[9.1 图计算概述 2](#_Toc48839105)

[9.1.1 图的简单介绍 2](#_Toc48839106)

[9.1.2 传统图计算解决方案的不足之处 2](#_Toc48839107)

[9.1.3 图计算通用软件 3](#_Toc48839108)

[9.2 Spark GraphX 4](#_Toc48839109)

[9.2.1 属性图 4](#_Toc48839110)

[1. 介绍 4](#_Toc48839111)

[2. 构建方法 5](#_Toc48839112)

[9.2.2 顶点RDD、边RDD以及三元组RDD 6](#_Toc48839113)

[1. 顶点RDD 6](#_Toc48839114)

[2. 边RDD 14](#_Toc48839115)

[3. 边三元组RDD 17](#_Toc48839116)

[4. 实例 18](#_Toc48839117)

[9.2.3 图操作 21](#_Toc48839118)

[1. 操作列表概要 21](#_Toc48839119)

[2. 属性操作 22](#_Toc48839120)

[3. 结构操作 26](#_Toc48839121)

[4. 连接操作 30](#_Toc48839122)

[5. 聚合操作（邻居之间） 32](#_Toc48839123)

[6. 缓存与取消缓存操作 35](#_Toc48839124)

[9.3 具体实例——Dijstra单源最短路径算法 36](#_Toc48839125)

[9.3.1 Dijstra求单源最短路径算法介绍 36](#_Toc48839126)

[9.3.2 Dijstra算法实例 36](#_Toc48839127)

[9.4 图算法——PageRank算法实例 40](#_Toc48839128)

[9.4.1 PageRank原理介绍 40](#_Toc48839129)

[9.4.2 PageRank实例 41](#_Toc48839130)

[9.5 Pregel API 43](#_Toc48839131)

[9.5.1 Pregel的计算过程 43](#_Toc48839132)

[9.5.2 Spark GraphX的Pregel操作与标准Pregel的不同 44](#_Toc48839133)

[9.5.3 Pregel API的实现草图 45](#_Toc48839134)

[9.5.4 Pregel API实例——单源最短路径 46](#_Toc48839135)

**第9章 Spark GraphX**

在大数据时代，许多大数据都是以大规模图或网络的形式呈现，此外许多非图结构的大数据也常常会被转换为图模型后再进行处理分析，随着图规模的逐渐增大，如何高效地处理图数据成为所面临的主要挑战。GraphX是Spark专门用于处理图以及图并行计算的新组件，实际上通过抽象出Graph来扩展了Spark RDD。

本章内容会从介绍图计算开始，接着讲述Spark GraphX的属性图、扩展的RDDs以及一些基本图操作，然后介绍下更适用于迭代计算的Pregel API，最终介绍一个经典图算法——PageRank的实例应用。

9.1 图计算概述

在实际应用中，存在许多图计算问题，如最短路径、集群、网页排名、最小切割、连通分支等。图计算算法的性能直接关系到应用问题解决的高效性，尤其对于大型图（如社交网络和网络图）而言，更是如此。下面进行图计算的具体介绍。

9.1.1 图的简单介绍

图的基本组成就是顶点与边，是由有穷非空顶点集合和边集合组成的，通常表示为G(V,E),其中，V是图G中顶点的集合，E是图G中边的集合，G则代表一个图。

接下来的图计算介绍中主体是有向图，有向图即为由有向边构成的图，有向边则是具有方向的边，即具有源顶点与目标顶点且方向从源顶点指向目标顶点的边。

9.1.2 传统图计算解决方案的不足之处

很长一段时间内，一直都缺少一个可扩展的通用系统来解决大型图的计算问题。很多传统的图计算算法都存在以下几个典型问题：常常表现出比较差的内存访问局部性；针对单个顶点的处理工作过少；计算过程中伴随着并行度的改变。

针对大型图（如社交网络和网络图）的计算问题，可能的解决方案及其不足之处具体如下。

① 为特定的图应用定制相应的分布式实现。不足之处是通用性不好，在面对新的图算法或者图表示方式时，就需要做大量的重复开发。

② 基于现有的分布式计算平台进行图计算。比如，MapReduce作为一个优秀的大规模数据处理框架，有时也能够用来对大规模图对象进行挖掘，不过在性能和易用性方面往往无法达到最优。

③ 使用单机的图算法库，比如BGL、LEAD、NetworkX、JDSL、Standford GraphBase和FGL等。但是，这种单机方式在可以解决的问题的规模方面具有很大的局限性。

④ 使用已有的并行图计算系统。Parallel BGL和CGM Graph这些库实现了很多并行图算法，但是对大规模分布式系统非常重要的一些方面（如容错），无法提供较好的支持。

9.1.3 图计算通用软件

正是因为传统的图计算解决方案无法解决大型图的计算问题，因此就需要设计能够用来解决这些问题的通用图计算软件。针对大型图的计算，目前通用的图处理软件主要包括两种：第一种主要是基于遍历算法的、实时的图数据库，如Neo4j、OrientDB、DEX和InfiniteGraph；第二种则是以图顶点为中心的、基于消息传递批处理的并行引擎，如Hama、GoldenOrb、Giraph和Pregel。

第二种图处理软件主要是基于BSP模型实现的并行图处理系统。BSP是由哈佛大学Viliant和牛津大学Bill Mc Coll提出的并行计算模型，全称为“整体同步并行计算模型”（Bulk Synchronous Parallel Computing Model，BSP模型），又名“大同步模型”。创始人希望BSP模型像冯·诺依曼体系结构那样，架起计算机程序语言和体系结构间的桥梁，故又称为“桥模型”。一个BSP模型由大量通过网络相互连接的处理器组成，每个处理器都有快速的本地内存和不同的计算线程，一次BSP计算过程包括一系列全局超步（超步就是指计算中的一次迭代），每个超步主要包括3个组件。

① 局部计算。每个参与的处理器都有自身的计算任务，它们只读取存储在本地内存中的值，不同处理器的计算任务都是异步并且独立的。

② 通信。处理器群相互交换数据，交换的形式是，由一方发起推送（Put）和获取（Get）操作。

③ 栅栏同步（Barrier Synchronization）。当一个处理器遇到“路障”（或栅栏），会等其他所有处理器完成它们的计算步骤；每一次同步也是一个超步的完成和下一个超步的开始。一个超步的垂直结构图如图9-1所示。

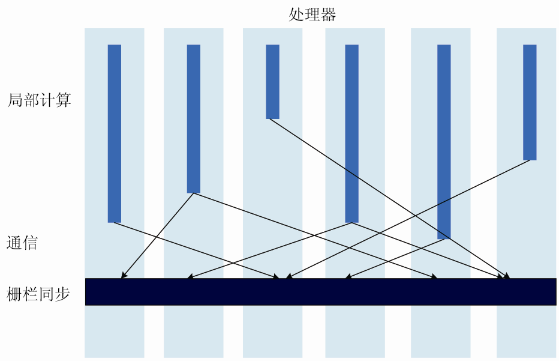


图9-1 一个超步的垂直结构

9.2 Spark GraphX

9.2.1 属性图

* 1. 介绍

GraphX中抽象出了Graph类来表示属性图，类中定义了对图的一些操作，属性图是有向图，包含顶点和边，其中每个顶点都有一个64位长标识符键，且该键是唯一的，类型为VertexId；属性图支持存在多条平行边（源顶点和目标顶点都一样），这是为了方便表示相同顶点之间可能存在的多种关系，例如小明和小王两人可能既是同学又是朋友，与顶点拥有标识符一样，边同样也拥有源顶点与目标顶点的标识符。

下面介绍下图类中包含的顶点与边：

class Graph[VD, ED] {

val vertices: VertexRDD[VD] // 顶点

val edges: EdgeRDD[ED] // 边

val triplets: RDD[EdgeTriplet[VD, ED]] // 集合顶点与边属性的三元组

}

上述代码中可以看出图的顶点的类型是VertexRDD[VD]，边则是EdgeRDD[ED]，而三元组则是RDD[EdgeTriplet[VD, ED]]，三者均是RDD集合，其中顶点和边的类型是由特定类型的RDD扩展得到的，三元组的类型EdgeTriplet则是由Edge类扩展的，下一节我们会重点讨论这个。其中VD表示的就是顶点的属性类型，ED则是边的属性类型，属性图中这些类型是可以用户自己定义的，通过更改顶点（VD）和边（ED）的类型即可。如果是原始数据类型（int，float等），GraphX会将它们存储在专用数组，这是为了减少内存占用。

如果需要同一图中的顶点属性类型不尽相同，可以采用继承的方法，继承一个基类，然后将你需要的属性类型扩展上去即可，例如需要用户和产品两种不同属性类型的顶点，可以参考如下操作：

class VertexProperty()

// 用户的顶点类型就包含一个String类，表示名字

case class UserProperty(val name: String) extends VertexProperty

// 而产品除了有名字还有价格，所以会增加一个double类

case class ProductProperty(val name: String, val price: Double) extends VertexProperty

// 图类中顶点的类型为基类VertexProperty

var graph: Graph[VertexProperty, String] = null

实际上属性图就是RDD的集合（顶点与边都是RDD集合），只是这些RDD是特定类型的，所以属性图也有RDD的性质，遇到故障时可以在不同计算机上重新创建出原来的每个分区，前面我们讲过RDD的转换操作是基于现有的数据集创建一个新的数据集，在Graph中对顶点RDD与边RDD的操作也同样是产生新的RDD，需要通过生成一个新图来成功进行更新，所以对图进行值或者结构更改都是生成新图，原来不改变的部分直接重新使用，如此一来可以降低成本。

* 1. 构建方法

object Graph {

def apply[VD, ED](

vertices: RDD[(VertexId, VD)],

edges: RDD[Edge[ED]],

defaultVertexAttr: VD = null)

: Graph[VD, ED]

def fromEdges[VD, ED](

edges: RDD[Edge[ED]],

defaultValue: VD): Graph[VD, ED]

def fromEdgeTuples[VD](

rawEdges: RDD[(VertexId, VertexId)],

defaultValue: VD,

uniqueEdges: Option[PartitionStrategy] = None): Graph[VD, Int]

}

1. apply方法通过传入RDD[(VertexId, VD)]和RDD[Edge[ED]]来构造图，如果在RDD[Edge[ED]]中存在的顶点在RDD[(VertexId, VD)]中不存在，那么该顶点用defaultVertexAttr来赋予默认属性值，所以有时候我们需要提前设置好默认值，以防出现源顶点/目标顶点不存在的边。构建图默认使用apply这个方法。
2. fromEdges方法可仅通过传入的RDD[Edge[ED]]来构造图，其中自动创建的顶点则是在RDD[Edge[ED]]中出现的，且使用defaultValue来赋予默认属性值。
3. fromEdgeTuples方法通过传入的RDD[(VertexId, VertexId)]来构造图，所有边的属性值默认为1，自动创建在RDD[(VertexId, VertexId)]中出现的顶点并使用defaultValue赋予默认属性值。
4. 除了上述方法外，还可使用GraphGenerators随机生成一张图或是使用GraphLoader.edgeListFile方法通过读取文件来构建图，若使用GraphLoader.edgeListFile方法的话，文件内容格式如下：

# 文件内容

源顶点id 目标顶点id

......

# 文件内容

主要读取顶点对，构建的图所有的顶点与边的属性值均默认为1。

9.2.2 顶点RDD、边RDD以及三元组RDD

* 1. 顶点RDD

图类中的VertexRDD[VD]继承了RDD[(VertexId, VD)]，VertexId是每个顶点唯一的标识符类型，而VD则是与顶点关联的对象类型（属性类型），其实基础就是二元组(标识符，属性类型)的顶点RDD，在此基础上又扩展了对顶点的一些操作构成VertexRDD[VD]，省略了标识符，可以看成属性类型为VD的顶点RDD以及作用在上面的相关操作的集成，下表涵盖部分简单操作。

表9-1 VertexRDD操作表

|  |  |
| --- | --- |
| 操作 | 含义 |
| filter | 将符合一定条件的顶点筛选出来，返回筛选后的VertexRDD |
| mapValues | 更改顶点属性类型，返回更改后的VertexRDD |
| minus | 需要另一个顶点集，返回的是原图中独有的顶点集 |
| diff | 需要另一个顶点集，返回的是两图中不同属性值的顶点集 |
| leftJoin | 根据原图顶点id进行左连接操作，返回新的VertexRDD |
| innerJoin | 根据两图共有顶点id进行连接操作，返回新的VertexRDD |
| aggregateUsingIndex | 将传入的信息RDD中相同顶点id的属性值进行聚合操作，返回新的VertexRDD |

首先我们可以先进入spark-shell交互式环境中，执行以下命令：

$ cd /usr/local/spark

$ ./bin/spark-shell

显示出



即为成功

开始之前先导入所需要的包文件，命令行中执行：

scala> import org.apache.spark.\_

scala> import org.apache.spark.graphx.\_

scala> import org.apache.spark.rdd.RDD

接着我们需要构成一张图便于后面的命令练习，比如：

scala> val sc = new SparkContext()

// 设置users顶点

scala> val users: RDD[(VertexId, (String, String))] =

sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),

(5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))

// 设置relationships边

scala> val relationships: RDD[Edge[String]] =

sc.parallelize(Array(Edge(3L, 7L, "collab"), Edge(5L, 3L, "advisor"),

Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))

// 定义默认的作者,以防与不存在的作者有relationship边

scala> val defaultUser = ("John Doe", "Missing")

// 构建图

scala> val graph = Graph(users, relationships, defaultUser)

图结构如图9-2所示：

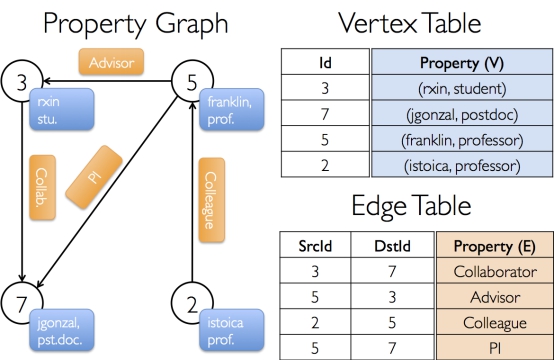


图9-2 graph的图结构

进行顶点RDD的相关操作前我们可以先查看一下新创建图中的顶点RDD信息：

执行：

scala> val verticesrdd = graph.vertices.collect.foreach(println(\_))

返回结果：

(5,(franklin,prof))

(2,(istoica,prof))

(3,(rxin,student))

(7,(jgonzal,postdoc))

verticesrdd: Unit = ()

下面介绍部分操作：

* filter(pred: Tuple2[VertexId, VD] => Boolean): VertexRDD[VD]

filter操作主要是将所有顶点遍历（用case解析结构（VertexId，VD）），将满足=>后面的条件筛选出来，返回的是筛选出的结果。

执行：

scala> val a = graph.vertices.filter { case (id, (name, pos)) => pos == "prof" }

返回结果：

a: Array[(org.apache.spark.graphx.VertexId, (String, String))] = Array((5,(franklin,prof)), (2,(istoica,prof)))

我们打算将职业为prof的选出来。如果想输出美观，可以在collect后加上.collect.foreach(println(\_))。

返回结果：

(5,(franklin,prof))

(2,(istoica,prof))

过程如下图所示：



图9-3 VertexRDD的filter操作

* mapValues[VD2](map: (VertexId, VD) => VD2): VertexRDD[VD2]

mapValues用来改变所有顶点关联对象的类型（属性类型），将原来的VD改为了VD2，返回的是更改后新的VertexRDD。

执行：

scala> val a = graph.vertices.mapValues((vid:VertexId, attr:(String, String)) => (attr.\_1 +" is a "+attr.\_2))

scala> a.collect.foreach(println(\_))

将属性类型从（String，String）改成了String，属性值改成了“name is a pos”这种形式。

返回结果：

(5,franklin is a prof)

(2,istoica is a prof)

(3,rxin is a student)

(7,jgonzal is a postdoc)

过程如下图所示：



图9-4 VertexRDD的mapValues操作

* minus(other: RDD[(VertexId, VD)])

执行minus需要另一个顶点集，假设是users1，返回的是当前vertices独有id的顶点集（顶点id不在另一个图users1中出现）。

首先创建一个RDD集users1：

scala> val users1: RDD[(VertexId, (String, String))] =

sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),

(5L, ("franklin", "prof")), (1L, ("istoica", "prof"))))

然后执行：

scala> val a = graph.vertices.minus(users1)

scala> a.collect.foreach(println(\_))

可以看出vertices中只有id为2的顶点是vertices有而users1没有的。

返回结果：

(2,(istoica,prof))

过程如下图所示：



图9-5 VertexRDD的minus操作

* diff(other: VertexRDD[VD]): VertexRDD[VD]

执行diff同样也需要另一个顶点集，假设是users1，返回的是当前vertices与users1不同属性值的顶点集，且返回的属性值是users1中的。

首先创建一个RDD集users1：

scala> val users1: RDD[(VertexId, (String, String))] =

sc.parallelize(Array((3L, ("wyn", "student")), (7L, ("jgonzal", "postdoc")),

(5L, ("franklin", "prof")), (1L, ("istoica", "prof"))))

然后执行：

scala> val a = graph.vertices.diff(users1)

scala> a.collect.foreach(println(\_))

可以看出vertices与users1之间属性值不同的顶点是(“rxin”,”student”)与(“wyn”,”student”)，所以返回的应该是后者。

返回结果：

(3,(wyn,student))

过程如下图所示：



图9-6 VertexRDD的diff操作

* leftJoin[VD2, VD3](other: RDD[(VertexId, VD2)])(f: (VertexId, VD, Option[VD2]) => VD3): VertexRDD[VD3]

leftjoin根据顶点id进行左连接操作，以vertices的id为准，可以将两个顶点集的属性合并起来或是形成新的属性，所以我们也需要创建一个新的顶点集：

scala> val users1: RDD[(VertexId, String)] =

sc.parallelize(Array((3L, "female"), (7L, "male"),

(5L, "female"), (1L, "male")))

执行：

scala> val a = graph.vertices.leftJoin(users1){

(vid,left,right) => (left.\_1,left.\_2,right.getOrElse("none"))}

scala> a.collect.foreach(println(\_))

可以看出users1中没有顶点id为2的信息，那么在左连接操作中id为2的性别就是”none”。

返回结果：

(5,(franklin,prof,female))

(2,(istoica,prof,none))

(3,(rxin,student,female))

(7,(jgonzal,postdoc,male))

过程如下图所示：



图9-7 VertexRDD的leftJoin操作

* innerJoin[U, VD2](other: RDD[(VertexId, U)])(f: (VertexId, VD, U) => VD2): VertexRDD[VD2]

innerJoin与leftJoin最大的区别在于，返回的结果只会有两者共出现的顶点id信息，而leftJoin是以vertices的键值为准进行连接。

在上一个的基础上，执行：

scala> val a = graph.vertices.innerJoin(users1){

(vid,left,right) => (left.\_1,left.\_2,right)}

scala> a.collect.foreach(println(\_))

所以同样的users1顶点集，这次不会返回顶点id为2的信息了。

返回结果：

(5,(franklin,prof,female))

(3,(rxin,student,female))

(7,(jgonzal,postdoc,male))

过程如下图所示：



图9-8 VertexRDD的innerJoin操作

* aggregateUsingIndex[VD2](other: RDD[(VertexId, VD2)], reduceFunc: (VD2, VD2) => VD2): VertexRDD[VD2]

aggregateUsingIndex操作其实就是在计算信息，传入的消息RDD内容就是在向每个顶点发送消息，例如(5,2)表示对顶点5发送消息2，其中相同顶点id的属性值进行聚合操作，这个聚合操作用户可以自己定义，聚合结果的类型与原来应保持一致，聚合操作定义就是上面通式中的reduceFunc，需要注意的是，返回结果只含有接受到消息的顶点。

首先创建一个传来的信息RDD：

scala> val message: RDD[(VertexId, Int)] =

sc.parallelize(Array((5L, 2), (2L, 6),

(2L, 1), (4L, 8)))

执行：

scala> val a = graph.vertices.aggregateUsingIndex(message, (vd1:Int,vd2:Int) => vd1+vd2)

scala> a.collect.foreach(println(\_))

原图中不存在id为4的顶点，所以不会返回结果不会有id为4的消息；由于id为3和7的顶点未接受到消息，所以也不会返回；消息RDD中存在两个顶点id均为2的元组，分别是(2L, 6)与(2L, 1)，聚合操作定义为属性值相加，所以返回结果之一会有(2,7)。

返回结果：

(5,2)

(2,7)

过程如下图所示：



图9-9 VertexRDD的aggregateUsingIndex操作

* 1. 边RDD

类中的EdgeRDD[ED]继承了RDD[Edge[ED]]，ED是与边关联的对象类型（属性类型），基础就是类型为Edge[ED]的边RDD，其中Edge[ED]具体如下：

class Edge[ED] (srcId: VertexId , dstId: VertexId , attr: ED )

边类中有三个成员变量，其中成员srcId与dstId分别是源顶点与目标顶点，类型均为VertexId；成员attr指边所关联的对象属性值，对象类型是ED，Edge[ED]其实就是表示属性类型为ED的边类。

在此基础上又扩展了对边的一些操作构成EdgeRDD[ED]，可以看成属性类型为ED的边RDD以及作用在上面的相关操作的集成，下表涵盖部分简单操作。

表9-2 EdgeRDD操作

|  |  |
| --- | --- |
| 操作 | 含义 |
| mapValues | 更改边属性类型，返回更改后的EdgeRDD |
| reverse | 将图中所有边进行反向操作，返回更改后的EdgeRDD |
| innerJoin | 基于源顶点与目标顶点，对两个边RDD进行连接操作，返回新的EdgeRDD |

进行边RDD的相关操作前我们可以先查看一下图中的边RDD信息：

执行：

scala> val edgesrdd = graph.edges.collect.foreach(println(\_))

返回结果：

Edge(3,7,collab)

Edge(5,3,advisor)

Edge(2,5,colleague)

Edge(5,7,pi)

edgesrdd: Unit = ()

下面介绍部分操作：

* mapValues[ED2](f: Edge[ED] => ED2): EdgeRDD[ED2]

边的mapValues与顶点的操作类似，都是将更改所有边的属性类型，将原来的ED改为了ED2，返回的是更改后新的EdgeRDD。

执行：

scala> val a = graph.edges.mapValues( e => ("relationship",e.attr))

scala> a.collect.foreach(println(\_))

将属性类型从String改成了（String，String），属性值改成了(”relationship”, attr)这种形式。

返回结果：

Edge(3,7,(relationship,collab))

Edge(5,3,(relationship,advisor))

Edge(2,5,(relationship,colleague))

Edge(5,7,(relationship,pi))

过程如下图所示：



图9-10 EdgeRDD的mapValues操作

* reverse: EdgeRDD[ED]

reverse操作将所有边方向进行反向，也就是源顶点和目标顶点会调换。

执行：

scala> val a = graph.edges.reverse

scala> a.collect.foreach(println(\_))

返回结果：

Edge(7,3,collab)

Edge(3,5,advisor)

Edge(5,2,colleague)

Edge(7,5,pi)

过程如下图所示：



图9-11 EdgeRDD的reverse操作

* innerJoin[ED2, ED3](other: EdgeRDD[ED2])(f: (VertexId, VertexId, ED, ED2) => ED3): EdgeRDD[ED3]

innerJoin操作也是类似，连接操作主要基于源顶点与目标顶点，以此为键对两个边RDD进行连接操作。首先创建另一个EdgeRDD（需要注意，在这个地方需要的是EdgeRDD，而不是RDD[Edge[ED]]，所以不再是单纯创建一个RDD那么简单，RDD[Edge[ED]]与EdgeRDD的区别详见这节的开头描述）：

首先先创建一个RDD[Edge[ED]]：

scala> val relationships1: RDD[Edge[String]] =

sc.parallelize(Array(Edge(3L, 7L, "good"), Edge(3L, 5L, "good"),

Edge(2L, 5L, "good"), Edge(5L, 7L, "bad")))

用来描述关系好坏，其中只有Edge(3L, 5L, "good")与原来的源顶点、目标顶点不一致。

创建一个新图graph1，顶点可以用原来的users：

scala> val graph1:Graph[(String, String), String] = Graph(users, relationships1)

创建完图后，使用.edges便是 EdgeRDD类型：

scala> val edges1 = graph1.edges

执行：

scala> val a = graph.edges.innerJoin(edges1){(sid,did,left,right) => (left,right)}

scala> a.collect.foreach(println(\_))

连接操作只会将分别有相同源顶点与目标顶点的两方属性值进行连接，连接方式将属性值拼接成(left,right)形式，变成(String,String)类型。

返回结果：

Edge(3,7,(collab,good))

Edge(2,5,(colleague,good))

Edge(5,7,(pi,bad))

过程如下图所示：



图9-12 EdgeRDD的innerJoin操作

* 1. 边三元组RDD

边三元组RDD类型是RDD[EdgeTriplet[VD, ED]]，也就是类型为EdgeTriplet[VD, ED]的RDD集合，其中EdgeTriplet类型是从边类Edge继承得来的，边三元组的添加的具体信息可以参考图9-13：



图9-13 边三元组RDD信息结构

在Edge的基础上主要添加了成员变量srcAttr与dstAttr，类型均为顶点所关联的对象类型VD，分别表示源顶点与目标顶点的关联对象的属性值。例如利用map函数遍历边三元组来生成新的RDD（map函数是RDD本身就有的，并不是边三元组RDD独有的函数）：

执行：

scala> val sentences = graph.triplets.map(triplet =>

| triplet.srcAttr.\_1 + " is the " + triplet.attr + " of " + triplet.dstAttr.\_1)

scala> sentences.collect.foreach(println(\_))

上述的操作将边两方的人物关系进行描述，边三元组triplets可以直接获取源顶点与目标顶点的属性值，所以读取很方便。

返回结果：

rxin is the collab of jgonzal

franklin is the advisor of rxin

istoica is the colleague of franklin

franklin is the pi of jgonzal

过程如下图所示：



图9-14 对EdgeTripletRDD进行map操作

* 1. 实例

下面是一个属性图的小实例，图结构与前面练习的一样。

1. 首先我们需要我们需要进入/usr/local/spark/mycode目录下：

$ cd /usr/local/spark/mycode

若显示没有该目录，则我们需要创建一个（sudo mkdir /usr/local/spark/mycode）

进入目录后在graphx/src/main/scala目录下创建scala文件SimpleGraphX.scala：

$ mkdir -p graphx/src/main/scala

$ vim graphx/src/main/scala/SimpleGraphX.scala

1. 在新建的SimpleGraphX.scala文件中按键盘“i”进行输入，输入以下内容：（主要用了filter的方法）

import org.apache.log4j.{Level,Logger}

import org.apache.spark.\_

import org.apache.spark.graphx.\_

import org.apache.spark.rdd.RDD

object SimpleGraphX {

def main(args: Array[String]) {

//屏蔽日志

Logger.getLogger("org.apache.spark").setLevel(Level.WARN)

Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

//设置运行环境

val conf = new SparkConf().setAppName("SimpleGraphX").setMaster("local")

val sc = new SparkContext(conf)

//设置users顶点

val users: RDD[(VertexId, (String, String))] =

sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")), (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))

//设置relationships边

val relationships: RDD[Edge[String]] =

sc.parallelize(Array(Edge(3L, 7L, "collab"),Edge(5L, 3L, "advisor"), Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))

// 定义默认的作者,以防与不存在的作者有relationship边

val defaultUser = ("John Doe", "Missing")

// Build the initial Graph

val graph = Graph(users, relationships, defaultUser)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*展示图的属性

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

println("属性展示")

println("---------------------------------------------")

println("找到图中属性是student的顶点")

graph.vertices.filter { case (id, (name, occupation)) => occupation=="student"}.collect.foreach {

case (id, (name, occupation)) => println(s"$name is $occupation")

}

println("---------------------------------------------")

println("找到图中边属性是advisor的边")

graph.edges.filter(e => e.attr == "advisor").collect.foreach(e => println(s"${e.srcId} to ${e.dstId} att ${e.attr}"))

println("---------------------------------------------")

println("找出图中最大的出度、入度、度数：")

def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {

if (a.\_2 > b.\_2) a else b

}

println("max of outDegrees:" + graph.outDegrees.reduce(max) + " max of inDegrees:" + graph.inDegrees.reduce(max) + " max of Degrees:" + graph.degrees.reduce(max))

}

}

这段代码主要是创建了一个图，然后过滤出特定信息的顶点以及边并输出，同时还输出最大出入度以及度数的信息。

输入后按“Esc”并输入:wq退出，如果显示是只读文件需要强制执行则输入:wq!

1. 退出后返回graphx文件下，创建simple.sbt文件，添加依赖信息：

$ cd /usr/local/spark/mycode/graphx

$ vim simple.sbt

在新建的文件中添加如下依赖信息：

name := "Simple Project"

version := "1.0"

scalaVersion := "2.11.12"

libraryDependencies += "org.apache.spark" %% "spark-core" % "2.4.0"

libraryDependencies += "org.apache.spark" %% "spark-graphx" %"2.4.0"

保存方法与之前一样。

1. 退出后用sbt编译打包（一定要保证在/usr/local/spark/mycode/graphx目录下）：

$ /usr/local/sbt/sbt package

下面会有输出信息，看到

[success] Total time: 5 s, completed 2020-8-5 17:12:22

即为编译打包成功。生成的jar包位置在./target/scala-2.11/simple-project\_2.11-1.0.jar

1. 接下来用spark-submit运行程序

最后我们可以通过spark-submit将jar包提交到Spark中运行了，命令如下：

$ /usr/local/spark/bin/spark-submit \

> --class "SimpleGraphX" \

> /usr/local/spark/mycode/graphx/target/scala-2.11/simple-project\_2.11-1.0.jar

执行后前面会有很多输出信息，直接看最后输出的结果，最终得到的结果如下：

属性展示

---------------------------------------------

找到图中属性是student的顶点

rxin is student

---------------------------------------------

找到图中边属性是advisor的边

5 to 3 att advisor

---------------------------------------------

找出图中最大的出度、入度、度数：

max of outDegrees:(5,2) max of inDegrees:(7,2) max of Degrees:(5,3)

结果正确显示，说明程序运行成功。

9.2.3 图操作

* 1. 操作列表概要

接下来我们具体了解一下图操作，表9-3是一些基本图操作的总表。

表9-3 Graph操作

|  |  |
| --- | --- |
| 操作 | 含义 |
| mapVertices | 更改顶点属性类型，返回更改后的新图 |
| mapEdges | 更改边属性类型，返回更改后的新图 |
| mapTriplets | 通过遍历三元组更改边属性类型，返回更改后的新图 |
| reverse | 将图中所有边进行反向操作，返回更改后的新图 |
| subgraph | 构造子图，筛选出满足条件的顶点、边组成新图，返回新图 |
| mask | 选取两个图共同的顶点和边组成新图，属性值与原图相同，返回新图 |
| groupEdges | 同一个顶点对之间的多重边合并为一条边，返回合并后新图 |
| joinVertices | 与外部RDD集进行连接来更改顶点属性值，但需保持属性类型与原来一致，返回更改后新图 |
| outerJoinVertices | 与joinVertices类似，不同的是可以忽略原来属性类型，且对于未出现在另一RDD集中的顶点，可以自定义更改其属性值 |
| aggregateMessages | 用于处理消息聚合，含有发送消息、聚合计算的函数 |
| pregel | PregelAPI |
| pageRank | 经典图算法——PageRank的API |

* 1. 属性操作

这类操作类似于RDD的map操作，用于修改顶点与边的对象类型，需要注意的是，每次运算都是产生一个新图，所以以下两段代码其实可以看作是等价的：（mapUdf为用户自定义函数）

// RDD进行map后生成新的RDD，需要通过生成新图完成更新

val newVertices = graph.vertices.map { case (id, attr) => (id, mapUdf(id, attr)) }

val newGraph = Graph(newVertices, graph.edges)

// 用mapVertices与上面效果同样，但它可以重用原来的结构，返回的直接是新图Graph类而不是新RDD

val newGraph = graph.mapVertices((id, attr) => mapUdf(id, attr))

下面介绍一些属性操作：

与之前的操作一致，创建图也与之前的一致，首先我们可以先进入spark-shell交互式环境中，执行以下命令：

$ cd /usr/local/spark

$ ./bin/spark-shell

显示出



即为成功

开始之前先导入所需要的包文件，命令行中执行：

scala> import org.apache.spark.\_

scala> import org.apache.spark.graphx.\_

scala> import org.apache.spark.rdd.RDD

接着我们需要构成一张图便于后面的命令练习，比如：

scala> val sc = new SparkContext()

// 设置users顶点

scala> val users: RDD[(VertexId, (String, String))] =

sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),

(5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))

// 设置relationships边

scala> val relationships: RDD[Edge[String]] =

sc.parallelize(Array(Edge(3L, 7L, "collab"), Edge(5L, 3L, "advisor"),

Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))

// 定义默认的作者,以防与不存在的作者有relationship边

scala> val defaultUser = ("John Doe", "Missing")

// 构建图，构建好图我们就可以开始使用相关操作了

scala> val graph = Graph(users, relationships, defaultUser)

下面介绍部分操作：

* mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]

mapVertices的操作类似于顶点RDD中的mapValues操作，不同的是mapVertices是graph类的方法，且返回的也是graph类。

执行：

scala> val newgraph = graph.mapVertices((vid:VertexId, attr:(String, String)) => (attr.\_1 +" is a "+attr.\_2))

scala> newgraph.vertices.collect.foreach(println(\_))

将属性类型从（String，String）改成了String，属性值改成了“name is a pos”这种形式。由于返回的是graph类，所以想查看返回的结果时需要在后面加上vertices获取新图的顶点RDD进行打印查看。

返回结果：

(5,franklin is a prof)

(2,istoica is a prof)

(3,rxin is a student)

(7,jgonzal is a postdoc)

过程如下图所示：



图9-15 Graph的mapVertices操作

* mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]

mapEdges的操作类似于边RDD中的mapValues操作，不同的是mapEdges是graph类的方法，且返回的也是graph类。

执行：

scala> val newgraph = graph.mapEdges( e => ("relationship",e.attr))

scala> newgraph.edges.collect.foreach(println(\_))

将属性类型从String改成了（String，String），属性值改成了(”relationship”, attr)这种形式。由于返回的是graph类，所以想查看返回的结果时需要在后面加上edges获取新图的边RDD进行打印查看。

执行结果：

Edge(3,7,(relationship,collab))

Edge(5,3,(relationship,advisor))

Edge(2,5,(relationship,colleague))

Edge(5,7,(relationship,pi))

过程如下图所示：



图9-16 Graph的mapEdges操作

* mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]

mapTriplets遍历的是三元组EdgeTriplet，更改的是边的属性类型，与mapEdges相似，不同的是三元组可获取访问的变量更多，它可以直接访问源顶点及目标顶点的属性具体信息，而mapEdges无法访问。

执行：

scala> val newgraph = graph.mapTriplets( triplet => triplet.srcAttr.\_1+" to "+triplet.dstAttr.\_1+" is "+triplet.attr)

scala> newgraph.edges.collect.foreach(println(\_))

将原本边属性值的信息加上了源顶点和目标顶点的属性值组合成新的String类。

返回结果：

Edge(3,7,rxin to jgonzal is collab)

Edge(5,3,franklin to rxin is advisor)

Edge(2,5,istoica to franklin is colleague)

Edge(5,7,franklin to jgonzal is pi)

过程如下图所示：



图9-17 Graph的mapTriplets操作

* 1. 结构操作

下面介绍一些结构操作：

* reverse: Graph[VD, ED]

与边RDD的reverse操作作用一样，都是将目前边的方向进行调转，通过将源顶点与目标顶点对调来实现。

执行：

scala> val newgraph = graph.reverse

scala> newgraph.edges.collect.foreach(println(\_))

返回结果：

Edge(7,3,collab)

Edge(3,5,advisor)

Edge(5,2,colleague)

Edge(7,5,pi)

过程如下图所示：



图9-18 Graph的reverse操作

* subgraph(epred: EdgeTriplet[VD,ED] => Boolean,vpred: (VertexId, VD) => Boolean): Graph[VD, ED]

subgraph操作是构造子图，选出满足=>后面条件的顶点、边组成新图，其中筛选边是遍历三元组EdgeTriplet，而筛选顶点是遍历(VertexId，VD)的结构。

执行：

scala> val newgraph = graph.subgraph(triplet => triplet.attr!="colleague",(vid,attr) => attr.\_2!="prof")

scala> newgraph.vertices.collect.foreach(println(\_))

scala> newgraph.edges.collect.foreach(println(\_))

上述操作筛选出原来图中属性值中职位不为”prof”的顶点与属性值不为”colleague”的边（关系为”colleague”）。

返回结果：

顶点读取结果：

(3,(rxin,student))

(7,(jgonzal,postdoc))

边读取结果：

Edge(3,7,collab)

过程如下图所示：



图9-19 Graph的subgraph操作

* mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]

mask操作选取两个图共同的顶点和边组成新图，且属性值与原图相同。

首先我们需要构建一个graph1图：

scala> val users1: RDD[(VertexId, String)] =

sc.parallelize(Array((3L, "female"), (7L, "male"),

(5L, "female"), (1L, "male")))

scala> val relationships1: RDD[Edge[String]] =

sc.parallelize(Array(Edge(3L, 7L, "none"), Edge(5L, 3L, "none"),

Edge(1L, 5L, "none"), Edge(5L, 1L, "none")))

scala> val defaultUser1 = "male"

scala> val graph1 = Graph(users1, relationships1, defaultUser1)

接下来执行：

scala> val newgraph = graph.mask(graph1)

scala> newgraph.vertices.collect.foreach(println(\_))

scala> newgraph.edges.collect.foreach(println(\_))

可以看出，graph1中仅有id为1的顶点不在原图，而边Edge(1L, 5L, "none"), Edge(5L, 1L, "none")均不在原图中。

返回结果：

第一个结果：

(5,(franklin,prof))

(3,(rxin,student))

(7,(jgonzal,postdoc))

第二个结果：

Edge(3,7,collab)

Edge(5,3,advisor)

过程如下图所示：



图9-20 Graph的mask操作

* groupEdges(merge: (ED, ED) => ED): Graph[VD,ED]

groupEdges操作将同一个顶点对之间的多重边合并为一条边，具体的merge方法可以自己定义，描述两条边合并一条边的方法，在实际执行时类似于reduce操作，保证边属性类型不变即可。

创建一个拥有多重边的图graph1：

scala> val users1: RDD[(VertexId, String)] =

sc.parallelize(Array((3L, "female"), (7L, "male"),

(5L, "female"), (1L, "male")))

scala> val relationships1: RDD[Edge[Int]] =

sc.parallelize(Array(Edge(1L, 7L, 1), Edge(1L, 7L, 3), Edge(1L, 7L, 2),

Edge(7L, 3L, 2), Edge(7L, 3L, 7)))

scala> val defaultUser1 = "male"

scala> val graph1 = Graph(users1, relationships1, defaultUser1)

创建后，在进行groupEdges之前需用partitionBy将图进行分区，否则可能结果出错。

scala> val newgraph = graph1.partitionBy(PartitionStrategy.EdgePartition1D).groupEdges((attr1,attr2) => attr1+attr2)

scala> newgraph.vertices.collect.foreach(println(\_))

scala> newgraph.edges.collect.foreach(println(\_))

返回结果：

Edge(1,7,6)

Edge(7,3,9)

过程如下图所示：



图9-21 Graph的groupEdges操作

* 1. 连接操作
* joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD): Graph[VD, ED]

joinVertices与外部RDD集进行连接，不出现在另一图的顶点属性值保留原值，连接后的属性值更改可以通过map函数来实现，注意更改属性值需保证类型不变即可。

首先创建一个RDD集：

scala> val rdd: RDD[(VertexId, String)] =

sc.parallelize(Array((3L, "female"), (7L, "male"),

(5L, "female"), (1L, "male")))

执行：

scala> val newgraph = graph.joinVertices(rdd)((vid,attr,rddattr) => {

rddattr match{

case "female" => (attr.\_1,attr.\_2+" ♀ ")

case "male" => (attr.\_1,attr.\_2+" ♂ ")}})

scala> newgraph.vertices.collect.foreach(println(\_))

原图中id为2的顶点未出现在rdd顶点中，所以保留原来的属性值，而其他的顶点的属性值均由自定义的map函数进行相应地改变。

执行结果：

(5,(franklin,prof ♀ ))

(2,(istoica,prof))

(3,(rxin,student ♀ ))

(7,(jgonzal,postdoc ♂ ))

过程如下图所示：



图9-22 Graph的joinVertices操作

* outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])(map: (VertexId, VD, Option[U]) => VD2): Graph[VD2, ED]

outerJoinVertices操作与joinVertices操作类似，不同的是它可以更改属性类型，属性类型可以不再与原来保持一致，而且外部RDD集的属性变成可选项，对于未出现在另一RDD集中的顶点，可以自定义更改其属性值。

首先创建一个RDD集：

scala> val rdd: RDD[(VertexId, String)] =

sc.parallelize(Array((3L, "female"), (7L, "male"),

(5L, "female"), (1L, "male")))

执行：

scala> val newgraph = graph.outerJoinVertices(rdd)((vid,attr,rddattr) => {

rddattr match{

case Some(x) => {

x match{

case "female" => (attr.\_1,attr.\_2," ♀ ")

case "male" => (attr.\_1,attr.\_2," ♂ ")

}

}

case none => (attr.\_1,attr.\_2," none")}})

scala> newgraph.vertices.collect.foreach(println(\_))

将原来的(String,String)类型更改为(String,String,String)，且未在rdd中出现的id为2的顶点属性值最后一个默认为none。

返回结果：

(5,(franklin,prof, ♀ ))

(2,(istoica,prof, none))

(3,(rxin,student, ♀ ))

(7,(jgonzal,postdoc, ♂ ))

过程如下图所示：



图9-23 Graph的outerJoinVertices操作

* 1. 聚合操作（邻居之间）

图计算中比较关键的步骤是聚合来自邻居的消息，一些需要迭代计算的图算法，比如单源最短路径、PageRank等等图算法就需要反复地聚合来自邻居的消息。

1. 消息聚合

* aggregateMessages[Msg: ClassTag](

sendMsg: EdgeContext[VD, ED, Msg] => Unit,

mergeMsg: (Msg, Msg) => Msg,

tripletFields: TripletFields = TripletFields.All): VertexRDD[Msg]

aggregateMessages操作内部的三个参数含义分别是：

sendMsg是发送消息的函数，该函数可以自己定义，接受一个EdgeContext类型数据，会将EdgeContext转换成EdgeTriplet对象，里面可以获取到边属性、源顶点及目标顶点的属性和对应顶点id信息，除此之外还有发送函数sendToSrc和sendToDst，分别表示发送消息给源顶点以及目标顶点的操作。

mergeMsg是聚合消息的函数，该函数同样也是可以自己定义，定义两个消息聚合成一个消息的过程，其实这个操作就类似与reduce操作。

tripletFields是一个可选参数，它主要用来设置在EdgeContext类数据中可获得的范围，比如默认是TripletFields.All，意思就是可以访问EdgeContext类中的全部数据，若只允许访问源顶点的数据，那么可以将参数设置为TripletFields.Src。

信息聚合函数返回的是VertexRDD类型，其中的每个顶点属性值都是聚合后的属性值，特殊的是，未收到消息的顶点是不会出现在返回结果中的。

下面介绍一个小实例，计算收到消息的平均值：

首先创建一个顶点属性类型是double的图：

// 顶点

scala> val users1: RDD[(VertexId, Double)] =

sc.parallelize(Array((0L, 0.0), (1L, 1.0),(2L, 2.0)))

// 边

scala> val relationships1: RDD[Edge[Double]] =

sc.parallelize(Array(Edge(0L, 1L, 1), Edge(0L, 2L, 1),Edge(1L,0L,1),

Edge(1L,2L,1),Edge(2L, 0L, 1), Edge(2L, 1L, 1)))

scala> val defaultUser1 = 0.0

scala> val graph1 = Graph(users1, relationships1, defaultUser1)

创建完图后，可以查看顶点与边的信息：

scala> graph1.vertices.collect.foreach(println(\_))

scala> graph1.edges.collect.foreach(println(\_))

返回结果：

顶点：

(0,0.0)

(1,1.0)

(2,2.0)

边：

Edge(0,1,1)

Edge(0,2,1)

Edge(1,0,1)

Edge(1,2,1)

Edge(2,0,1)

Edge(2,1,1)

执行聚合操作：

scala> val olderFollowers = graph1.aggregateMessages[(Int, Double)](

triplet => {

if (triplet.srcAttr > triplet.dstAttr) {

triplet.sendToDst((1, triplet.srcAttr))}},

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

发送的消息的第一项累加的结果表示该顶点一共接受了多少个邻居的信息，第二项则是发送源顶点的属性值，需要注意的是只有当源顶点的属性值大于目标顶点属性值时才会有发送操作，累加结果则是发送给该顶点消息的所有源顶点的属性值之和。

只有Edge(1,0,1)、Edge(2,0,1)和Edge(2,1,1)三条边在传递消息，因为只有这三边满足源顶点的属性值大于目标顶点属性值这个条件，所以顶点0接收到的属性值之和为3.0，而顶点1接受到的属性值之和为2.0。

聚合操作后查看一下olderFollowers的信息：

scala> olderFollowers.collect.foreach(println(\_))

返回结果：

(0,(2,3.0))

(1,(1,2.0))

接下来求出每个顶点接受到的平均值：

scala> val average = olderFollowers.mapValues((id,attr) => attr match{

case (count,total) => total / count})

scala> average.collect.foreach(println(\_))

返回结果：

(0,1.5)

(1,2.0)

过程如下图所示：



图9-24 Graph的aggregateMessages操作

1. 计算顶点度数

各顶点度数情况可以通过graph.inDegrees,graph.outDegrees,graph.degrees获取，返回的均为VertexRDD类型，若要求最大/最小度数的顶点，则可以使用reduce操作：

首先定义一个执行函数，假设我们要求的是最大度数顶点：

scala> def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {

if (a.\_2 > b.\_2) a else b}

接着我们可以开始进行reduce操作：

scala> val maxInDegree = graph.inDegrees.reduce(max)

scala> val maxOutDegree = graph.outDegrees.reduce(max)

scala> val maxDegrees = graph.degrees.reduce(max)

分别求出最大入度、最大出度以及最大度数的顶点信息。

返回结果：

maxInDegree: (org.apache.spark.graphx.VertexId, Int) = (7,2)

maxOutDegree: (org.apache.spark.graphx.VertexId, Int) = (5,2)

maxDegrees: (org.apache.spark.graphx.VertexId, Int) = (5,3)

1. 另一种方法——直接收集邻居信息

graph类中提供了收集相邻顶点的id以及属性值信息，通过collectNeighborIds和collectNeighbors方法可以实现，但是这样的操作开销太大，每次进行一轮计算需要大量的通信，所以还是建议使用aggregateMessages进行聚合运算，因此不再详细赘述。

* 1. 缓存与取消缓存操作

在介绍RDD时讲到在默认情况RDD是不会保留在内存中，当进行迭代计算时，数据需要重复使用，这个时候将其进行缓存就变得很重要了，在GraphX中需要多次使用该图可以使用Graph.cache()将其缓存。

9.3 具体实例——Dijstra单源最短路径算法

前面已经介绍了很多基本的图操作，下面我们通过一个具体事例——实现Dijstra单源最短路径算法来更好地掌握它们。

9.3.1 Dijstra求单源最短路径算法介绍

Dijstra算法是求单源最短路径的经典算法，它每轮会找到一个当前拥有最短路径的顶点，并对该顶点的相邻顶点进行距离更新来求解源顶点到每个顶点的最短路径，算法的具体过程如下：

1. 初始化图，距离指的就是从源顶点出发到达的最短路径距离，所以源顶点的距离为0，其余顶点初始化为无穷大，所有顶点都标记为未访问状态
2. 开始迭代计算，每轮都选出当前拥有从源顶点出发的最短路径的目标顶点为当前顶点，对当前顶点的相邻顶点进行路径更新
3. 将当前顶点标记为已访问
4. 返回2)步继续迭代计算，若所有顶点均已访问则结束计算

9.3.2 Dijstra算法实例

下面我们来看一下具体实例，我们创建的图中的每个顶点表示一个用户，假设需要指定一个用户负责通知其他的用户，为了节省时间，需要求出这个源顶点到达每个顶点的最短路径，这就是单源最短路径问题。

1. 首先还是进入之前的graphx/src/main/scala中创建新的scala文件，命令如下：

$ cd /usr/local/spark/mycode/graphx/src/main/scala

$ vim Dijstra.scala

1. 按下i键开始编辑，代码如下：

import org.apache.log4j.{Level,Logger}

import org.apache.spark.\_

import org.apache.spark.graphx.\_

import org.apache.spark.rdd.RDD

import org.apache.spark.graphx.{Graph, VertexId}

object Dijstra {

def main(args: Array[String]) {

// 屏蔽日志

Logger.getLogger("org.apache.spark").setLevel(Level.WARN)

Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

// 设置运行环境

val conf = new SparkConf().setAppName("Dijstra").setMaster("local")

val sc = new SparkContext(conf)

val inf = Double.PositiveInfinity

// 初始化图，每个顶点表示用户，边表示用户各自的距离以及可达情况

val users: RDD[(VertexId,String)] =

sc.parallelize(Array((1L,"wyn"), (2L,"hh"),(3L,"zyy"),(4L,"pj")))

val distances: RDD[Edge[Double]] =

sc.parallelize(Array(Edge(1L,2L,2.0), Edge(1L,3L,0.5), Edge(2L,3L,2.0), Edge(2L,4L,1.0), Edge(3L,1L,1.0), Edge(4L,1L,2.0), Edge(4L,3L,0.5)))

val defaultUser = "none"

var graph = Graph(users, distances, defaultUser)

// 设置源顶点是id为2的点，即用户hh负责通知其他用户

val sourceId: VertexId = 2

// 初始化所有顶点，顶点属性值格式为(dist,0/1)，其中dist表示源顶点到各顶点的距离，初始值为无穷，表示与源顶点的距离为无穷，而源顶点本身则为0;第二项0/1，为0表示顶点还未访问，为1表示顶点已访问

var initialGraph = graph.mapVertices((vid,attr) =>

(if (vid == sourceId) 0.0 else inf, false))

println("原来图的顶点用户信息:")

println(initialGraph.vertices.collect.mkString("\n"))

println("原来图的边距离信息:")

println(initialGraph.edges.collect.mkString("\n"))

// 自定义一个求较小值的函数，用于选取当前拥有最短路径的顶点

def min(a: (VertexId,(Double,Boolean)), b: (VertexId,(Double,Boolean))): (VertexId,(Double,Boolean)) = {

if (a.\_2.\_1 < b.\_2.\_1) a else b

}

// 开始迭代计算，每轮遍历图中所有未被访问的顶点，将源顶点出发到达的最短路径的目的顶点作为当前顶点，开始对该顶点的相邻顶点进去路径更新

for (i <- 1L to initialGraph.vertices.count){

// 在这里将未访问过的顶点过滤出来从中选出拥有最短路径的顶点

val currentid = initialGraph.vertices.filter{ case (vid,attr) => !attr.\_2}.reduce(min).\_1

val newdist = initialGraph.aggregateMessages[Double](

// sendMsg: 向当前用户的相邻用户发送消息，内容为到相邻边的距离与最短路径值之和

triplet => {

if (triplet.srcId == currentid)

triplet.sendToDst(triplet.srcAttr.\_1 + triplet.attr)

},

// mergeMsg: 选择较小的值为源用户到该用户的最短路径值，实际上就是最短路径的更新

(a,b) => math.min(a,b))

// 一轮消息发送完后需要执行outerJoinVertices对原来的顶点进行更新，只会对上一轮接受到消息的顶点进行更新，并且将当前顶点设置成已访问状态

initialGraph = initialGraph.outerJoinVertices(newdist)((vid, attr, newdist) =>

(math.min(attr.\_1, newdist.getOrElse(inf)), attr.\_2 || vid == currentid))

}

// 最终显示出(id,(name,dist))的格式信息

val results = graph.vertices.join(initialGraph.vertices).map{

case (vid, (username, attr)) => (vid, (username, attr.\_1))

}

// 打印最终信息

println("迭代后的顶点信息:")

println(results.collect.mkString("\n"))

}

}

1. 返回graphx目录用sbt编译打包

如果没有创建simple.sbt的话需要先创建：（之前已经创建过可以跳过这一步）

$ vim simple.sbt

输入以下信息：

name := "Simple Project"

version := "1.0"

scalaVersion := "2.11.12"

libraryDependencies += "org.apache.spark" %% "spark-core" % "2.4.0"

libraryDependencies += "org.apache.spark" %% "spark-graphx" %"2.4.0"

保存即可，然后sbt编译打包

$ cd /usr/local/spark/mycode/graphx

$ /usr/local/sbt/sbt package

显示

[success] Total time: 6 s, completed 2020-8-20 11:16:21

说明成功

1. 使用spark-submit执行程序，命令如下：

$ /usr/local/spark/bin/spark-submit \

> --class "Dijstra" \

> /usr/local/spark/mycode/graphx/target/scala-2.11/simple-project\_2.11-1.0.jar

执行后会有很多输出信息，直接看最终结果：

原来图的顶点用户信息:

(4,(Infinity,false))

(1,(Infinity,false))

(3,(Infinity,false))

(2,(0.0,false))

原来图的边距离信息:

Edge(1,2,2.0)

Edge(1,3,0.5)

Edge(2,3,2.0)

Edge(2,4,1.0)

Edge(3,1,1.0)

Edge(4,1,2.0)

Edge(4,3,0.5)

迭代后的顶点信息:

(4,(pj,1.0))

(1,(wyn,2.5))

(3,(zyy,1.5))

(2,(hh,0.0))

过程如下：



图9-25 Dijstra过程

Dijstra实例运行成功。

9.4 图算法——PageRank算法实例

9.4.1 PageRank原理介绍

PageRank是一个函数，它为网络中每个网页赋一个权值。通过该权值来判断该网页的重要性。该权值分配的方法并不是固定的，对PageRank算法的一些简单变形都会改变网页的相对PageRank值（或PR值）。PageRank作为谷歌的网页链接排名算法，其基本公式如下。



也就是说，对于任意一个网页链接，其PR值为链入该链接的源链接的PR值对该链接的贡献和。其中，*N*表示该网络中所有网页的数量，*Ni*为第*i*个源链接的链出度，*PRi*表示第*i*个源链接的PR值。

网络链接之间的关系可以用一个连通图来表示。图9-26就是4个网页（A、B、C、D）互相链入链出组成的连通图，从图中可以看出，网页A中包含指向网页B、C和D的外链，网页B和D是网页A的源链接。



图9-26 一个反映网页之间链接关系的连通图

9.4.2 PageRank实例

下面我们来看一下PageRank实例：

1. 首先还是进入之前的graphx/src/main/scala中创建新的scala文件，命令如下：

$ cd /usr/local/spark/mycode/graphx/src/main/scala

$ vim PageRank.scala

1. 按下i键开始编辑，代码如下：

import org.apache.log4j.{Level,Logger}

import org.apache.spark.\_

import org.apache.spark.graphx.GraphLoader

object PageRank {

def main(args: Array[String]) {

// 屏蔽日志

Logger.getLogger("org.apache.spark").setLevel(Level.WARN)

Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

// 设置运行环境

val conf = new SparkConf().setAppName("PageRank").setMaster("local")

val sc = new SparkContext(conf)

// 读取followers.txt

val graph = GraphLoader.edgeListFile(sc, "file:///usr/local/spark/data/graphx/followers.txt")

// 运行PageRank，参数指定前后两次迭代的结果差值应小于0.0001，达到最终收敛的效果时才停止计算，然后使用vertices返回顶点RDD

val ranks = graph.pageRank(0.0001).vertices

// 读取用户信息，得到(顶点id, 对应用户名)类型的RDD数据集

val users = sc.textFile("file:///usr/local/spark/data/graphx/users.txt").map { line =>

val fields = line.split(",")

(fields(0).toLong, fields(1))

}

// 根据顶点id进行连接操作后，只保留键值，也就是名字与排名

val ranksByUsername = users.join(ranks).map {

case (id, (username, rank)) => (username, rank)

}

// 打印结果

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

}

}

1. 返回graphx目录用sbt编译打包

如果没有创建simple.sbt的话需要先创建：（之前已经创建过可以跳过这一步）

$ vim simple.sbt

输入以下信息：

name := "Simple Project"

version := "1.0"

scalaVersion := "2.11.12"

libraryDependencies += "org.apache.spark" %% "spark-core" % "2.4.0"

libraryDependencies += "org.apache.spark" %% "spark-graphx" %"2.4.0"

保存即可，然后sbt编译打包

$ cd /usr/local/spark/mycode/graphx

$ /usr/local/sbt/sbt package

显示

[success] Total time: 4 s, completed 2020-8-6 15:11:00

说明成功

1. 使用spark-submit执行程序，命令如下：

$ /usr/local/spark/bin/spark-submit \

> --class "PageRank" \

> /usr/local/spark/mycode/graphx/target/scala-2.11/simple-project\_2.11-1.0.jar

执行后会有很多输出信息，直接看最终结果：

(justinbieber,0.15007622780470478)

(BarackObama,1.4596227918476916)

(matei\_zaharia,0.7017164142469724)

(jeresig,0.9998520559494657)

(odersky,1.2979769092759237)

(ladygaga,1.3907556008752426)

PageRank实例运行成功。

9.5 Pregel API

前面提到了RDD默认是不会保留在内存中的，当使用迭代计算时，也就是需要重复用到数据时，为了计算更快，我们必须手动进行缓存操作，也就是执行Graph.cache()，而且过后还需要清理缓存（将无用的删除），那么这个时候Pregel API会更加适用，它让我们可以用Pregel的计算框架来处理图数据，是处理迭代计算的有利工具，同时它也是一种优化后的API，与原来的标准Pregel模型存在一些区别。

在进行介绍前，我们需要先了解Pregel是什么，Pregel是一种基于BSP模型实现的并行图处理系统。对于不同顶点之间的信息交换，Pregel采用的是纯消息传递模型。GraphX中的Pregel API便是批量同步并行消息传递，在上一个超步每个顶点将收到来自其他顶点的消息并计算出自己新的属性值，在下一个超步将对其他顶点发送新的消息，下面我们会详细介绍Pregel的计算过程。

9.5.1 Pregel的计算过程

Pregel的计算过程是由一系列被称为“超步”的迭代组成的（见图9-27）。在每个超步中，每个顶点上面都会并行执行用户自定义的函数，该函数描述了一个顶点V在一个超步S中需要执行的操作。该函数可以读取前一个超步（S−1）中其他顶点发送给顶点V的消息，执行相应计算后，修改顶点V及其出射边的状态，然后沿着顶点V的出射边发送消息给其他顶点，而且一个消息可能经过多条边的传递后被发送到任意已知ID的目标顶点上去。这些消息将会在下一个超步（S+1）中被目标顶点接收，然后像上述过程一样开始下一个超步（S+1）的迭代过程。

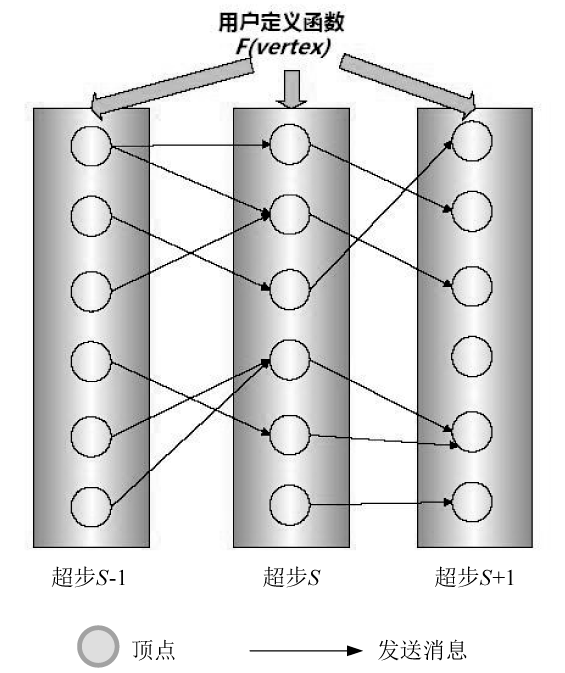


图9-27 Pregel计算模型中的超步

在Pregel计算过程中，一个算法什么时候可以结束，是由所有顶点的状态决定的，当图中所有的顶点都已经标识其自身达到“非活跃（Inactive）”状态时，算法就可以停止运行。在第0个超步，图中所有顶点都处于“活跃（Active）”状态，这些活跃顶点都会参与对应超步的计算过程。当一个顶点不需要继续执行进一步的计算时，就会调用VoteToHalt()把自己的状态设置成“停机”，从而表示自己不再是活跃顶点。一旦一个顶点进入“非活跃”状态，Pregel计算过程在后续的超步中就不会再在该顶点上执行计算，除非其他顶点给该顶点发送消息再次把它激活。当一个已经处于“非活跃”状态的顶点再次接收到来自其他顶点的消息时，Pregel计算框架必须根据条件判断来决定是否将其显式“唤醒”进入“活跃”状态。当所有顶点都达到“非活跃”状态，并且没有消息在传送的时候，整个计算过程就宣告结束。这种计算过程可以用图9-28所示的简单状态机来描述。

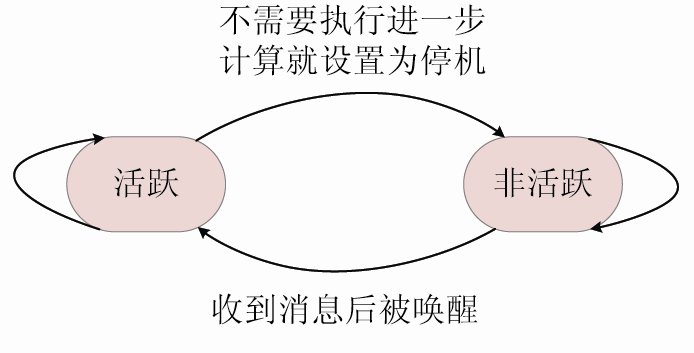


图9-28 一个简单状态机

9.5.2 Spark GraphX的Pregel操作与标准Pregel的不同

Spark GraphX的Pregel操作与标准Pregel模型是有所不同的，具体如下：

* GraphX中Pregel API的消息计算是基于三元组的，也就是说在进行消息计算时可以同时访问源顶点与目标顶点的属性，而且在超步执行中将直接跳过未接受到消息的顶点，这与标准Pregel有所不同。
* GraphX的顶点只能将消息发送给相邻顶点，且消息的组成可以用户自己定义方法（mergeMsg），这样有利于扩展到很多不同情况。

9.5.3 Pregel API的实现草图

以下是Pregel接口的实现草图：

class GraphOps[VD, ED] {

def pregel[A]

// 第一个参数列表，主要包含配置参数

(initialMsg: A,

maxIter: Int = Int.MaxValue,

activeDir: EdgeDirection = EdgeDirection.Out)

// 第二个参数列表，主要包含消息的接受、计算以及组成函数

(vprog: (VertexId, VD, A) => VD,

sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],

mergeMsg: (A, A) => A)

: Graph[VD, ED] = {

// 从每个定点接受初始化消息

var g = mapVertices( (vid, vdata) => vprog(vid, vdata, initialMsg) ).cache()

// 计算消息

var messages = g.mapReduceTriplets(sendMsg, mergeMsg)

var activeMessages = messages.count()

// 当没有消息传递或者达到最大迭代次数时退出循环，不再计算

var i = 0

while (activeMessages > 0 && i < maxIterations) {

// 接受消息更新顶点

g = g.joinVertices(messages)(vprog).cache()

val oldMessages = messages

// 发送新消息，跳过不接受消息的顶点，对顶点进行缓存为了下一次的使用，如果缓存满了可以将之前迭代轮次的取消缓存

messages = GraphXUtils.mapReduceTriplets(

g, sendMsg, mergeMsg, Some((oldMessages, activeDirection))).cache()

activeMessages = messages.count()

i += 1

}

}

}

下面通过一个单源最短路径的实例来体会Pregel API的使用。

9.5.4 Pregel API实例——单源最短路径

单源最短路径实例中，每个顶点关联的对象属性值表示的是距离源顶点的距离，一开始我们可以随便选一个顶点作为源顶点，源顶点的属性值是0（到自身的距离为0），将所有其他顶点都初始化无穷（一个很大的数），然后进行迭代计算最终生成的图中每个顶点的属性值就代表源顶点到该顶点的最短距离。

1. 首先进入之前的graphx/src/main/scala目录中建立一个新的scala文件，命令如下：

$ cd /usr/local/spark/mycode/graphx/src/main/scala

$ vim SSSPExample.scala

1. 进入后按i键进行编辑，输入以下代码：

import org.apache.log4j.{Level,Logger}

import org.apache.spark.\_

import org.apache.spark.graphx.\_

import org.apache.spark.rdd.RDD

import org.apache.spark.graphx.util.GraphGenerators

import org.apache.spark.graphx.{Graph, VertexId}

object SSSPExample {

def main(args: Array[String]) {

//屏蔽日志

Logger.getLogger("org.apache.spark").setLevel(Level.WARN)

Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

//设置运行环境

val conf = new SparkConf().setAppName("SSSPExample").setMaster("local")

val sc = new SparkContext(conf)

val inf = Double.PositiveInfinity

// 初始化图，每个顶点表示用户，边表示用户各自的距离以及可达情况

val users: RDD[(VertexId,String)] =

sc.parallelize(Array((1L,"wyn"), (2L,"hh"),(3L,"zyy"),(4L,"pj")))

val distances: RDD[Edge[Double]] =

sc.parallelize(Array(Edge(1L,2L,2.0), Edge(1L,3L,0.5), Edge(2L,3L,2.0), Edge(2L,4L,1.0), Edge(3L,1L,1.0), Edge(4L,1L,2.0), Edge(4L,3L,0.5)))

val defaultUser = "none"

var graph = Graph(users, distances, defaultUser)

// 设置源顶点是id为2的点

val sourceId: VertexId = 2

// 初始化所有顶点的属性值为无穷，表示与源顶点的距离为无穷，而源顶点则为0

var initialGraph = graph.mapVertices((vid,attr) =>

if (vid == sourceId) 0.0 else inf)

// 打印出原来图的信息

println("原来图的顶点用户信息:")

println(initialGraph.vertices.collect.mkString("\n"))

println("原来图的边距离信息:")

println(initialGraph.edges.collect.mkString("\n"))

// Pregel API

val sssp = initialGraph.pregel(inf)(

// 顶点程序

(id, dist, newDist) => math.min(dist, newDist),

// 发送消息

triplet => {

if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {

Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))

}

else {

Iterator.empty

}

},

// 组成消息

(a, b) => math.min(a, b)

)

// 最终显示出(id,(name,dist))的格式信息

val results = graph.vertices.join(sssp.vertices)

println("迭代后的顶点信息:")

println(results.collect.mkString("\n"))

}

}

完成后按Esc后输入:wq退出

1. 返回graphx目录下用sbt编译打包

如果没有创建simple.sbt的话需要先创建：（之前已经创建过可以跳过这一步）

$ vim simple.sbt

输入以下信息：

name := "Simple Project"

version := "1.0"

scalaVersion := "2.11.12"

libraryDependencies += "org.apache.spark" %% "spark-core" % "2.4.0"

libraryDependencies += "org.apache.spark" %% "spark-graphx" %"2.4.0"

保存即可，然后sbt编译打包

$ cd /usr/local/spark/mycode/graphx

$ /usr/local/sbt/sbt package

显示

[success] Total time: 4 s, completed 2020-8-6 13:38:11

说明成功

1. 使用spark-submit执行程序，命令如下：

$ /usr/local/spark/bin/spark-submit \

> --class "SSSPExample" \

> /usr/local/spark/mycode/graphx/target/scala-2.11/simple-project\_2.11-1.0.jar

执行后会有很多输出信息，直接看最终结果：

原来图的顶点用户信息:

(4,Infinity)

(1,Infinity)

(3,Infinity)

(2,0.0)

原来图的边距离信息:

Edge(1,2,2.0)

Edge(1,3,0.5)

Edge(2,3,2.0)

Edge(2,4,1.0)

Edge(3,1,1.0)

Edge(4,1,2.0)

Edge(4,3,0.5)

迭代后的顶点信息:

(4,(pj,1.0))

(1,(wyn,2.5))

(3,(zyy,1.5))

(2,(hh,0.0))

顶点的属性值代表源顶点到该顶点的最短路径，可以看出显示出了正确结果。

过程如下：



图9-29 Pregel API实现单源最短路径

与之前的Dijstra单源最短路径实例相比，使用Pregel API的接口不需要手动执行outerJoinVertices操作，对于迭代计算来说使用Pregel API会更加方便。