**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

9.1.1 图的简单介绍

9.1.2 传统图计算解决方案的不足之处

9.1.3 图计算通用软件

9.2 Spark GraphX

9.2.1 属性图介绍

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，其中顶点和边是由特定的RDD类型扩展得到的，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表示的就是顶点的对象类型，ED则是边的对象类型，属性图中这些对象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扩展上去即可，例如需要用户和产品两种不同属性的顶点，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。

9.2.2 顶点RDD、边RDD以及三元组RDD

* 1. 顶点RDD todo

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，其实基础就是二元组(标识符，对象类型)的顶点RDD集合，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，可以看成对象类型为VD的顶点RDD集合以及作用在上面的相关操作的集成，下面我们简单介绍部分操作：

* filter(pred: Tuple2[VertexId, VD] => Boolean): VertexRDD[VD]
* mapValues[VD2](map: (VertexId, VD) => VD2): VertexRDD[VD2]
* minus(other: RDD[(VertexId, VD)])
* diff(other: VertexRDD[VD]): VertexRDD[VD]
* innerJoin[U, VD2](other: RDD[(VertexId, U)])(f: (VertexId, VD, U) => VD2): VertexRDD[VD2]
* aggregateUsingIndex[VD2](other: RDD[(VertexId, VD2)], reduceFunc: (VD2, VD2) => VD2): VertexRDD[VD2]
  1. 边RDD todo

类中的EdgeRDD[ED]继承了RDD[Edge[ED]]，ED是与边关联的对象类型，基础就是类型是Edge[ED]的边RDD集合，其中Edge[ED]指的是关联对象类型为ED的边类，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指边所关联的对象值，对象类型是ED，Edge[ED]其实就是表示attr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]，可以看成对象类型为ED的边RDD集合以及作用在上面的相关操作的集成，下面我们简单介绍部分操作：

* mapValues[ED2](f: Edge[ED] => ED2): EdgeRDD[ED2]
* reverse: EdgeRDD[ED]
* innerJoin[ED2, ED3](other: EdgeRDD[ED2])(f: (VertexId, VertexId, ED, ED2) => ED3): EdgeRDD[ED3]
  1. 三元组RDD

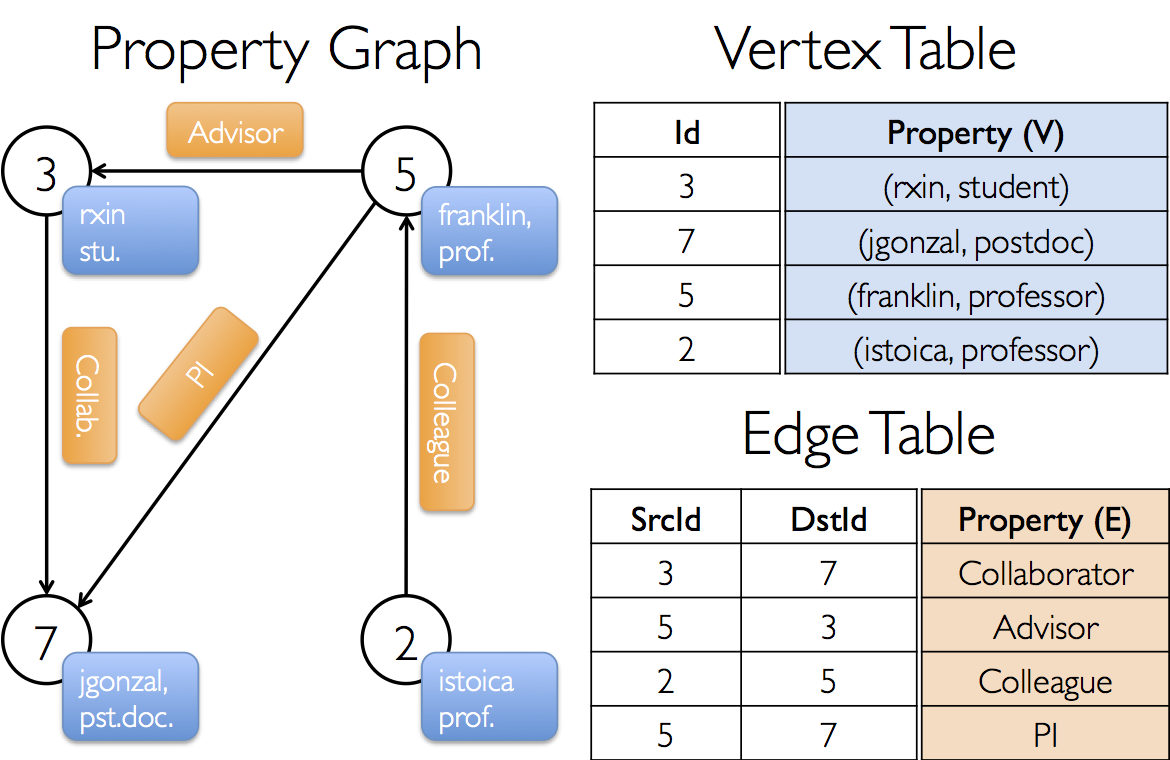
三元组RDD类型是RDD[EdgeTriplet[VD, ED]]，也就是类型为EdgeTriplet[VD, ED]的RDD集合，其中EdgeTriplet类型是从边类Edge继承得来的，三元组的添加的具体信息可以参考下图：



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。

* 1. 实例

下面是一个属性图的小实例，主要图结构如下图所示：



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

表格todo

* 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：

// 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：todo

* mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
* mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
* mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
  1. 结构操作

下面介绍一些结构操作：todo

* reverse: Graph[VD, ED]
* subgraph(epred: EdgeTriplet[VD,ED] => Boolean,vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
* mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
* groupEdges(merge: (ED, ED) => ED): Graph[VD,ED]
  1. 连接操作
* joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD): Graph[VD, ED]
* outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])(map: (VertexId, VD, Option[U]) => VD2): Graph[VD2, ED]
  1. 聚合操作（邻居之间）todo

1. 信息聚合
2. 计算顶点度数
3. 另一种方法——直接收集邻居信息
   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。当然，如果是针对迭代计算，下面要介绍的Pregel API会更加适用。

9.3 Pregel API

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。

9.3.1 Pregel的计算过程

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

* 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。

9.3.3 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

}

g

}

}

下面通过一个单源最短路径的实例来体会Pregel API的使用。

9.3.4 Pregel API应用实例——单源最短路径

9.4 构建图

9.5 图算法——PageRank算法实例

9.5.1 PageRank原理介绍

9.5.2 PageRank实例