。
- RandomRDD提供工厂方法生成随机的double RDD或者Vector RDD。

MLlib：基础统计分析

- 例7-17：分层抽样
  - // 获取一个键值对RDD： RDD[(K, V)]
  - val data = sc.parallelize(
  - Seq((1, 'a'), (1, 'b'), (2, 'c'), (2, 'd'), (2, 'e'), (3, 'f')))
  - // 确定每一个key的抽样比例
  - val fractions = Map(1 -> 0.1, 2 -> 0.6, 3 -> 0.3)
  - // 从每一层中获取粗略的样本
  - val approxSample = data.sampleByKey(withReplacement = false, fractions = fractions)
  - // 从每一层中获取精确的样本
  - val exactSample = data.sampleByKeyExact(withReplacement = false, fractions = fractions)

MLlib:性能考量

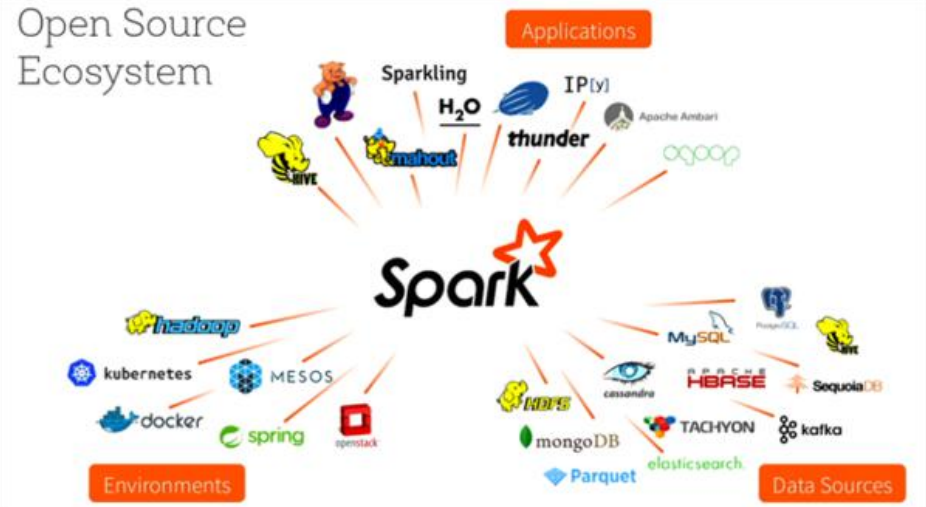
- **准备特征**：尽管机器学习中经常强调所使用的算法，但是在实践环节中算法的好坏只取决于所使用的特性，特征准备尤为重要：
  - 添加信息更丰富的特征
  - 将现有特征转化为合适的向量表
- **配置算法**：合理地配置算法，比如改变基于SGD或者ALS算法的迭代次数，确保在评估参数时将测试数据排除在训练集之外；
- **缓存RDD以重复使用**：MLlib中大多算法是迭代的，对数据进行反复操作，缓存数组非常重要，当内存cache不下的时候尝试persist(StorageLevel.DISK_ONLY)
- **识别稀疏程度**：在数据的处理上，当至多10%的位非零时采取稀疏表示花费的代价更小；
- **并行度**：对大多程序来说，RDD分区数至少和CPU核心数相当，这样才能达到完全的并行；同时，分区也不宜过多，这样会增加通信开销。

本章总结

- 本章我们介绍了Spark这一当下流行的大数据计算处理框架，并且重点讲解了其RDD操作的原理以及GraphX、MLlib 、Spark Streaming等计算框架：
  - 7.2节我们介绍了RDD运行模型及RDD的常见操作，这里包括了Spark的核心概念，是学习应用Spark的关键，而且在后面的学习中，我们看到，无论是图操作、机器学习还是Spark Streaming，其基本操作都是继承于RDD操作的；
  - 7.3节我们介绍了Spark的大规模图计算框架——GraphX，它集成了一些有用的图算法如pagerank、shortest path以及triangle counting等，方便了对大规模图数据的处理；
  - 7.4节讲的是Spark下的机器学习MLlib，Spark基于内存的计算框架非常适合于机器学习中需要多次迭代的算法，因为不像Hadoop那样每次迭代后需要将数据存入磁盘然后再Load，Spark将数据放在内存中，每次迭代后的结果RDD不需要存入磁盘，大大减少了磁盘I/O，将处理速度提升了2个数量级；
  - 7.5节是Spark Streaming的内容，流上的处理是Spark的一大优势（相对于传统的大数据分析处理框架），这个框架使得Spark可以进行实时数据的计算，而且性能尚好，支持滑动窗口的操作以及数据的重新划分，使得其在流式计算与实时应用中能大放异彩。

本章总结

- Spark致力于基于内存的大数据计算和处理框架，不断融合大数据处理所需的各种功能和算法，在带来了处理性能提升的同时，也将大数据计算技术带向向一栈式平台发展，使得在开发大数据应用的过程变得越来越方便快捷



All In One

谢谢！