# Spark入门实战系列--9.Spark图计算GraphX介绍及实例

【注】该系列文章以及使用到安装包/测试数据 可以在《倾情大奉送--Spark入门实战系列》获取

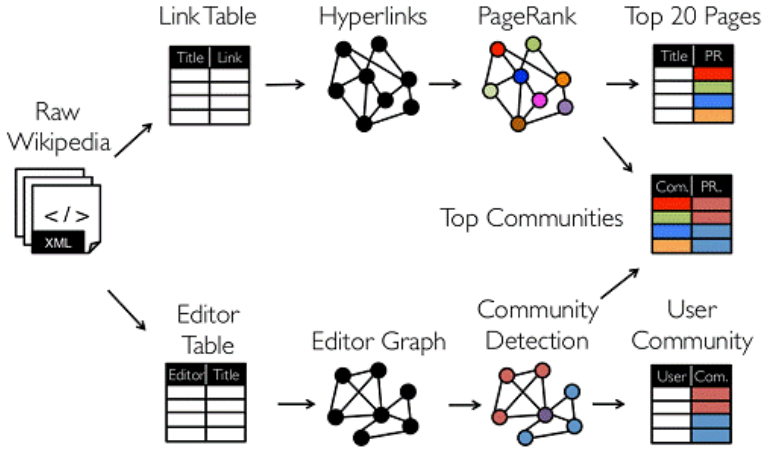
## 1、GraphX介绍

### 1.1 GraphX应用背景

Spark GraphX是一个分布式图处理框架，它是基于Spark平台提供对图计算和图挖掘简洁易用的而丰富的接口，极大的方便了对分布式图处理的需求。

众所周知，社交网络中人与人之间有很多关系链，例如Twitter、Facebook、微博和微信等，这些都是大数据产生的地方都需要图计算，现在的图处理基本都是分布式的图处理，而并非单机处理。Spark GraphX由于底层是基于Spark来处理的，所以天然就是一个分布式的图处理系统。

图的分布式或者并行处理其实是把图拆分成很多的子图，然后分别对这些子图进行计算，计算的时候可以分别迭代进行分阶段的计算，即对图进行并行计算。下面我们看一下图计算的简单示例：




从图中我们可以看出：拿到Wikipedia的文档以后，可以变成Link Table形式的视图，然后基于Link Table形式的视图可以分析成Hyperlinks超链接，最后我们可以使用PageRank去分析得出Top Communities。在下面路径中的Editor Graph到Community，这个过程可以称之为Triangle Computation，这是计算三角形的一个算法，基于此会发现一个社区。从上面的分析中我们可以发现图计算有很多的做法和算法，同时也发现图和表格可以做互相的转换。

### 1.2 GraphX的框架

设计GraphX时，点分割和GAS都已成熟，在设计和编码中针对它们进行了优化，并在功能和性能之间寻找最佳的平衡点。如同Spark本身，每个子模块都有一个核心抽象。GraphX的核心抽象是Resilient Distributed Property Graph，一种点和边都带属性的有向多重图。它扩展了Spark RDD的抽象，有Table和Graph两种视图，而只需要一份物理存储。两种视图都有自己独有的操作符，从而获得了灵活操作和执行效率。

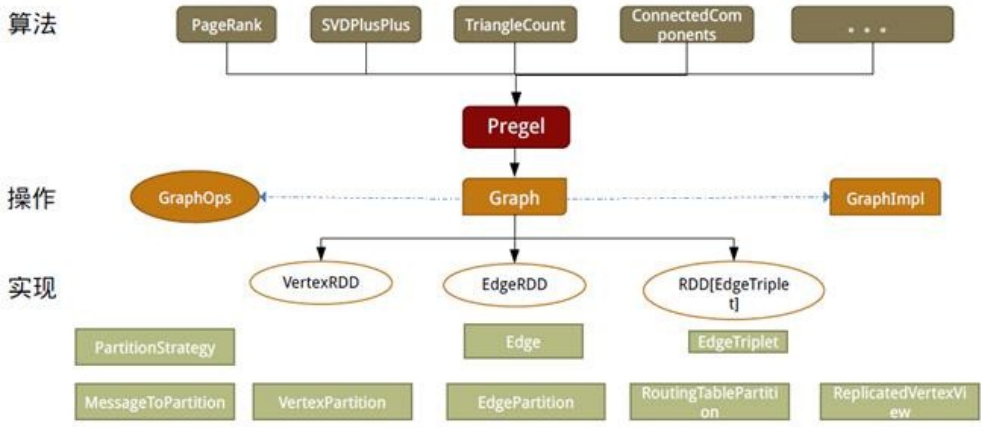
**个人简介**

郭景瞻，博客以家乡石山园为名，大数据图书作者，著《图解Spark：核心技术与案例实战》。

**Spark核心技术与案例[《图解Spark：核心技术与案例实战》]**




昵称：shishanyuan
园龄：7年11个月
荣誉：推荐博客
粉丝：617
关注：16
+加关注



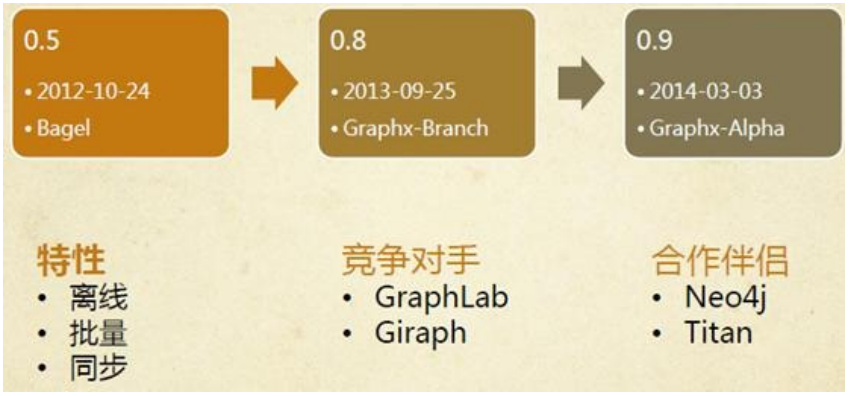

如同Spark，GraphX的代码非常简洁。GraphX的核心代码只有3千多行，而在此之上实现的Pregel模式，只要短短的20多行。GraphX的代码结构整体下图所示，其中大部分的实现，都是围绕Partition的优化进行的。这在某种程度上说明了点分割的存储和相应的计算优化，的确是图计算框架的重点和难点。

### 1.3 发展历程

●早在0.5版本，Spark就带了一个小型的Bagel模块，提供了类似Pregel的功能。当然，这个版本还非常原始，性能和功能都比较弱，属于实验型产品。

●到0.8版本时，鉴于业界对分布式图计算的需求日益见涨，Spark开始独立一个分支Graphx Branch，作为独立的图计算模块，借鉴GraphLab，开始设计开发GraphX。

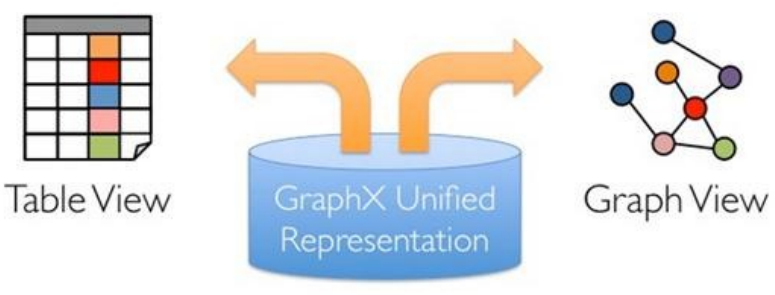
●在0.9版本中，这个模块被正式集成到主干，虽然是Alpha版本，但已可以试用，小面包圈Bagel告别舞台。1.0版本，GraphX正式投入生产使用。




值得注意的是，GraphX目前依然处于快速发展中，从0.8的分支到0.9和1.0，每个版本代码都有不少的改进和重构。根据观察，在没有改任何代码逻辑和运行环境，只是升级版本、切换接口和重新编译的情况下，每个版本有10%~20%的性能提升。虽然和GraphLab的性能还有一定差距，但凭借Spark整体上的一体化流水线处理，社区热烈的活跃度及快速改进速度，GraphX具有强大的竞争力。

## 2、GraphX实现分析

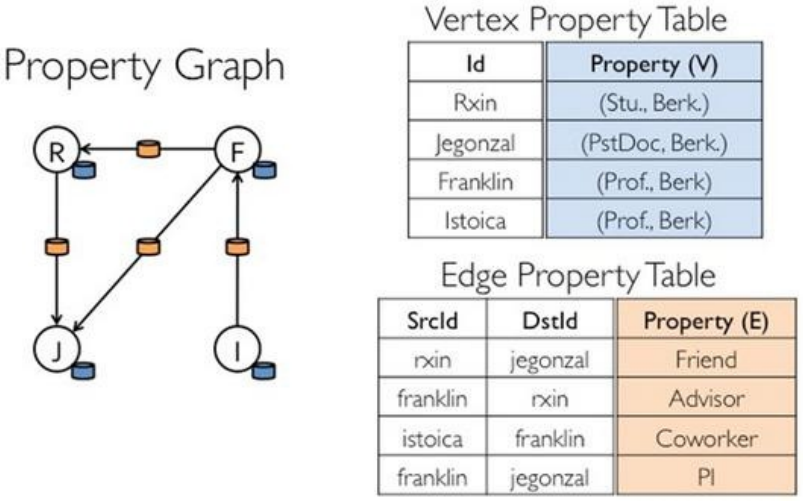
如同Spark本身，每个子模块都有一个核心抽象。GraphX的核心抽象是Resilient Distributed Property Graph，一种点和边都带属性的有向多重图。它扩展了Spark RDD的抽象，有Table和Graph两种视图，而只需要一份物理存储。两种视图都有自己独有的操作符，从而获得了灵活操作和执行效率。




GraphX的底层设计有以下几个关键点。

对Graph视图的所有操作，最终都会转换成其关联的Table视图的RDD操作来完成。这样对一个图的计算，最终在逻辑上，等价于一系列RDD的转换过程。因此，Graph最终具备了RDD的3个关键特性：Immutable、Distributed和Fault-Tolerant，其中最关键的是Immutable（不变性）。逻辑上，所有图的转换和操作都产生了一个新图；物理上，GraphX会有一定程度的不变顶点和边的复用优化，对用户透明。

两种视图底层共用的物理数据，由RDD[Vertex-Partition]和RDD[EdgePartition]这两个RDD组成。点和边实际都不是以表Collection[tuple]的形式存储的，而是由VertexPartition/EdgePartition在内部存储一个带索引结构的分片数据块，以加速不同视图下的遍历速度。不变的索引结构在RDD转换过程中是共用的，降低了计算和存储开销。



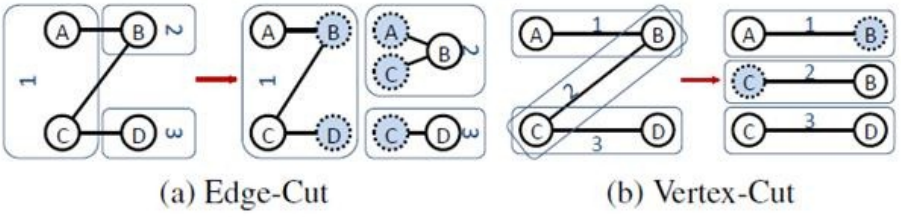

图的分布式存储采用点分割模式，而且使用partitionBy方法，由用户指定不同的划分策略（PartitionStrategy）。划分策略会将边分配到各个EdgePartition，顶点Master分配到各个VertexPartition，EdgePartition也会缓存本地边关联点的Ghost副本。划分策略的不同会影响到所需要缓存的Ghost副本数量，以及每个EdgePartition分配的边的均衡程度，需要根据图的结构特征选取最佳策略。目前有EdgePartition2d、EdgePartition1d、RandomVertexCut和CanonicalRandomVertexCut这四种策略。

### 2.1 存储模式

#### 2.1.1 图存储模式

巨型图的存储总体上有边分割和点分割两种存储方式。2013年，GraphLab2.0将其存储方式由边分割变为点分割，在性能上取得重大提升，目前基本上被业界广泛接受并使用。

- **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。




虽然两种方法互有利弊，但现在是点分割占上风，各种分布式图计算框架都将自己底层的存储形式变成了点分割。主要原因有以下两个。

1.磁盘价格下降，存储空间不再是问题，而内网的通信资源没有突破性进展，集群计算时内网带宽是宝贵的，时间比磁盘更珍贵。这点就类似于常见的空间换时间的策略。

2.在当前的应用场景中，绝大多数网络都是“无尺度网络”，遵循幂律分布，不同点的邻居数量相差非常悬殊。而边分割会使那些多邻居的点所相连的边大多数被分到不同的机器上，这样的数据分布会使得内网带宽更加捉襟见肘，于是边分割存储方式被渐渐抛弃了。

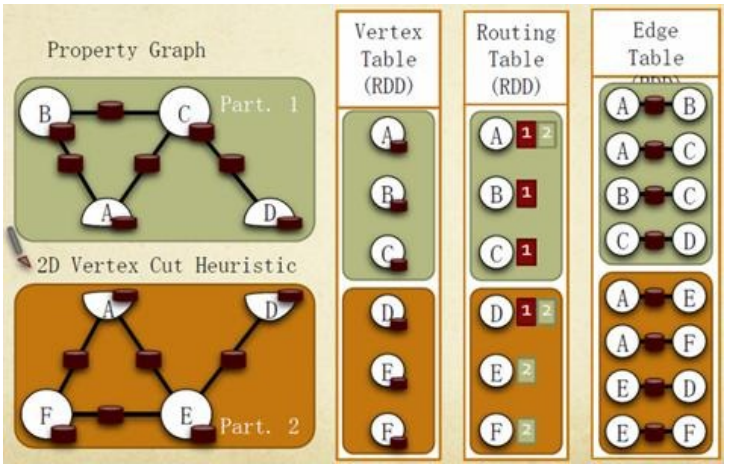
#### 2.1.2 GraphX存储模式

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Partion id

点分割存储实现如下图所示：

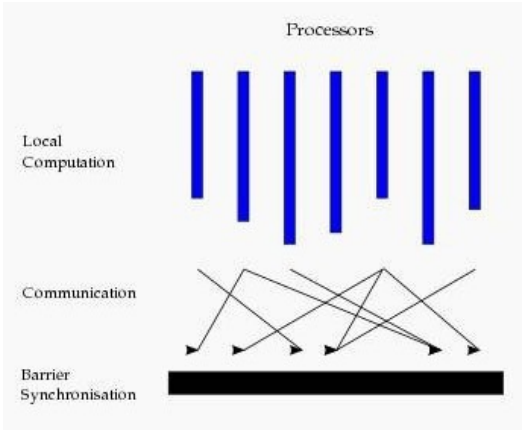



### 2.2 计算模式

#### 2.2.1 图计算模式

目前基于图的并行计算框架已经有很多，比如来自Google的Pregel、来自Apache开源的图计算框架Giraph/HAMA以及最为著名的GraphLab，其中Pregel、HAMA和Giraph都是非常类似的，都是基于BSP（Bulk Synchronous Parallell）模式。

Bulk Synchronous Parallell，即整体同步并行，它将计算分成一系列的超步（superstep）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superstep之间设置一个栅栏（barrier），即整体同步点，确定所有并行的计算都完成后再启动下一轮superstep。




每一个超步（superstep）包含三部分内容：

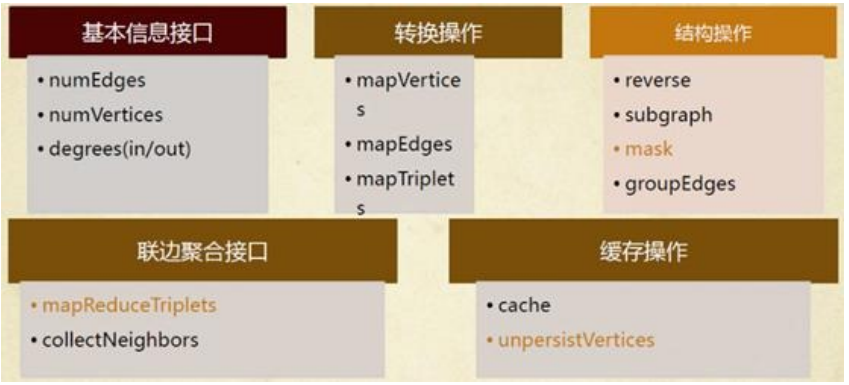
1.**计算compute**：每一个processor利用上一个superstep传过来的消息和本地的数据进行本地计算；

2.**消息传递**：每一个processor计算完毕后，将消息传递个与之关联的其它processors

3.**整体同步点**：用于整体同步，确定所有的计算和消息传递都进行完毕后，进入下一个superstep。

#### 2.2.2GraphX计算模式

如同Spark一样，GraphX的Graph类提供了丰富的图运算符，大致结构如下图所示。可以在官方GraphX Programming Guide中找到每个函数的详细说明，本文仅讲述几个需要注意的方法。




#### 2.2.2.1 图的缓存

每个图是由3个RDD组成，所以会占用更多的内存。相应图的cache、unpersist和checkpoint，更需要注意使用技巧。出于最大限度复用边的理念，GraphX的默认接口只提供了unpersistVertices方法。如果要释放边，调用g.edges.unpersist()方法才行，这给用户带来了一定的不便，但为GraphX的优化提供了便利和空间。参考GraphX的Pregel代码，对一个大图，目前最佳的实践是：

```
var g=...
var prevG: Graph[VD, ED] = null
while(...){
  prevG = g
  g = doSomething(g)
  g.cache()
  prevG.unpersistVertices(blocking=false)
  prevG.edges.unpersist(blocking=false)
}
```

大体之意是根据GraphX中Graph的不变性，对g做操作并赋回给g之后，g已不是原来的g了，而且会在下一轮迭代使用，所以必须cache。另外，必须先用prevG保留住对原来图的引用，并在新图产生后，快速将旧图彻底释放掉。否则，十几轮迭代后，会有内存泄漏问题，很快耗光作业缓存空间。

#### 2.2.2.2 邻边聚合

mrTriplets（mapReduceTriplets）是GraphX中最核心的一个接口。Pregel也基于它而来，所以对它的优化能很大程度上影响整个GraphX的性能。mrTriplets运算符的简化定义是：

```
def mapReduceTriplets[A](
  map: EdgeTriplet[VD, ED] =>
Iterator[(VertexID, A)],
  reduce: (A, A) => A)
  : VertexRDD[A]
```

它的计算过程为：map，应用于每一个Triplet上，生成一个或者多个消息，消息以Triplet关联的两个顶点中的任意一个或两个为目标顶点；reduce，应用于每一个Vertex上，将发送给每一个顶点的消息合并起来。

mrTriplets最后返回的是一个VertexRDD[A]，包含每一个顶点聚合之后的消息（类型为A），没有接收到消息的顶点不会包含在返回的VertexRDD中。

在最近的版本中，GraphX针对它进行了一些优化，对于Pregel以及所有上层算法工具包的性能都有重大影响。主要包括以下几点。

1. **Caching for Iterative mrTriplets & Incremental Updates for Iterative mrTriplets**：在很多图分析算法中，不同点的收敛速度变化很大。在迭代后期，只有很少的点会有更新。因此，对于没有更新的点，下一次mrTriplets计算时EdgeRDD无需更新相应点值的本地缓存，大幅降低了通信开销。

2. **Indexing Active Edges**：没有更新的顶点在下一轮迭代时不需要向邻居重新发送消息。因此，mrTriplets遍历边时，如果一条边的邻居点值在上一轮迭代时没有更新，则直接跳过，避免了大量无用的计算和通信。

3. **Join Elimination**：Triplet是由一条边和其两个邻居点组成的三元组，操作Triplet的map函数常常只需访问其两个邻居点值中的一个。例如，在PageRank计算中，一个点值的更新只与其源顶点的值有关，而与其所指向的目的顶点的值无关。那么在mrTriplets计算中，就不需要VertexRDD和EdgeRDD的3-way join，而只需要2-way join。

所有这些优化使GraphX的性能逐渐逼近GraphLab。虽然还有一定差距，但一体化的流水线服务和丰富的编程接口，可以弥补性能的微小差距。

#### 2.2.2.3 进化的Pregel模式

GraphX中的Pregel接口，并不严格遵循Pregel模式，它是一个参考GAS改进的Pregel模式。定义如下：

```
def pregel[A](initialMsg: A, maxIterations:
  Int, activeDirection: EdgeDirection)(
  vprog: (VertexID, VD, A) => VD,
  sendMsg: EdgeTriplet[VD, ED] =>
Iterator[(VertexID,A)],
  mergeMsg: (A, A) => A)
  : Graph[VD, ED]
```

这种基于mrTrilets方法的Pregel模式，与标准Pregel的最大区别是，它的第2段参数体接收的是3个函数参数，而不接收messageList。它不会在单个顶点上进行消息遍历，而是将顶点的多个Ghost副本收到的消息聚合后，发送给Master副本，再使用vprog函数来更新点值。消息的接收和发送都被自动并行化处理，无需担心超级节点的问题。

常见的代码模板如下所示：

```
// 更新顶点
vprog(vid: Long, vert: Vertex, msg: Double):
Vertex = {
  v.score = msg + (1 - ALPHA) * v.weight
}
// 发送消息
sendMsg(edgeTriplet: EdgeTriplet[…]):
Iterator[(Long, Double)]
  (destId, ALPHA * edgeTriplet.srcAttr.
score * edgeTriplet.attr.weight)
}
// 合并消息
mergeMsg(v1: Double, v2: Double): Double = {
  v1+v2
}
```

可以看到，GraphX设计这个模式的用意。它综合了Pregel和GAS两者的优点，即接口相对简单，又保证性能，可以应对点分割的图存储模式，胜任符合幂律分布的自然图的大型计算。另外，值得注意的是，官方的Pregel版本是最简单的一个版本。对于复杂的业务场景，根据这个版本扩展一个定制的Pregel是很常见的做法。

#### 2.2.2.4 图算法工具包

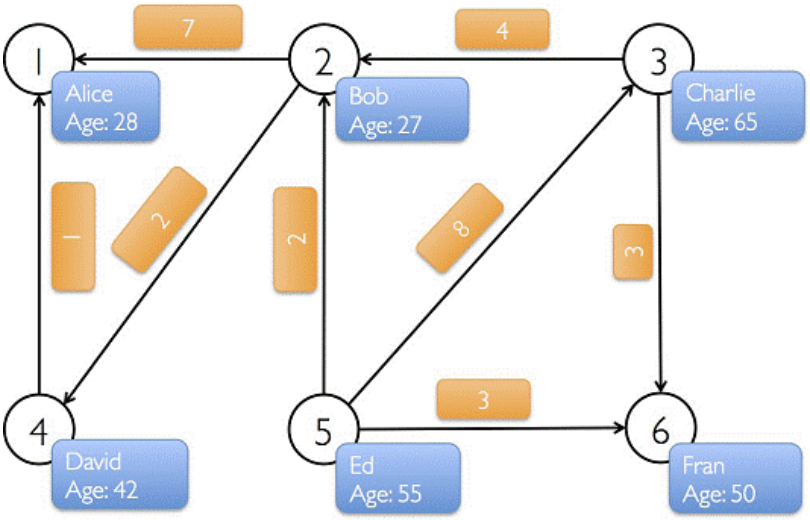
GraphX也提供了一套图算法工具包，方便用户对图进行分析。目前最新版本已支持PageRank、数三角形、最大连通图和最短路径等6种经典的图算法。这些算法的代码实现，目的和重点在于通用性。如果要获得最佳性能，可以参考其实现进行修改和扩展满足业务需求。另外，研读这些代码，也是理解GraphX编程最佳实践的好方法。

## 3、GraphX实例

### 3.1 图例演示

#### 3.1.1 例子介绍

下图中有6个人，每个人有名字和年龄，这些人根据社会关系形成8条边，每条边有其属性。在以下例子演示中将构建顶点、边和图，打印图的属性、转换操作、结构操作、连接操作、聚合操作，并结合实际要求进行演示。




### 3.1.2 程序代码

```
import org.apache.log4j.{Level, Logger}
import org.apache.spark.{SparkContext, SparkConf}
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

object GraphXExample {
  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)

    //设置顶点和边，注意顶点和边都是用元组定义的Array
    //顶点的数据类型是VD:(String,Int)
    val vertexArray = Array(
      (1L, ("Alice", 28)),
      (2L, ("Bob", 27)),
      (3L, ("Charlie", 65)),
      (4L, ("David", 42)),
      (5L, ("Ed", 55)),
      (6L, ("Fran", 50))
    )
    //边的数据类型ED:Int
    val edgeArray = Array(
      Edge(2L, 1L, 7),
      Edge(2L, 4L, 2),
      Edge(3L, 2L, 4),
      Edge(3L, 6L, 3),
      Edge(4L, 1L, 1),
      Edge(5L, 2L, 2),
      Edge(5L, 3L, 8),
      Edge(5L, 6L, 3)
    )

    //构造vertexRDD和edgeRDD
    val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)
    val edgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)

    //构造图Graph[VD,ED]
    val graph: Graph[(String, Int), Int] = Graph(vertexRDD, edgeRDD)

    //******************************************************************************
    //***************************  图的属性    ***************************************
    //******************************************************************************
println("**********************************************")
    println("属性演示")
    println("**********************************************************")
```

```
println("找出图中年龄大于30的顶点：")
graph.vertices.filter { case (id, (name, age)) => age > 30}.collect.foreach {
  case (id, (name, age)) => println(s"$name is $age")
}

//边操作：找出图中属性大于5的边
println("找出图中属性大于5的边：")
  graph.edges.filter(e => e.attr > 5).collect.foreach(e => println(s"${e.srcId} to ${e.dstId}
  att ${e.attr}"))
println

//triplets操作，((srcId, srcAttr), (dstId, dstAttr), attr)
println("列出边属性>5的tripltes：")
for (triplet <- graph.triplets.filter(t => t.attr > 5).collect) {
  println(s"${triplet.srcAttr._1} likes ${triplet.dstAttr._1}")
}
println

//Degrees操作
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))
println

//******************************************************************************
//*************************** 转换操作 ********************************************
//******************************************************************************
println("*************************************************************")
println("转换操作")
println("*************************************************************")
println("顶点的转换操作，顶点age + 10：")
graph.mapVertices{ case (id, (name, age)) => (id, (name,
  age+10))}.vertices.collect.foreach(v => println(s"${v._2._1} is ${v._2._2}"))
println
println("边的转换操作，边的属性*2：")
graph.mapEdges(e=>e.attr*2).edges.collect.foreach(e => println(s"${e.srcId} to ${e.dstId
  att ${e.attr}"))
println

 //******************************************************************************
//*************************** 结构操作 ********************************************
//******************************************************************************
println("*************************************************************")
println("结构操作")
println("*************************************************************")
println("顶点年纪>30的子图：")
val subGraph = graph.subgraph(vpred = (id, vd) => vd._2 >= 30)
println("子图所有顶点：")
subGraph.vertices.collect.foreach(v => println(s"${v._2._1} is ${v._2._2}"))
println
println("子图所有边：")
subGraph.edges.collect.foreach(e => println(s"${e.srcId} to ${e.dstId} att ${e.attr}"))
println


 //******************************************************************************
//*************************** 连接操作 ********************************************
//******************************************************************************
println("*************************************************************")
println("连接操作")
println("*************************************************************")
val inDegrees: VertexRDD[Int] = graph.inDegrees
case class User(name: String, age: Int, inDeg: Int, outDeg: Int)

//创建一个新图，顶点VD的数据类型为User，并从graph做类型转换
val initialUserGraph: Graph[User, Int] = graph.mapVertices { case (id, (name, age)) =>
```

```
  User(name, age, 0, 0)}

//initialUserGraph与inDegrees、outDegrees（RDD）进行连接，并修改initialUserGraph中
  inDeg值、outDeg值
val userGraph = initialUserGraph.outerJoinVertices(initialUserGraph.inDegrees) {
  case (id, u, inDegOpt) => User(u.name, u.age, inDegOpt.getOrElse(0), u.outDeg)
}.outerJoinVertices(initialUserGraph.outDegrees) {
  case (id, u, outDegOpt) => User(u.name, u.age, u.inDeg,outDegOpt.getOrElse(0))
}

println("连接图的属性：")
  userGraph.vertices.collect.foreach(v => println(s"${v._2.name} inDeg: ${v._2.inDeg}
  outDeg: ${v._2.outDeg}"))
println

println("出度和入读相同的人员：")
userGraph.vertices.filter {
  case (id, u) => u.inDeg == u.outDeg
}.collect.foreach {
  case (id, property) => println(property.name)
}
println

  //*************************************************************************************
//***************************  聚合操作   ***************************************
//*************************************************************************************
println("*********************************************************")
println("聚合操作")
println("*********************************************************")
println("找出年纪最大的追求者：")
val oldestFollower: VertexRDD[(String, Int)] = userGraph.mapReduceTriplets[(String, Int)]
  (
  // 将源顶点的属性发送给目标顶点，map过程
  edge => Iterator((edge.dstId, (edge.srcAttr.name, edge.srcAttr.age))),
  // 得到最大追求者，reduce过程
  (a, b) => if (a._2 > b._2) a else b
)

userGraph.vertices.leftJoin(oldestFollower) { (id, user, optOldestFollower) =>
  optOldestFollower match {
    case None => s"${user.name} does not have any followers."
    case Some((name, age)) => s"${name} is the oldest follower of ${user.name}."
  }
}.collect.foreach { case (id, str) => println(str)}
println

 //*************************************************************************************
//***************************  实用操作   ***************************************
//*************************************************************************************
println("*********************************************************")
println("聚合操作")
println("*********************************************************")
println("找出5到各顶点的最短：")
val sourceId: VertexId = 5L // 定义源点
val initialGraph = graph.mapVertices((id, _) => if (id == sourceId) 0.0 else
  Double.PositiveInfinity)
val sssp = initialGraph.pregel(Double.PositiveInfinity)(
  (id, dist, newDist) => math.min(dist, newDist),
  triplet => {  // 计算权重
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {
      Iterator.empty
    }
  },
  (a,b) => math.min(a,b) // 最短距离
)
println(sssp.vertices.collect.mkString("\n"))
```

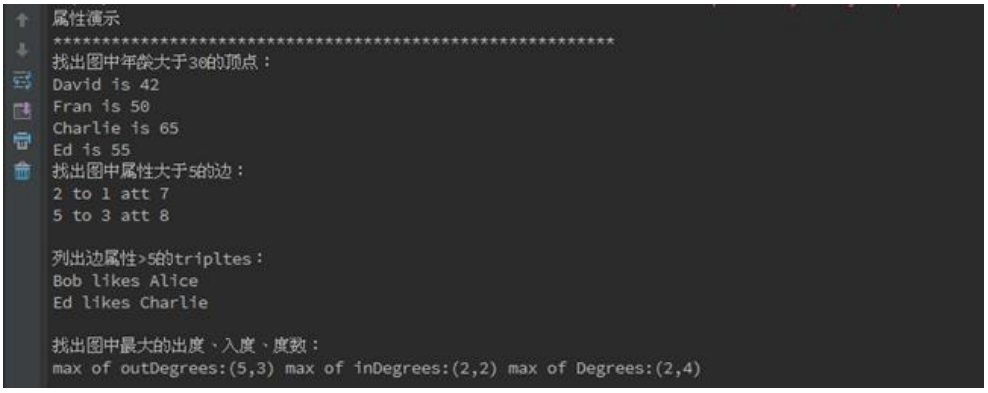
```
    sc.stop()
  }
}
```

**3.1.3 运行结果**

在IDEA（如何使用IDEA参见第3课《3.Spark编程模型（下）--IDEA搭建及实战》）中首先对GraphXExample.scala代码进行编译，编译通过后进行执行，执行结果如下：

```
*************************************************************
属性演示
*************************************************************
找出图中年龄大于30的顶点：
David is 42
Fran is 50
Charlie is 65
Ed is 55
找出图中属性大于5的边：
2 to 1 att 7
5 to 3 att 8

列出边属性>5的tripltes：
Bob likes Alice
Ed likes Charlie

找出图中最大的出度、入度、度数：
max of outDegrees:(5,3) max of inDegrees:(2,2) max of Degrees:(2,4)

*************************************************************
转换操作
*************************************************************
顶点的转换操作，顶点age + 10：
4 is (David,52)
1 is (Alice,38)
6 is (Fran,60)
3 is (Charlie,75)
5 is (Ed,65)
2 is (Bob,37)

边的转换操作，边的属性*2：
2 to 1 att 14
2 to 4 att 4
3 to 2 att 8
3 to 6 att 6
4 to 1 att 2
5 to 2 att 4
5 to 3 att 16
5 to 6 att 6

*************************************************************
结构操作
*************************************************************
顶点年纪>30的子图：
子图所有顶点：
David is 42
Fran is 50
Charlie is 65
Ed is 55

子图所有边：
3 to 6 att 3
5 to 3 att 8
5 to 6 att 3

*************************************************************
连接操作
*************************************************************
连接图的属性：
David inDeg: 1  outDeg: 1
Alice inDeg: 2  outDeg: 0
```

```
Fran inDeg: 2  outDeg: 0
Charlie inDeg: 1  outDeg: 2
Ed inDeg: 0  outDeg: 3
Bob inDeg: 2  outDeg: 2

出度和入读相同的人员：
David
Bob

**********************************************************
聚合操作
**********************************************************
找出年纪最大的追求者：
Bob is the oldest follower of David.
David is the oldest follower of Alice.
Charlie is the oldest follower of Fran.
Ed is the oldest follower of Charlie.
Ed does not have any followers.
Charlie is the oldest follower of Bob.

**********************************************************
实用操作
**********************************************************
找出5到各顶点的最短：
(4,4.0)
(1,5.0)
(6,3.0)
(3,8.0)
(5,0.0)
(2,2.0)
```




### 3.2 PageRank 演示

#### 3.2.1 例子介绍

PageRank, 即网页排名，又称网页级别、Google 左侧排名或佩奇排名。它是Google 创始人拉里· 佩奇和谢尔盖· 布林于1997 年构建早期的搜索系统原型时提出的链接分析算法。目前很多重要的链接分析算法都是在PageRank 算法基础上衍生出来的。PageRank 是Google 用于用来标识网页的等级/ 重要性的一种方法，是Google 用来衡量一个网站的好坏的唯一标准。在揉合了诸如Title 标识和Keywords 标识等所有其它因素之后， Google 通过PageRank 来调整结果，使那些更具“等级/ 重要性”的网页在搜索结果中令网站排名获得提升，从而提高搜索结果的相关性和质量。

这是Google最核心的算法，用于给每个网页价值评分，是Google“在垃圾中找黄”的关键算法，这个算法成就了今天的Google

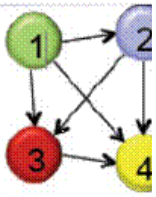
PageRank vector **q** is defined as $q = Gq$

where $G = \alpha S + (1-\alpha)\frac{1}{n}U$

- S is the destination-by-source stochastic matrix,
- U is all one matrix.
- n is the number of nodes
- α is the weight between 0 and 1 (e.g., 0.85)

Algorithm: Iterative powering for finding the first eigen-vector

$$q^{next} = Gq^{cur}$$




$$G = \begin{bmatrix} 0 & 0 & 0 \\ 1/3 & 0 & 0 \\ 1/3 & 1/2 & 0 \\ 1/3 & 1/2 & 1 \end{bmatrix}$$

#### 3.2.2 测试数据

在这里测试数据为顶点数据graphx-wiki-vertices.txt和边数据graphx-wiki-edges.txt，可以在本系列附带资源/data/class9/目录中找到这两个数据文件，其中格式为：

- 顶点为顶点编号和网页标题

```
138  3932908833397101503 St. Michael's College School
139  2708418598117237725 Metropolitan Junior Hockey League
140  5090956282167233219 List of mountain ranges of California
141  4102223989096646779 Java (programming language)
```

- 边数据由两个顶点构成

```
 98  1746517089350976281 443952852637640503
 99  1746517089350976281 497048819723143676
100  1746517089350976281 533544275833168761
```

#### 3.2.3 程序代码

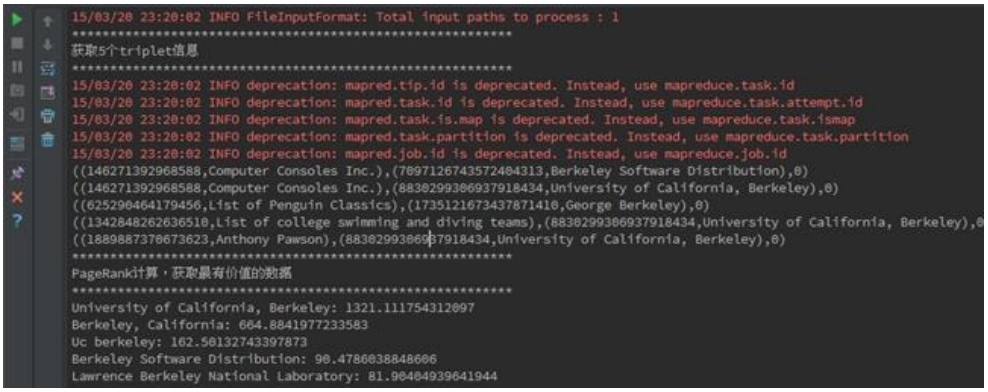
```
import org.apache.log4j.{Level, Logger}
import org.apache.spark.{SparkContext, SparkConf}
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

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)

    //读入数据文件
    val articles: RDD[String] = sc.textFile("/home/hadoop/IdeaProjects/data/graphx/graphx-
      wiki-vertices.txt")
    val links: RDD[String] = sc.textFile("/home/hadoop/IdeaProjects/data/graphx/graphx-
      wiki-edges.txt")

    //装载顶点和边
    val vertices = articles.map { line =>
      val fields = line.split('\t')
      (fields(0).toLong, fields(1))
    }

    val edges = links.map { line =>
      val fields = line.split('\t')
      Edge(fields(0).toLong, fields(1).toLong, 0)
    }

    //cache操作
```

```
    //val graph = Graph(vertices, edges, "").persist(StorageLevel.MEMORY_ONLY_SER)
    val graph = Graph(vertices, edges, "").persist()
    //graph.unpersistVertices(false)

    //测试
    println("*************************************************************")
    println("获取5个triplet信息")
    println("*************************************************************")
    graph.triplets.take(5).foreach(println(_))

    //pageRank算法里面的时候使用了cache()，故前面persist的时候只能使用MEMORY_ONLY
    println("*************************************************************")
    println("PageRank计算，获取最有价值的数据")
    println("*************************************************************")
    val prGraph = graph.pageRank(0.001).cache()

    val titleAndPrGraph = graph.outerJoinVertices(prGraph.vertices) {
      (v, title, rank) => (rank.getOrElse(0.0), title)
    }

    titleAndPrGraph.vertices.top(10) {
      Ordering.by((entry: (VertexId, (Double, String))) => entry._2._1)
    }.foreach(t => println(t._2._2 + ": " + t._2._1))

    sc.stop()
  }
}
```

**3.2.4 运行结果**

在IDEA中首先对PageRank.scala代码进行编译，编译通过后进行执行，执行结果如下：

```
*************************************************************
获取5个triplet信息
*************************************************************
((146271392968588,Computer Consoles Inc.),(7097126743572404313,Berkeley Software
Distribution),0)
((146271392968588,Computer Consoles Inc.),(8830299306937918434,University of Californ
Berkeley),0)
((625290464179456,List of Penguin Classics),(1735121673437871410,George Berkeley),0)
((1342848262636510,List of college swimming and diving teams),
(8830299306937918434,University of California, Berkeley),0)
((1889887370673623,Anthony Pawson),(8830299306937918434,University of California,
Berkeley),0)

*************************************************************
PageRank计算，获取最有价值的数据
*************************************************************
University of California, Berkeley: 1321.111754312097
Berkeley, California: 664.8841977233583
Uc berkeley: 162.50132743397873
Berkeley Software Distribution: 90.4786038848606
Lawrence Berkeley National Laboratory: 81.90404939641944
George Berkeley: 81.85226118457985
Busby Berkeley: 47.871998218019655
Berkeley Hills: 44.76406979519754
Xander Berkeley: 30.324075347288037
Berkeley County, South Carolina: 28.908336483710308
```




**4、参考资料**

（1）《GraphX: 基于Spark的弹性分布式图计算系统》 http://lidrema.blog.163.com/blog/static/20970214820147199643788/

（2）《快刀初试：Spark GraphX在淘宝的实践》 http://www.csdn.net/article/2014-0807/2821097



分类：20.Spark入门实战系列





« 上一篇：Spark入门实战系列--8.Spark MLlib（下）--机器学习库SparkMLlib实战
» 下一篇：新版本来袭：Apache Spark 1.5新特性介绍

posted @ 2015-09-14 08:59 shishanyuan 阅读(19766) 评论(1) 编辑 收藏

发表评论

#1楼 2016-12-29 22:00 | littlezz

老师您好，我刚刚开始学spark graphx，看了您的帖子感到受益匪浅，按照您的一系列教程，我安装了IDEA，但在行您的代码时，出现了错误，是说GraphXExample is already defined as object GraphXExample，恕我刚刚接park和Scala，我确实不知道这个问题代表的是什么意思，百度了也没有答案，您能告诉一下这个问题是出自何处，该如何解决吗？谢谢您！

支持(0) 反对

刷新评论 刷新页面 返回顶

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新**IT**新闻:
· AI校招程序员最高薪酬曝光！腾讯80万年薪领跑，还送北京户口
· 王健林讲话完整版曝光：万达苏宁明年将在资本方面有动作
· 比特币网创始人卖掉所有比特币：投资风险太高
· 支付宝福利：免费扫码领红包 赏金翻倍
· Opera Software更名为Otello公司
» 更多新闻...



最新知识库文章:
· 以操作系统的角度述说线程与进程
· 软件测试转型之路
· 门内门外看招聘
· 大道至简，职场上做人做事做管理
· 关于编程，你的练习是不是有效的？
» 更多知识库文章...