Spark GraphX分布式图计算实战

2016-10-31 22:16 | 畅游DT时代

作者简介

陆昕 (大数据分析师)

- 硕士，毕业于美国纽约州立大学石溪分校
- 中国联通网研院网优网管部IT技术研究团队成员
- 专注于大数据行业应用、分布式计算技术（MPP数据库、Hadoop、Spark等）

1. 概述

在公众号之前发表的文章《社区发现算法初探》中，介绍了图算法的一些基础理论，对于关注我们的读者，相信已经对图算法在大数据领域中的应用已经有了初步的了解。之前的文章中已经提及，如今互联网、电信网络、金融网络的规模不断增长，企业对庞大网络中社区结构的洞察需求也越发迫切，基于分布式架构的图处理引擎应运而生，比如Google的Pregel， Apache的开源的图计算框架Giraph，以及卡内基梅隆大学主导的GraphLab等，当然还有本文的主角——基于Spark的GraphX。

本文首先介绍了GraphX与几种常见图处理引擎的区别与联系，随后以实战与理论相结合的方式对GraphX中的基本概念和图的操作做了简介，最后以Louvain算法的一个开源实现为例，讲解GraphX程序在Spark集群中进行图计算的全流程。

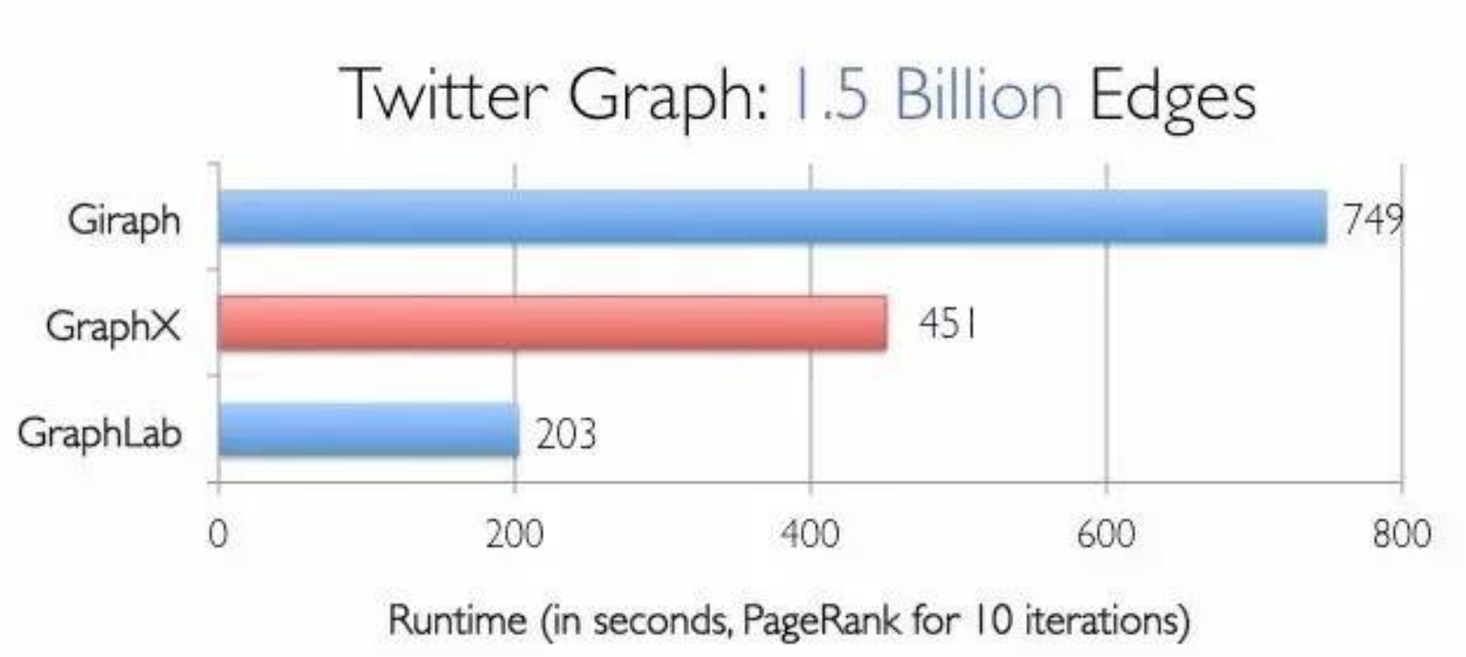
2. GraphX与其他分布式图处理引擎对比

在各类图的并行计算框架中，Pregel、GraphLab和Giraph是非常类似的，都是基于BSP（Bulk Synchronous Parallel）模式，即整体同步并行。它将计算分成一系列的超步（super step）的迭代（iteration）。从纵向上看，它是一个串行模式，而从横向上看，它是一个并行的模式，每两个超步之间设置整体同步点，确定所有并行的计算都完成后再启动下一轮迭代。BSP最大的好处是编程简单，但在某些情况下BSP运算的性能非常

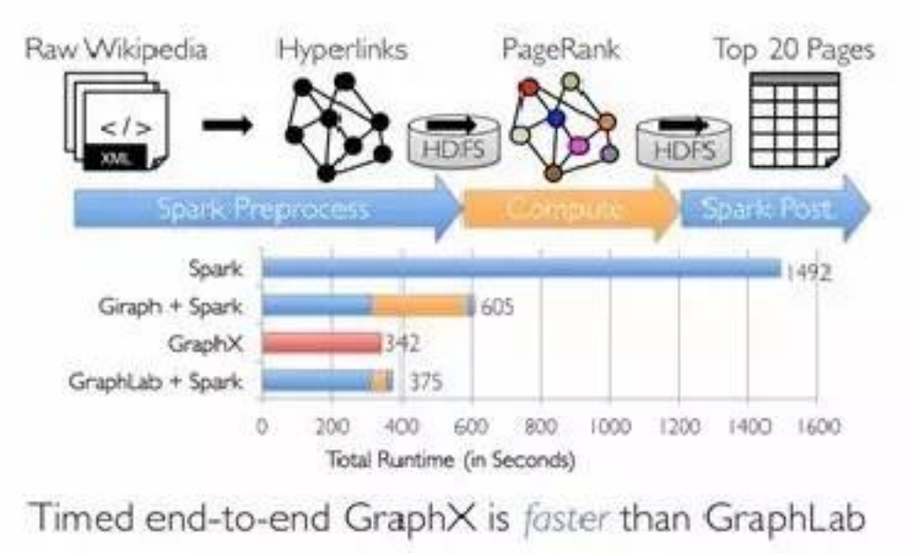
差，系统速度取决于最慢的计算任务。

GraphX实现了一套基于Spark框架的分布式图处理API，借助于Spark的资源管理系统，一定程度上避免并行运算资源分配不均导致的木桶效应。其使用了扩展自RDD的VertexRDD、EdgeRDD以及Graph等抽象数据结构，实现了数据的图存储，并与Spark的其他组件有着天然的兼容性。在图算法的实现上GraphX则可认为是GraphLab(C++)和Pregel(C++)在Spark(Scala)上的重写及优化。

性能上，GraphX比GraphLab慢2-3倍，主要原因是GraphX运行在JVM上，无法与GraphLab的C++程序相比；此外 GraphLab不受Spark框架的限制，可以通过线程来共享内存。然而与其他分布式图计算框架相比，GraphX最大的优势在于将基于大数据并行处理架构Spark与图并行处理引擎合而为一，数据可以在数据集（RDD）、表（SparkSQL）和图之间快速灵活地进行转换，帮助分析师方便高效地完成图计算的一整套流水作业。相比之下，其他计算框架则没有Spark这种完善的处理流程，在诸如数据预处理、存储和数据库交互方面会耗费大量时间，其端到端的处理速度反而不及Spark GraphX。



GraphX与其他图计算框架执行PageRank算法性能对比



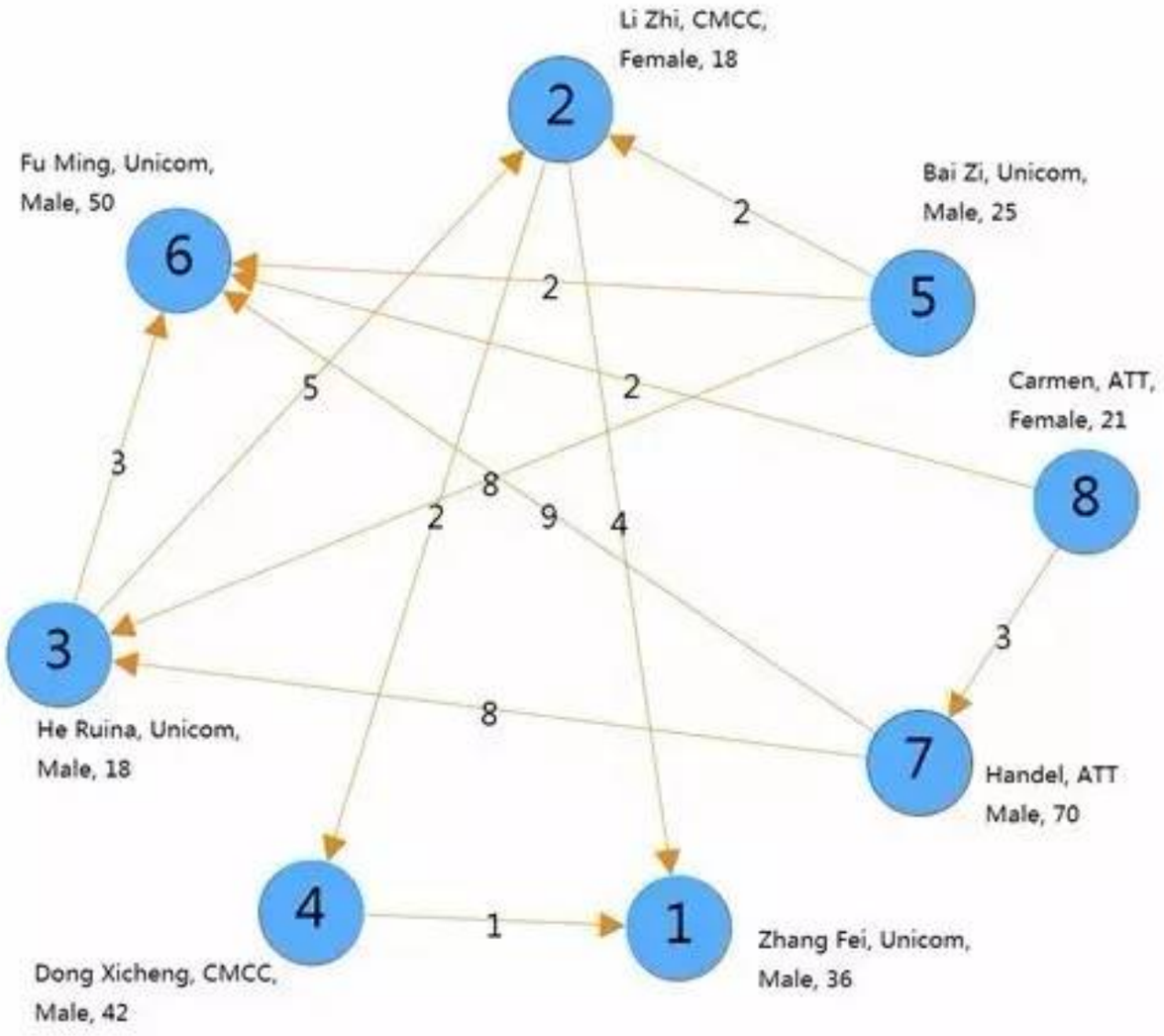
GraphX与其他计算框架“端到端”图计算性能对比

3. GraphX框架

1) 属性图

在GraphX中使用有向属性图（Property GraphX）对图数据进行描述，图中每个顶点使用一个64bit的长整形作为其在图中的唯一编号，而每一条边则包含源节点和目标节点的编号。此外，每个节点和边还具备“属性”，即携带的信息，例如用户姓名和标签、网页的链接、两个节点间的关系等。

下图是一个简化的电信网络，图中的每个节点表示一个手机用户，其在网络中有唯一的编号；每个节点携带有4个“属性”，分别是姓名、运营商、性别以及年龄；每条边是有向的，由通话发起者指向通话接听者，边携带的“属性”是通话的次数，也可视为权重，标识了两个用户之间的联系紧密程度。这个简单的电信网络将在下面的GraphX操作介绍中多次用到。



一个简单的电信网络构成的“属性图”

```
//在Spark中使用GraphX模块，首先要导入相应的库
import org.apache.Spark.graphx._
import org.apache.Spark.rdd.RDD
//数据准备：在scala中使用数组初始化一组节点和边
val vertexArray = Array(
  (1L, ("Zhang Fei", "Unicom", "Male", 36)),
  (2L, ("Li Zhi", "CMCC", "Female", 18)),
  (3L, ("He Ruina", "Unicom", "Male", 23)),
  (4L, ("Dong Xicheng", "CMCC", "Male", 42)),
  (5L, ("Bai Zi", "Unicom", "Male", 25)),
  (6L, ("Fu Ming", "Unicom", "Male", 50)),
  (7L, ("Handel", "ATT", "Male", 78)),
  (8L, ("Carmen", "ATT", "Female", 21))
  )
val edgeArray = Array(
  Edge(2L, 1L, 4),
  Edge(2L, 4L, 2),
  Edge(3L, 2L, 5),
  Edge(3L, 6L, 3),
  Edge(4L, 1L, 1),
  Edge(5L, 2L, 2),
  Edge(5L, 3L, 8),
  Edge(5L, 6L, 2),
  Edge(7L, 6L, 9),
  Edge(7L, 3L, 8),
  Edge(8L, 7L, 3),
  Edge(8L, 6L, 2)
  )
```

2) 图、顶点与边的数据类型

一个属性图由顶点与边构成，GraphX为顶点定义了基本数据类型VertexRDD，而为边定义了EdgeRDD。从名称就可以看出，这两种数据类型都继承自Spark的弹性分布式数据集RDD。和基本的RDD一样，用户可以使用parallelize方法将Scala对象转化成顶点和边，当然也可以从HDFS中直接读取数据文件，将分布式存放的图数据批量装载进Spark。

```
//将顶点数据和边数据转化为RDD

val vertexRDD: RDD[(Long, (String, String, String, Int))] = sc.parallelize(vertexArray)

val edgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)
```

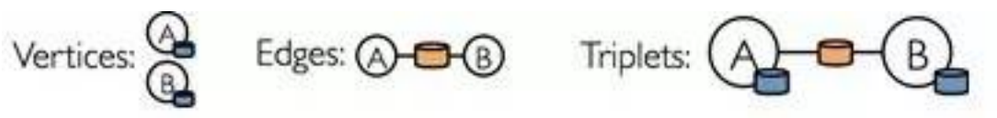
有了顶点和边，我们就可以使用图构建函数读入顶点和边，建立“属性图”。注意下面这行代码中第一个Graph是图数据类型，第二个Graph是图构建函数。

```
//基于以上顶点和边构建一个电信网络图

val telecommGraph: Graph[(String, String, String, Int), Int] = Graph(vertexRDD, edgeRDD)
```

3) 读取图中的数据

访问一个图对象中的数据有三种方法：访问节点、访问边以及访问Triplet。Triplet相当于把一条边和这条便两头的顶点做了一个join操作，可以同时获得节点和边携带的“属性”，其反映了两个节点之间的关系。三种访问方式的效果如下。



三种数据表示方式：节点、边和Triplet

```
//分别访问图的顶点、边、以及Triplet

scala> telecommGraph.vertices.collect.foreach(println)

(1,(Zhang Fei,Unicom,Male,36))

(2,(Li Zhi,CMCC,Female,18))

(3,(He Ruina,Unicom,Male,23))

(4,(Dong Xicheng,CMCC,Male,42))

(5,(Bai Zi,Unicom,Male,25))

(6,(Fu Ming,Unicom,Male,50))

(7,(Handel,ATT,Male,78))

(8,(Carmen,ATT,Female,21))

scala> telecommGraph.edges.collect.foreach(println)

Edge(2,1,4)

Edge(2,4,2)

Edge(3,2,5)

Edge(3,6,3)

Edge(4,1,1)

Edge(5,2,2)

Edge(5,3,8)

Edge(5,6,2)

Edge(7,6,9)

Edge(7,3,8)

Edge(8,7,3)

Edge(8,6,2)

scala> telecommGraph.triplets.collect.foreach(println)

((2,(Li Zhi,CMCC,Female,18)),(1,(Zhang Fei,Unicom,Male,36)),4)

((2,(Li Zhi,CMCC,Female,18)),(4,(Dong Xicheng,CMCC,Male,42)),2)

((3,(He Ruina,Unicom,Male,23)),(2,(Li Zhi,CMCC,Female,18)),5)

((3,(He Ruina,Unicom,Male,23)),(6,(Fu Ming,Unicom,Male,50)),3)

((4,(Dong Xicheng,CMCC,Male,42)),(1,(Zhang Fei,Unicom,Male,36)),1)
```

((5,(Bai Zi,Unicom,Male,25)),(2,(Li Zhi,CMCC,Female,18)),2)

((5,(Bai Zi,Unicom,Male,25)),(3,(He Ruina,Unicom,Male,23)),8)

((5,(Bai Zi,Unicom,Male,25)),(6,(Fu Ming,Unicom,Male,50)),2)

((7,(Handel,ATT,Male,78)),(6,(Fu Ming,Unicom,Male,50)),9)

((7,(Handel,ATT,Male,78)),(3,(He Ruina,Unicom,Male,23)),8)

((8,(Carmen,ATT,Female,21)),(7,(Handel,ATT,Male,78)),3)

((8,(Carmen,ATT,Female,21)),(6,(Fu Ming,Unicom,Male,50)),2)

4. GraphX基本操作

我们知道，Hadoop程序仅有Map和Reduce两种操作，具体数据分析功能都要封装到这两类操作中，这极大提高了程序开发的难度，并导致程序的可移植性较差。后发的Spark吸取了这方面的经验，在其基本数据类型RDD中封装了80多种预定义的数据操作方法，包括常用的filter、count、join、groupBy等，通常称为“算子”。而在GraphX中，扩展自RDD的属性图不仅可以调用RDD的全部算子，还新增了许多适用于图运算的方法，供开发人员根据分析需求使用或扩展，十分方便。属性图的一些常用方法列表如下。

图结构信息获取	
val numEdges: Long	计算边的总数
val numVertices: Long	计算顶点总数
val inDegrees: VertexRDD[Int]	计算所有节点的入度
val outDegrees: VertexRDD[Int]	计算所有节点的出度
val degrees: VertexRDD[Int]	计算所有节点的度
图属性访问	
val vertices: VertexRDD[VD]	以节点的形式访问图中数据
val edges: EdgeRDD[ED]	以边的形式访问图中数据
val triplets: RDD[EdgeTriplet[VD, ED]]	以 Triplet 的形式访问图中数据

图属性转换	
def mapVertices[VD2](map: (VertexID, VD) => VD2): Graph[VD2, ED]	对图中顶点、边或者 triplet 的属性进行映射和修改，每一个操作都会产生新图。常用于初始化图进行特殊计算或者排除不需要的属性。
def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]	
def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]	
图的重构	
def subgraph(epred: EdgeTriplet[VD,ED] => Boolean = (x => true), vpred: (VertexID, VD) => Boolean = ((v, d) => true)) : Graph[VD, ED]	求子图操作，通过对原图的边、节点定义筛选条件，从原图中提取一部分节点和边，构成一个新图

邻边聚合	
def aggregateMessages[Msg: ClassTag](sendMsg: EdgeContext[VD, ED, Msg] => Unit, mergeMsg: (Msg, Msg) => Msg, tripletFields: TripletFields = TripletFields.All) : VertexRDD[Msg]	邻边聚合，用于将相邻节点的信息按一定条件进行合并或运算。
基本图算法	
def pageRank(tol: Double, resetProb: Double = 0.15): Graph[Double, Double]	PageRank 算法，即网页排名算法
def connectedComponents(): Graph[VertexID, ED]	连通图算法，使用最小编号的顶点标记图的连通体。在社交网络中实现近似聚类效果
def triangleCount(): Graph[Int, ED]	三角形计数算法，其通过计算每一个顶点的三角形数量，实现对潜在集群的测量

上一节我们已经使用了获取图中节点、边和Triplet的属性的方法，下面用介绍另外几个常用函数的用法。

1) 求子图

顾名思义，该方法常用于图结构的简化，将原图按一定条件筛选出部分节点和边，构建一张只包含一部分的子图，通常子图的节点和边会比原图有所减少。

//从上面的电信网络中，筛选出只包含通话次数超过5次的边和顶点，构建子图

scala> telecommGraph.subgraph(epred = e => e.attr > 5).triplets.collect.foreach(println)

((5,(Bai Zi,Unicom,Male,25)),(3,(He Ruina,Unicom,Male,23)),8)

((7,(Handel,ATT,Male,78)),(6,(Fu Ming,Unicom,Male,50)),9)

((7,(Handel,ATT,Male,78)),(3,(He Ruina,Unicom,Male,23)),8)

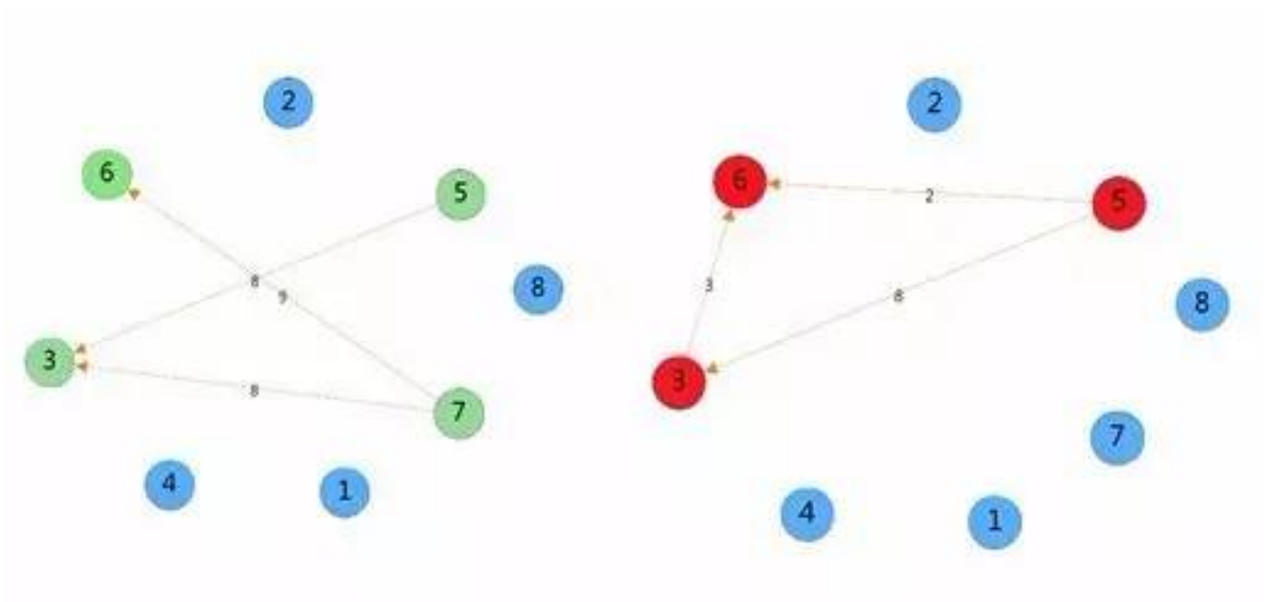
//筛选原图中通话双方均为为联通用户，且通话次数在1次以上的边和节点，构建子图

scala> telecommGraph.subgraph(vpred = (id,vd) => vd._2 == "Unicom", epred = e => e.attr > 1).triplets.collect.foreach(println)

((3,(He Ruina,Unicom,Male,23)),(6,(Fu Ming,Unicom,Male,50)),3)

((5,(Bai Zi,Unicom,Male,25)),(3,(He Ruina,Unicom,Male,23)),8)

((5,(Bai Zi,Unicom,Male,25)),(6,(Fu Ming,Unicom,Male,50)),2)



从原图得到的两个子图

2) 属性转换

属性转换通常是对原图的顶点和边的属性做映射操作，构建新图，但是原图的大小和结构不会改变。该类操作的目的通常是为了初始化图进行特殊计算或者排除不需要的属性。

//使用mapVerices函数对原图坐转换，转换后的节点只包含用户年龄属性

val newGraph = telecommGraph.mapVertices((id,vd) => vd._4)

scala> newGraph.vertices.collect.foreach(println)

(1,36)

(2,18)

(3,23)

(4,42)

(5,25)

(6,50)

(7,78)

(8,21)

3) 邻边聚合

邻边聚合是图分析中最常用到的操作之一，用于将相邻节点的信息按一定条件进行合并或运算。例如，统计某个用户的所有联系人数量，或者计算一个群体内平均通话时长等。此外，很多算法的实现也需要迭代地计算相邻节点的数值，例如PageRank算法中的PageRank值，Louvain算法中的Q值等。

在GraphX中，最核心的邻边聚合操作是aggregateMessages。这个操作首先需要用户定义一个消息发送函数到图中每一个triplet，类似于map操作；然后用合并消息函数在目的节点聚合这些信息，类似于reduce操作。下例中，我们统计每个用户拥有的年长者联系人人数及平均年龄

//在aggregateMessages中定义方法，迭代地比较两两节点的用户年龄，

//若发现年长联系人，由源节点向目标节点发送消息，并对人数和年龄进行加和

scala> val olderCallers: VertexRDD[(Int, Int)] = newGraph.aggregateMessages[(Int, Int)](

    | triplet => { if (triplet.srcAttr > triplet.dstAttr) { triplet.sendToDst(1, triplet.srcAttr)}},

    | (a,b) => (a._1 + b._1, a._2 + b._2))

//求平均年龄

scala> val avgAgeOfOlderCallers: VertexRDD[Double] =

    | olderCallers.mapValues((id, value) =>

    | value match { case (count, totalAge) => totalAge / count })

//查看结果：图中有4个用户拥有更年长的联系人，并计算了年长联系人的平均年龄

scala> avgAgeOfOlderCallers.collect.foreach(println)

```
(1,42.0)

(2,24.0)

(3,51.0)

(6,78.0)
```

4) 内置分析函数

GraphX提供了一套图算法工具包，如PageRank、数三角形、最大连通图等经典的图算法，方便用户对图进行分析。但是这些算法的代码实现目的和重点在于通用性。如果要获得最佳性能，需要参考其实现进行修改和扩展满足业务需求。另外，研读这些代码，也是理解GraphX编程最佳实践的好方法。还以上面构建的电信图为例，调用PageRank方法，找到被指向最多的节点。

```
//该函数需要传入一个参数，参数的值越小，PageRank计算的值就越精确

val ranks = telecommGraph.pageRank(0.0001).vertices

// 将结果与图顶点做join，获取用户姓名，进行结果呈现

val ranksByUsername = vertexRDD.join(ranks).map {
  case (id, (username, rank)) => (username._1, rank)
}

//查看结果，由于网络中节点数太少，此处结果仅仅反映了排序的趋势

scala> println(ranksByUsername.collect().mkString("\n"))

(Zhang Fei,0.523****9609375)

(Li Zhi,0.312****75)

(He Ruina,0.283****75)

(Dong Xicheng,0.282****84375)

(Bai Zi,0.15)

(Fu Ming,0.467****75)

(Handel,0.21375)

(Carmen,0.15)
```

5. Louvain社区发现算法的GraphX实现

Louvain算法是一种在社交网络中常用的社区发现算法，该算法易于实现，由其计算得到的社区结构是分层的，在并发环境下能够快速收敛，拥有百万级别以上节点处理能力。该算法细节请参考文章《社区发现算法初探》 。在开源社区中有不少基于GraphX的Louvain算法实现，感兴趣的同学可以在GitHub上下载源代码进行学**和改进，此处给出其中的一个链接：

https://github.com/Sotera/distributed-graph-analytics/tree/master/dga-graphx

这里截取其中的一段关键代码，其实现了Louvain算法中需要反复迭代的ΔQ值的计算。

```
 /**
  * Returns the change in modularity that would result from a vertex moving to a specified community.
  */
private def q(currCommunityId: Long, testCommunityId: Long, testSigmaTot: Long, edgeWeightInCommunity: Long, nodeWeight: Long,
internalWeight: Long, totalEdgeWeight: Long): BigDecimal = {
  val isCurrentCommunity = currCommunityId.equals(testCommunityId)
  val M = BigDecimal(totalEdgeWeight)
  val k_i_in_L = if (isCurrentCommunity) edgeWeightInCommunity + internalWeight else edgeWeightInCommunity
  val k_i_in = BigDecimal(k_i_in_L)
  val k_i = BigDecimal(nodeWeight + internalWeight)
  val sigma_tot = if (isCurrentCommunity) BigDecimal(testSigmaTot) - k_i else BigDecimal(testSigmaTot)
  var deltaQ = BigDecimal(0.0)
  if (!(isCurrentCommunity && sigma_tot.equals(BigDecimal.valueOf(0.0)))) {
    deltaQ = k_i_in - (k_i * sigma_tot / M)
  }
  deltaQ
 }
```

Louvain算法的具体代码实现无法在这里详细展开，本节重点介绍如何在Spark集群内使用以上该算法进行社区发现，方便读者快速上手。

1) 工程配置文件修改

使用IDEA或其他IDE新建工程并导入下载的源码。由于该模块使用gradle进行工程管理，首先编辑配置文件buidle.gradle，主要修改Spark的版本，以便让程序在自己的Spark集群上正常运行，见下图。其他依赖的包可同理进行相应修改，完成后IDE会自动从网络上的软件库中下载各种依赖关系。

```
dependencies {
    compile project(':dga-core')
    compile group: 'org.apache.spark', name: 'spark-core_2.10', version: '1.5.2'
    compile group: 'org.apache.spark', name: 'spark-graphx_2.10', version: '1.5.2'
    compile group: 'com.github.scopt', name: 'scopt_2.10', version: '3.2.0'
    compile group: 'com.typesafe', name:'config', version:'1.2.1'
    compile(
            [group: 'org.slf4j', name: 'slf4j-api', version: '1.6.6'],
            [group: 'org.slf4j', name: 'slf4j-log4j12', version: '1.6.6']
    )
```

2) 源码编译

在IDE中执行gradle的编译命令：

```
gradle dist
```

完成后左侧项目结构下多出了./build/dist这个路径，这就是我们需要运行的程序。将其拷贝到Spark集群内的任一节点上。

3) 修改运行脚本

我们使用Spark-yarn模式运行，编辑dist目录下名为dga-yran-graphx的脚本。打开该脚本可以发现其本质上是Spark任务提交命令spark-submit，其指定了软件的主程序，程序运行依赖的jar包，以及为该任务分配的executor数量、CPU核数、内存数等。当分析数据较大时，这些参数的设置将极大影响到程序运行效率，具体可参考Spark官方文档。

```
spark-submit \

--executor-cores 18 \

--num-executors 21 \

--executor-memory 62G \

--driver-memory 20G \

--class com.soteradefense.dga.graphx.DGARunner \

--master yarn-cluster \

--jars "lib/Spark-graphx_2.10-1.6.1.jar,lib/dga-core-0.0.1.jar,lib/config-1.2.1.jar,lib/scopt_2.10-3.2.0.jar" \

--files "conf/application.conf,conf/log4j.properties" \

lib/dga-graphx-0.1.jar "$@"
```

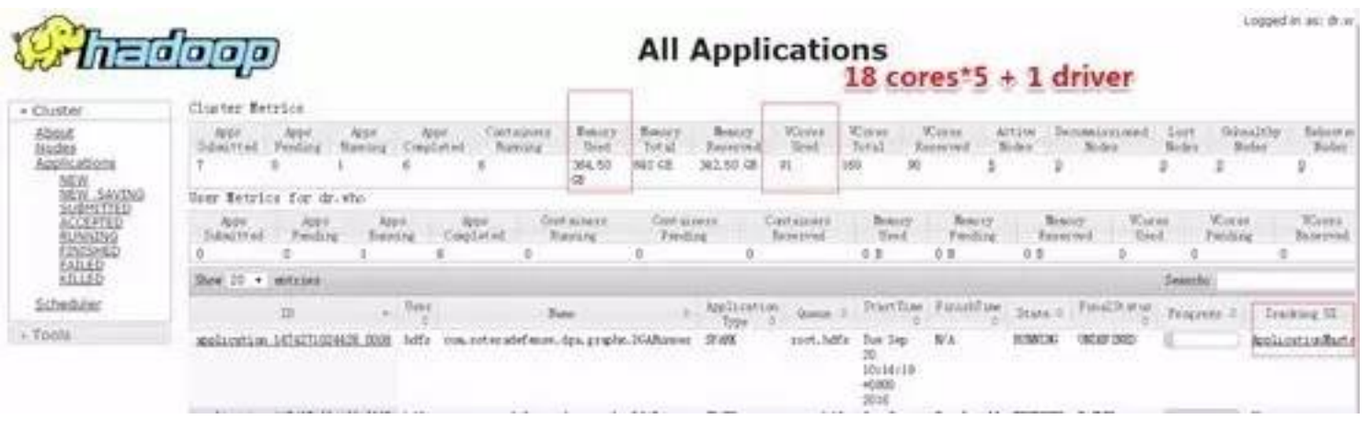
4) 执行Louvain算法

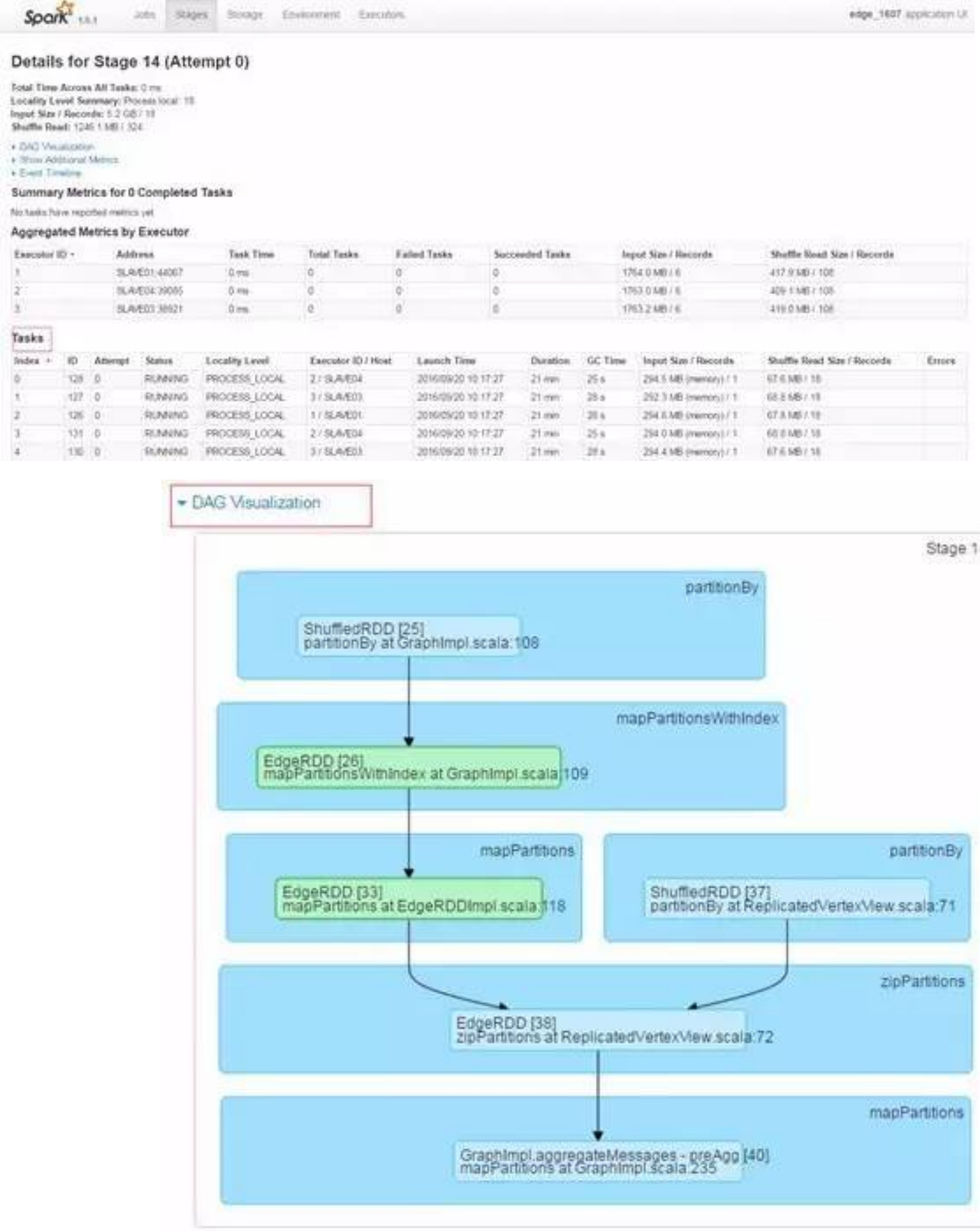
运行脚本命令，其中-i参数指定存储在HDFS上的边文件，-o参数指定输出结果存储的HDFS路径，-n是为程序在Spark中命名。

```
./dga-yarn-graphx louvain -i /user/oper/edge.csv -o /user/oper/out -n edge_1607
```

5) 程序监控

程序提交成功后，在YARN的监控界面中就可以看到该任务了，其消耗的CPU核数和内存数一目了然。而在Spark监控界面下，可依次查看为程序分配的Jobs、Stages、Tasks以及运行DAG图。这些页面提供了程序运行的流程细节和资源消耗，这些信息对程序的优化意义重大。





6) 结果查看

程序运行结果是一个路径，根据收敛情况其结果可能有多层，每层的结果包括一个边文件，一个节点文件以及本轮计算得到的Q值，根据边和节点可以构建每层计算完成后得到的新图。结果路径结构如下：

```
out /
├── level_0_edges
│   ├── _SUCCESS
│   └── part-00000
├── level_0_vertices
│   ├── _SUCCESS
│   └── part-00000
└── qvalues
    ├── _SUCCESS
    └── part-00000
```

打开其中节点文件，可以看到level_0生成的新图的所有节点，例如第一行中的数据代表某个编号为7的节点，属于一个编号为8的社区；而另一编号为4的节点属于编号为4的社区。

```
cat out/level_0_vertices/part-00000

(7,{community:8,communitySigmaTot:13,internalWeight:0,nodeWeight:3})

(4,{community:4,communitySigmaTot:21,internalWeight:0,nodeWeight:4})

(2,{community:4,communitySigmaTot:21,internalWeight:0,nodeWeight:4})

(6,{community:8,communitySigmaTot:13,internalWeight:0,nodeWeight:4})

(8,{community:8,communitySigmaTot:13,internalWeight:0,nodeWeight:3})
```

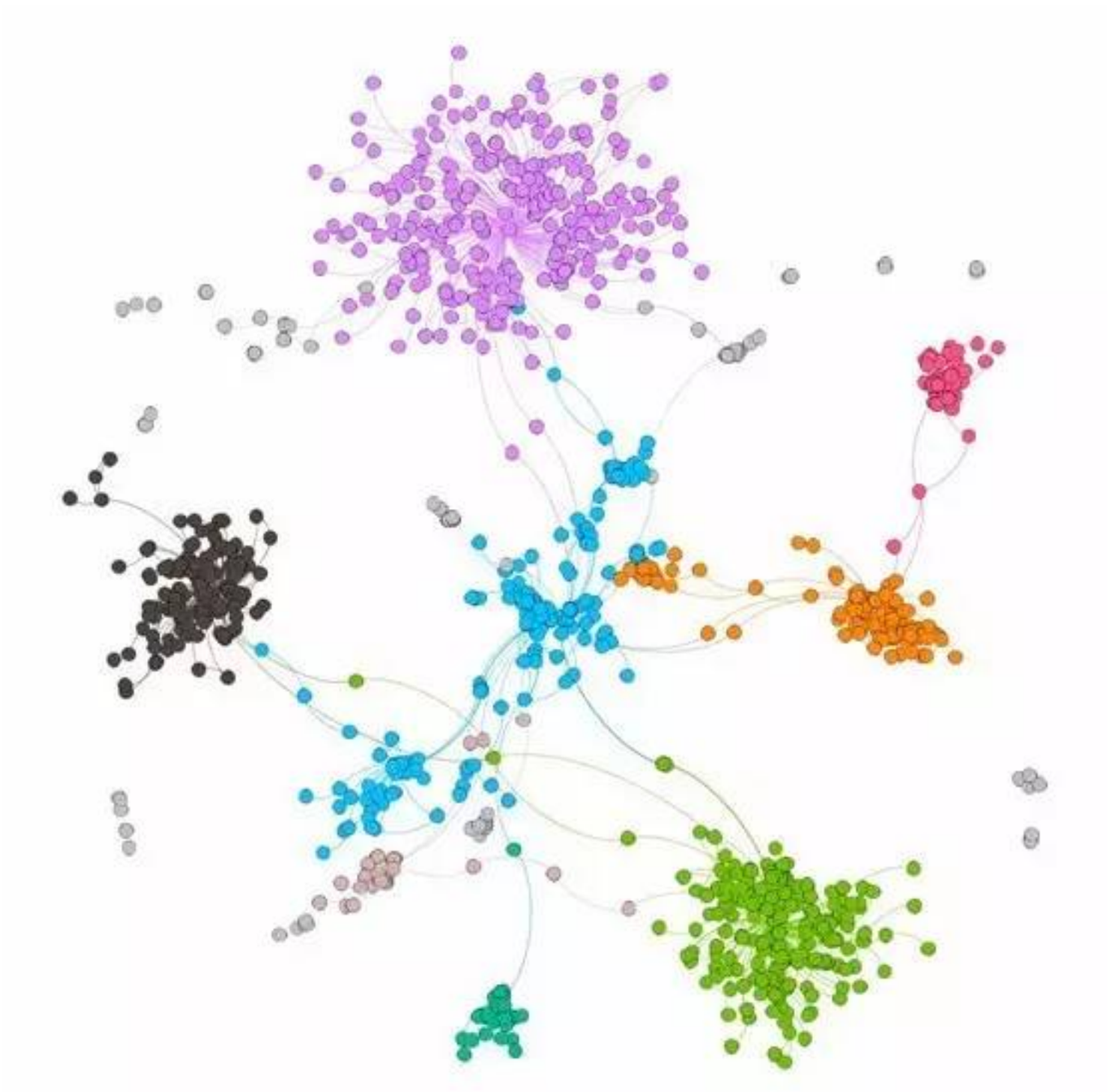
打开记录Q值的文件，可以看到这一轮生成的新图的Q值等于0.86。注意Q值（即Modularity）取值[0,1]，用于度量网络中各社区内部联系的强度，Q值越高的网络中，社区分类的效果越好。

```
cat out/qvalues/part-00000
```

(0,0.8607826****46949)

7) 结果可视化呈现

最后，我们可以使用开源的网络可视化工具Gephi渲染我们的社区发现结果，效果如下。



应用Louvain算法在网络中发现的社区

6. 小结

基于分布式计算框架Spark的图计算引擎GraphX，已经成为大数据挖掘领域不可或缺的一部分。例如，在某电商搭建的实时用户划分系统中，采用了“图流合璧”的方式：使用Spark Streaming和GraphX进行在线消息收集与实时图构建，根据业务需要对网站的用户进行实时社区划分，及时为不同用户群体进行商品推荐。

通过本文的介绍可以发现，运用Spark GraphX进行分布式的图计算涉及Spark分布式计算框架和GraphX图计算框架的使用，分布式图算法程序的开发，原始数据收集、处理与转换，开发环境与集群的搭建以及图结构的可视化呈现等诸多方面。尤其需要注意的是，虽然GraphX提供了一个极佳的图分析平台，但是具体的算法实现还是要根据业务需求、数据集大小、集群环境等对代码进行修改与扩展。正因如此，相关程序的开发者需要对分布式计算与图算法具备一定的理论深度和实践经验，如仅仅把它作为一个可以拿来就用的工具，直接套用自己数据上的做法并不可取，切记！受本人能力所限，本文仅对这两点的理解还不够深入，愿与大家共同学**交流。

-END-

推荐公众号