亚太研究院

Spark 官方文档翻译

图计算编程指南

(V1.1.0)

翻译者　吴卓华

Spark 官方文档翻译团成员

前 言

世界上第一个Spark 1.1.0 中文文档问世了！

伴随着大数据相关技术和产业的逐步成熟，继Hadoop之后，Spark技术以集大成的无可比拟的优势，发展迅速，将成为替代Hadoop的下一代云计算、大数据核心技术。

Spark是当今大数据领域最活跃最热门的高效大数据通用计算平台，基于RDD，Spark成功的构建起了一体化、多元化的大数据处理体系，在“One Stack to rule them all”思想的引领下，Spark成功的使用Spark SQL、Spark Streaming、MLLib、GraphX近乎完美的解决了大数据中Batch Processing、Streaming Processing、Ad-hoc Query等三大核心问题，更为美妙的是在Spark中Spark SQL、Spark Streaming、MLLib、GraphX四大子框架和库之间可以无缝的共享数据和操作，这是当今任何大数据平台都无可匹敌的优势。

在实际的生产环境中，世界上已经出现很多一千个以上节点的Spark集群，以eBay为例，eBay的Spark集群节点已经超过2000个，Yahoo 等公司也在大规模的使用Spark，国内的淘宝、腾讯、百度、网易、京东、华为、大众点评、优酷土豆等也在生产环境下深度使用Spark。2014 Spark Summit上的信息，Spark已经获得世界 20 家顶级公司的支持，这些公司中包括Intel、IBM等，同时更重要的是包括了最大的四个Hadoop发行商，都提供了对Spark非常强有力的支持。

与Spark火爆程度形成鲜明对比的是Spark人才的严重稀缺，这一情况在中国尤其严重，这种人才的稀缺，一方面是由于Spark技术在 2013、2014 年才在国内的一些大型企业里面被逐步应用，另一方面是由于匮乏Spark相关的中文资料和系统化的培训。为此，Spark亚太研究院和 51CTO联合推出了“Spark亚太研究院决胜大数据时代 100 期公益大讲堂”，来推动Spark技术在国内的普及及落地。

具体视频信息请参考 http://edu.51cto.com/course/course_id-1659.html

与此同时，为了向Spark学习者提供更为丰富的学习资料，Spark亚太研究院发起并号召，结合网络社区的力量构建了Spark中文文档专家翻译团队，历经 1 个月左右的艰苦努力和反复修改，Spark中文文档V1.1 终于完成。尤其值得一提的是，在此次中文文档的翻译期间，Spark官方团队发布了Spark 1.1.0 版本，为了让学习者了解到最新的内容，Spark中文文档专家翻译团队主动提出基于最新的Spark 1.1.0 版本，更新了所有已完成的翻译内容，在此，我谨代表Spark亚太研究院及广大Spark学习爱好者向专家翻译团队所有成员热情而专业的工作致以深刻的敬意！

当然，作为世界上第一份相对系统的Spark中文文档，不足之处在所难免，大家有任何建议或者意见都可以发邮件到marketing@sparkinchina.com ;同时如果您想加入Spark中文文档翻译团队，也请发邮件到marketing@sparkinchina.com进行申请；Spark中文文档的翻译是一个持续更新的、不断版本迭代的过程，我们会尽全力给大家

提供更高质量的Spark中文文档翻译。

最后，也是最重要的，请允许我荣幸的介绍一下我们的Spark中文文档第一个版本翻译的专家团队成员，他们分别是（排名不分先后）：

- 傅智勇，《快速开始(v1.1.0)》（和唐海东翻译的是同一主题，大家可以对比参考）
- 吴洪泽,《Spark机器学习库（v1.1.0)》（其中聚类和降维部分是蔡立宇翻译）
- 武扬，《在Yarn上运行Spark (v1.1.0)》《Spark 调优(v1.1.0)》
- 徐骄，《Spark配置(v1.1.0)》《Spark SQL编程指南(v1.1.0)》（Spark SQL和韩保礼翻译的是同一主题，大家可以对比参考）
- 蔡立宇，《Bagel 编程指南(v1.1.0)》
- harli，《Spark 编程指南（v1.1.0)》
- 吴卓华，《图计算编程指南(1.1.0)》
- 樊登贵，《EC2(v1.1.0)》《Mesos(v1.1.0)》
- 韩保礼，《Spark SQL编程指南(v1.1.0)》（和徐骄翻译的是同一主题，大家可以对比参考）
- 颜军，《文档首页(v1.1.0)》
- Jack Niu，《Spark实时流处理编程指南(v1.1.0)》
- 俞杭军，《sbt-assembly》《使用Maven编译Spark(v1.1.0)》
- 唐海东，《快速开始(v1.1.0)》（和傅智勇翻译的是同一主题，大家可以对比参考）
- 刘亚卿，《硬件配置(v1.1.0)》《Hadoop 第三方发行版(v1.1.0)》《给Spark提交代码(v1.1.0)》
- 耿元振《集群模式概览(v1.1.0)》《监控与相关工具(v1.1.0)》《提交应用程序(v1.1.0)》
- 王庆刚，《Spark作业调度(v1.1.0)》《Spark安全(v1.1.0)》
- 徐敬丽，《Spark Standalone 模式 (v1.1.0)》

另外关于Spark API的翻译正在进行中，敬请大家关注。

Life is short, You need Spark!

Spark亚太研究院院长 王家林
2014 年 10 月

Spark 亚太研究院决胜大数据时代 100 期公益大讲堂

简 介

作为下一代云计算的核心技术，Spark性能超Hadoop百倍，算法实现仅有其 1/10 或 1/100,是可以革命Hadoop的目前唯一替代者，能够做Hadoop做的一切事情，同时速度比Hadoop快了 100 倍以上。目前Spark已经构建了自己的整个大数据处理生态系统，国外一些大型互联网公司已经部署了Spark。甚至连Hadoop的早期主要贡献者Yahoo现在也在多个项目中部署使用Spark；国内的淘宝、优酷土豆、网易、Baidu、腾讯、皮皮网等已经使用Spark技术用于自己的商业生产系统中，国内外的应用开始越来越广泛。Spark正在逐渐走向成熟，并在这个领域扮演更加重要的角色，刚刚结束的 2014 Spark Summit上的信息，Spark已经获得世界 20 家顶级公司的支持，这些公司中包括Intel、IBM等，同时更重要的是包括了最大的四个Hadoop发行商都提供了对非常强有力的支持Spark的支持.

鉴于Spark的巨大价值和潜力，同时由于国内极度缺乏Spark人才，Spark亚太研究院在完成了对Spark源码的彻底研究的同时，不断在实际环境中使用Spark的各种特性的基础之上，推出了Spark亚太研究院决胜大数据时代 100 期公益大讲堂，希望能够帮助大家了解Spark的技术。同时，对Spark人才培养有近一步需求的企业和个人，我们将以公开课和企业内训的方式，来帮助大家进行Spark技能的提升。同样，我们也为企业提供一体化的顾问式服务及Spark一站式项目解决方案和实施方案。

Spark亚太研究院决胜大数据时代 100 期公益大讲堂是国内第一个Spark课程免费线上讲座，每周一期，从 7 月份起，每周四晚 20:00-21:30，与大家不见不散！老师将就Spark内核剖析、源码解读、性能优化及商业实战案例等精彩内容与大家分享，干货不容错过！

时间：从 7 月份起，每周一期，每周四晚 20:00-21:30
形式：腾讯课堂在线直播
学习条件：对云计算大数据感兴趣的技术人员
课程学习地址：http://edu.51cto.com/course/course_id-1659.html

图计算编程指南(V1.1.0)

（翻译者：吴卓华）

GraphX Programming Guide，原文档链接：

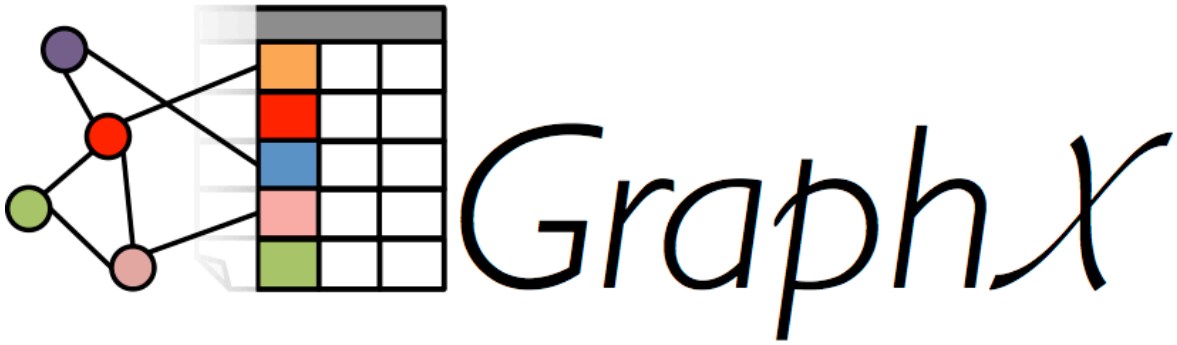
http://spark.apache.org/docs/latest/graphx-programming-guide.html

目录

第一章 Graphx 编程指南

http://spark.apache.org/docs/latest/graphx-programming-guide.html

1.1.0



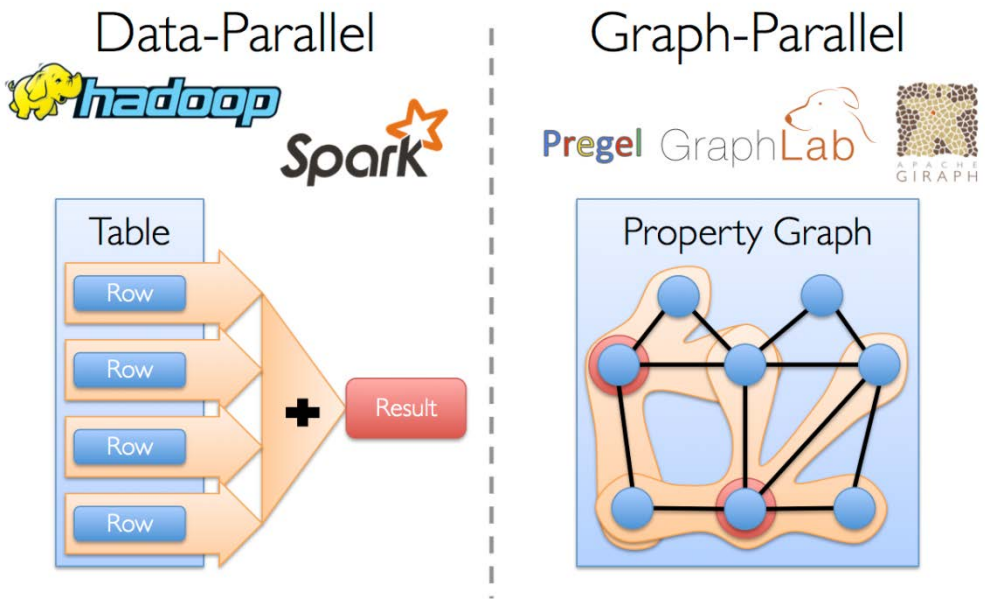
1.1 概述

GraphX 是新的(alpha)的图形和图像并行计算的Spark API。从整理上看，GraphX 通过引入 弹性分布式属性图(Resilient Distributed Property Graph)继承了Spark RDD：一个将有效信息放在顶点和边的有向多重图。为了支持图形计算，GraphX 公开了一组基本的运算（例如，subgraph，joinVertices和 mapReduceTriplets），以及在一个优化后的PregelAPI的变形。此外，GraphX 包括越来越多的图像算法和 builder 构造器，以简化图形分析任务。

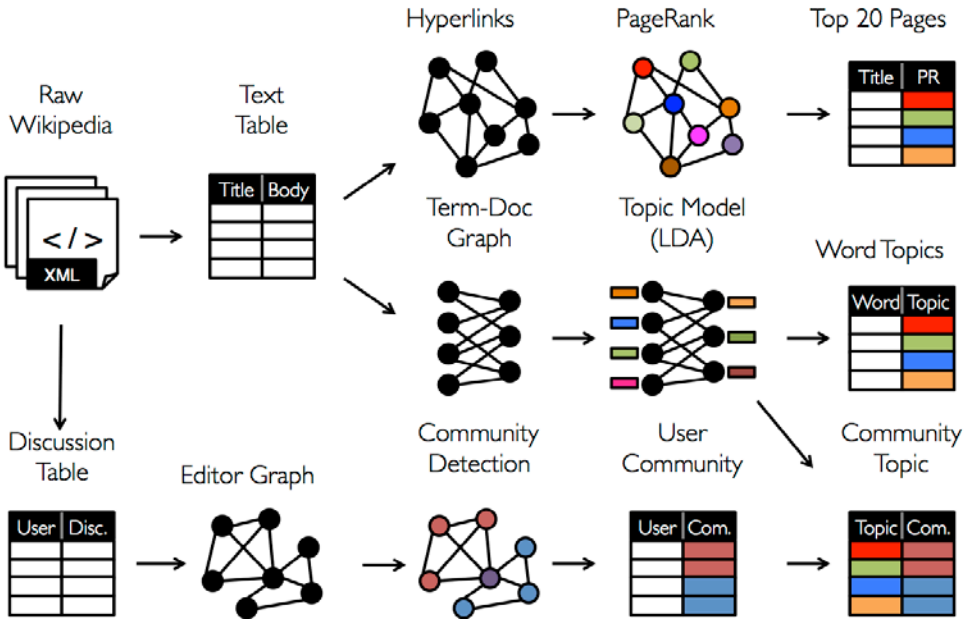
GraphX 目前是一个 alpha 组件。虽然我们会尽量减少 API 的变化，但是一些 API 可能会在将来的版本中改变。

1.2 图并行计算的背景

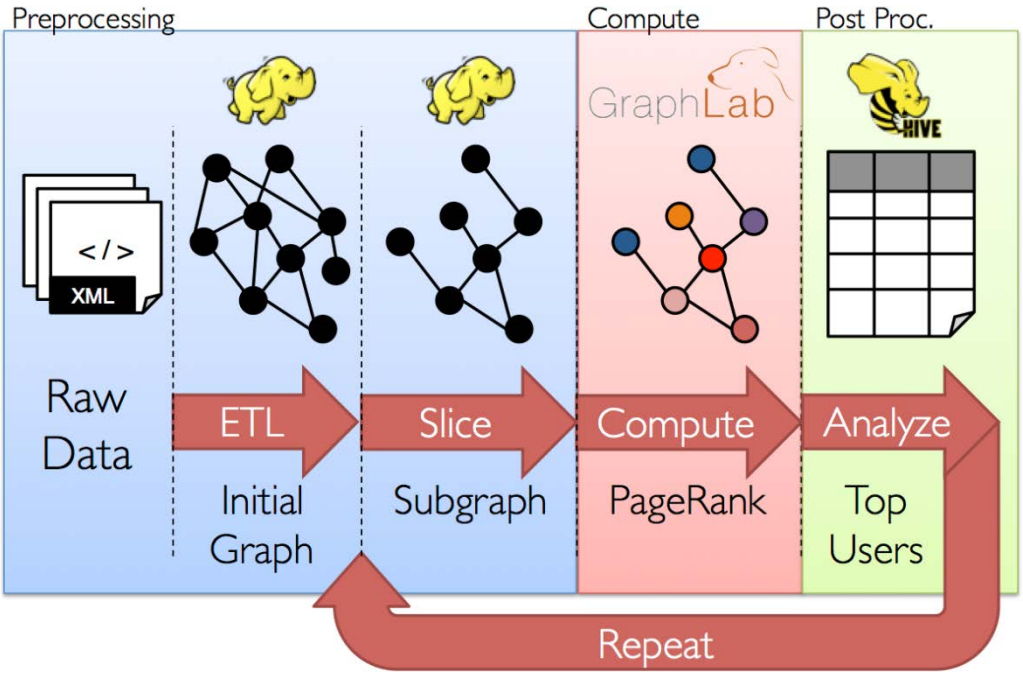
从社交网络到语言建模，日益扩大的规模和图形数据的重要性已带动许多新的图像并行系统（例如，Giraph和 GraphLab）。通过限制可表示计算的类型以及引入新的技术来划分和分布图，这些系统比一般的数据并行系统在执行复杂图形算法方面有大幅度地提高。



然而，这些限制在获得重大性能提升的同时，也使其难以表达一个典型的图表分析流程中的许多重要阶段：构造图，修改它的结构或表达计算跨越多重图的计算。此外，如何看待数据取决于我们的目标，相同的原始数据，可能有许多不同的表(table)和图表视图(graph views)。



因此，能够在同一组物理数据的表和图表视图之间切换是很有必要的，并利用各视图的属性，以方便地和有效地表达计算。但是，现有的图形分析管道必须由图并行和数据并行系统组成，从而导致大量的数据移动和重复以及复杂的编程模型。



该 GraphX 项目的目标是建立一个系统，建立一个统一的图和数据并行计算的 API。该 GraphX API 使用户能够将数据既可以当作一个图，也可以当作集合（即 RDDS）而不用进行数据移动或数据复制。通过引入在图并行系统中的最新进展，GraphX 能够优化图形操作的执行。

1.3 GraphX 替换 Spark Bagel 的 API

在GraphX 的发布之前，Spark的图计算是通过Bagel实现的，后者是Pregel的一个具体实现。GraphX提供了更丰富的图属性API，从而增强了Bagel。从而达到一个更加精简的Pregel抽象，系统优化，性能提升以及减少内存开销。虽然我们计划最终弃用Bagel，我们将继续支持 Bagel的API和 Bagel编程指南。不过，我们鼓励Bagel用户，探索新的GraphX API，并就从Bagel升级中遇到的障碍反馈给我们。

1.4 从 Spark 0.9.1 迁移

GraphX 在 Spark 1.1.0 包含 Spark 0.9.1 一个用户面向接口的改变。EdgeRDD 现在可以存储相邻顶点属性来构建 triplets，因此它获得了一个类型参数。一个 Graph[VD，ED] 的边的类型是 EdgeRDD[ED，VD]而不是 EdgeRDD[ED]。

入门

首先，你要导入 Spark 和 GraphX 到你的项目，如下所示：

```
import org.apache.spark._
import org.apache.spark.graphx._
// To make some of the examples work we will also need RDD
import org.apache.spark.rdd.RDD
```

如果你不使用Spark shell，你还需要一个 SparkContext。要了解更多有关如何开始使用Spark参考 Spark快速入门指南。

属性图

该 属性图是一个用户定义的顶点和边的有向多重图。有向多重图是一个有向图，它可能有多个平行边共享相同的源和目的顶点。多重图支持并行边的能力简化了有多重关系（例如，同事和朋友）的建模场景。每个顶点是 *唯*一的 64 位长的标识符（VertexID）作为主键。GraphX 并没有对顶点添加任何顺序的约束。同样，每条边具有相应的源和目的顶点的标识符。

该属性表的参数由顶点（VD）和边缘（ED）的类型来决定。这些是分别与每个顶点和边相关联的对象的类型。

GraphX 优化顶点和边的类型的表示方法，当他们是普通的旧的数据类型(例如，整数，双精度等）通过将它们存储在专门的阵列减小了在内存占用量。

在某些情况下，可能希望顶点在同一个图中有不同的属性类型。这可以通过继承来实现。例如，以用户和产品型号为二分图我们可以做到以下几点：

```
class VertexProperty()
case class UserProperty(val name: String) extends VertexProperty
case class ProductProperty(val name: String, val price: Double) extends VertexProperty
// The graph might then have the type:
var graph: Graph[VertexProperty, String] = null
```

和 RDDS 一样，属性图是不可变的，分布式的和容错的。对图中的值或结构的改变是通过生成具有所需更改的新图来完成的。注意原始图的该主要部分（即不受影响的结构，属性和索引）被重用，从而减少这个数据结构的成本。该图是通过启发式执行顶点分区，在不同的执行器(executor)中进行顶点的划分。与 RDDS 一样，在发生故障的情况下，图中的每个分区都可以重建。

逻辑上讲，属性图对应于一对类型集合（RDDS），这个组合记录顶点和边的属性。因此，该图表类包含成员访问该图的顶点和边：

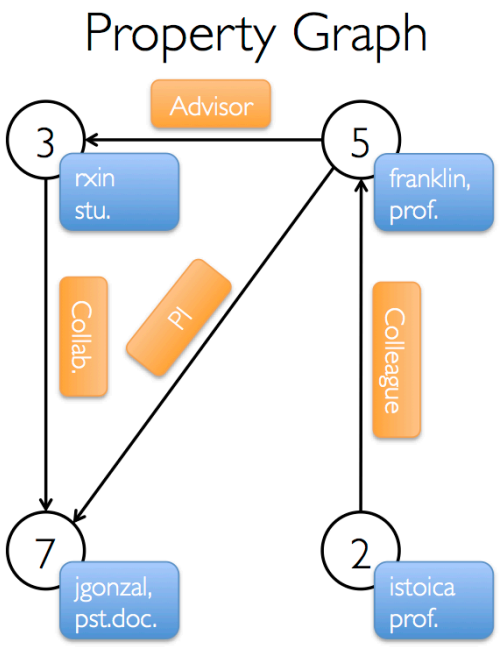
```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
```

```
  val edges: EdgeRDD[ED, VD]
}
```

类 VertexRDD [VD]和 EdgeRDD [ED，VD]继承和并且分别是一个优化的版本的 RDD[(VertexID,VD)]和 RDD[Edge[ED]]。这两个 VertexRDD[VD]和 EdgeRDD[ED, VD]提供各地图的计算内置附加功能，并充分利用内部优化。我们在上一节顶点和边 RDDS 中详细讨论了 VertexRDD 和 EdgeRDD 的 API，但现在，他们可以简单地看成是 RDDS 形式的： RDD[(VertexID,VD)]和 RDD [EDGE[ED]] 。

1.4.1 属性图的例子

假设我们要建立一个 GraphX 项目各合作者的属性图。顶点属性可能会包含用户名和职业。我们可以使用一组字符注释来描述代表合作者关系的边：



Vertex Table

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

Edge Table

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

由此产生的图形将有类型签名：

```
val userGraph: Graph[(String, String), String]
```

有许多方法可以从原始数据文件，RDDS，甚至合成生成器来生成图，我们会在 graph builders 更详细的讨论。可能是最通用的方法是使用 Graph ojbect。例如，下面的代码从一系列的 RDDS 的集合中构建图：

```
// Assume the SparkContext has already been constructed
val sc: SparkContext
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
```

```
                    (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
                    Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
```

在上面的例子中，我们利用了 Edge 的 case 类。Edge 具有 srcId 和 dstId，它们分别对应于源和目的地顶点的标识符。此外，Edge 类具有 attr 属性，并存储的边的特性。

我们可以通过 graph.vertices 和 graph.edges 属性 得到图到各自的顶点和边的视图。

```
val graph: Graph[(String, String), String] // Constructed from above
// Count all users which are postdocs
graph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count
// Count all the edges where src > dst
graph.edges.filter(e => e.srcId > e.dstId).count
```

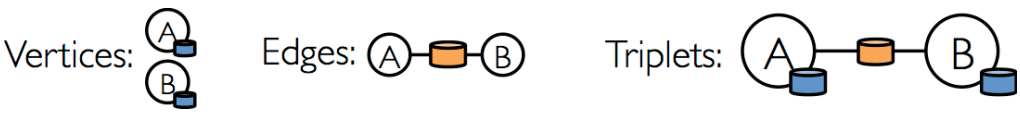
需要注意的是 graph.vertices 返回 VertexRDD[(String, String)]延伸 RDD[(VertexID，(String ,String))] ,所以我们使用 Scala 的 case 表达来解构元组。在另一方面 ,graph.edges 返回 EdgeRDD 包含 Edge[String]对象。我们可以也使用的如下的类型的构造器：

```
graph.edges.filter { case Edge(src, dst, prop) => src > dst }.count
```

除了 图的顶点和边的意见，GraphX 也提供了三重视图。三重视图逻辑连接点和边的属性产生的RDD[EdgeTriplet [VD，ED]包含的实例 EdgeTriplet类。此*连接*可以表示如下的SQL表达式：

```
SELECT src.id, dst.id, src.attr, e.attr, dst.attr
FROM edges AS e LEFT JOIN vertices AS src, vertices AS dst
ON e.srcId = src.Id AND e.dstId = dst.Id
```

或图形方式：



该 EdgeTriplet类继承了 Edge并加入了类属性: srcAttr和 dstAttr,用于包含了源和目标属性。我们可以用一个图的三元组视图渲染描述用户之间的关系字符串的集合。

```
val graph: Graph[(String, String), String] // Constructed from above
// Use the triplets view to create an RDD of facts.
val facts: RDD[String] =
  graph.triplets.map(triplet =>
    triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1)
facts.collect.foreach(println(_))
```

Graph 操作

正如RDDs有这样基本的操作 mpa，filter，和 reduceByKey，属性图也有一系列基本的运算,采用用户定义的函数，并产生新的图形与变换的性质和结构。 定义核心运算已优化的实现方式中定义的 Graph，并且被表示为核心操作的组合定义在 GraphOps。然而，由于Scala的implicits特性, GraphOps中的操作会自动作为Graph的成员。例如，我们可以计算各顶点的入度（定义在的 GraphOps）：

```
val graph: Graph[(String, String), String]
// Use the implicit GraphOps.inDegrees operator
val inDegrees: VertexRDD[Int] = graph.inDegrees
```

将核心图操作和 GraphOps区分开来的原因是为了将来能够支持不同的图表示。每个图的表示必须实现核心操作并且复用 GraphOps中很多有用的操作。

1.4.2 运算列表总结

以下列出了Graph图 和 GraphOps中同时定义的操作.为了简单起见,我们都定义为Graph的成员函数。请注意，某些函数签名已被简化（例如，默认参数和类型的限制被删除了）,还有一些更高级的功能已被删除，完整的列表,请参考API文档。

```
/** Summary of the functionality in the property graph */
class Graph[VD, ED] {
  // Information about the Graph
  ==================================================================
  val numEdges: Long
  val numVertices: Long
  val inDegrees: VertexRDD[Int]
  val outDegrees: VertexRDD[Int]
  val degrees: VertexRDD[Int]
  // Views of the graph as collections
  ================================================================
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED, VD]
  val triplets: RDD[EdgeTriplet[VD, ED]]
  // Functions for caching graphs
  ================================================================
```

```
  def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD, ED]
  def cache(): Graph[VD, ED]
  def unpersistVertices(blocking: Boolean = true): Graph[VD, ED]
              //       Change       the       partitioning       heuristic
============================================================
  def partitionBy(partitionStrategy: PartitionStrategy): Graph[VD, ED]
           //      Transform      vertex      and      edge      attributes
============================================================
  def mapVertices[VD2](map: (VertexID, VD) => VD2): Graph[VD2, ED]
  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
  def mapEdges[ED2](map: (PartitionID, Iterator[Edge[ED]]) => Iterator[ED2]):
Graph[VD, ED2]
  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
   def mapTriplets[ED2](map: (PartitionID, Iterator[EdgeTriplet[VD, ED]]) =>
Iterator[ED2])
    : Graph[VD, ED2]
                //       Modify       the       graph       structure
=====================================================================
  def reverse: Graph[VD, ED]
  def subgraph(
      epred: EdgeTriplet[VD,ED] => Boolean = (x => true),
      vpred: (VertexID, VD) => Boolean = ((v, d) => true))
    : Graph[VD, ED]
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
  def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
                //       Join       RDDs       with       the       graph
=====================================================================
  def joinVertices[U](table: RDD[(VertexID, U)])(mapFunc: (VertexID, VD, U) =>
VD): Graph[VD, ED]
  def outerJoinVertices[U, VD2](other: RDD[(VertexID, U)])
      (mapFunc: (VertexID, VD, Option[U]) => VD2)
    : Graph[VD2, ED]
          //     Aggregate     information     about     adjacent     triplets
================================================
             def      collectNeighborIds(edgeDirection:      EdgeDirection):
VertexRDD[Array[VertexID]]
              def       collectNeighbors(edgeDirection:       EdgeDirection):
VertexRDD[Array[(VertexID, VD)]]
  def mapReduceTriplets[A: ClassTag](
      mapFunc: EdgeTriplet[VD, ED] => Iterator[(VertexID, A)],
      reduceFunc: (A, A) => A,
      activeSetOpt: Option[(VertexRDD[_], EdgeDirection)] = None)
    : VertexRDD[A]
```

```
  // Iterative graph-parallel computation
  ========================================================
  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]
  // Basic graph algorithms
  ========================================================================
  def pageRank(tol: Double, resetProb: Double = 0.15): Graph[Double, Double]
  def connectedComponents(): Graph[VertexID, ED]
  def triangleCount(): Graph[Int, ED]
  def stronglyConnectedComponents(numIter: Int): Graph[VertexID, ED]
}
```

1.5 属性操作

和 RDD 的 map 操作类似，属性图包含以下内容：

```
class Graph[VD, ED] {
  def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
}
```

每个运算产生一个新的图,这个图的顶点和边属性通过 map 方法修改。

请注意，在所有情况下的图的机构不受影响。这是这些运算符的关键所在，它允许新得到图可以复用初始图的结构索引。下面的代码段在逻辑上是等效的，但第一个不保留结构索引，所以不会从 GraphX 系统优化中受益：

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

相反，使用 mapVertices保存索引：

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

这些操作经常被用来初始化图的特定计算或者去除不必要的属性。例如，给定一个将出度作为顶点的属性图(我们之后将介绍如何构建这样的图)，我们初始化它作为 PageRank：

```
// Given a graph where the vertex property is the out-degree
val inputGraph: Graph[Int, String] =
      graph.outerJoinVertices(graph.outDegrees)((vid, _, degOpt) => degOpt.getOrElse(0))
// Construct a graph where each edge contains the weight
// and each vertex is the initial PageRank
```

```
val outputGraph: Graph[Double, Double] =
  inputGraph.mapTriplets(triplet => 1.0 / triplet.srcAttr).mapVertices((id, _)
=> 1.0)
```

1.6 结构操作

当前 GraphX 只支持一组简单的常用结构化操作，我们希望将来增加更多的操作。以下是基本的结构运算符的列表。

```
class Graph[VD, ED] {
  def reverse: Graph[VD, ED]
  def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
               vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
  def groupEdges(merge: (ED, ED) => ED): Graph[VD,ED]
}
```

该 reverse操作符返回一个新图,新图的边的方向都反转了。这是非常实用的，例如，试图计算逆向PageRank。因为反向操作不修改顶点或边属性或改变的边的数目，它的实现不需要数据移动或复制。

该 子图subgraph将顶点和边的预测作为参数，并返回一个图，它只包含满足了顶点条件的顶点图（值为true），以及满足边条件 *并连接顶点* 的边。subgraph子运算符可应用于很多场景，以限制图表的顶点和边是我们感兴趣的，或消除断开的链接。例如，在下面的代码中，我们删除已损坏的链接：

```


// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
                       (5L, ("franklin", "prof")), (2L, ("istoica", "prof")),
                       (4L, ("peter", "student"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
                       Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi"),
                       Edge(4L, 0L, "student"),   Edge(5L, 0L, "colleague")))
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
// Notice that there is a user 0 (for which we have no information) connected
to users
// 4 (peter) and 5 (franklin).
```

```
graph.triplets.map(
    triplet => triplet.srcAttr._1 + " is the " + triplet.attr + " of " +
triplet.dstAttr._1
  ).collect.foreach(println(_))
// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
// The valid subgraph will disconnect users 4 and 5 by removing user 0
validGraph.vertices.collect.foreach(println(_))
validGraph.triplets.map(
    triplet => triplet.srcAttr._1 + " is the " + triplet.attr + " of " +
triplet.dstAttr._1
  ).collect.foreach(println(_))
```

注意，在上面的例子中，仅提供了顶点条件。如果不提供顶点或边的条件，在subgraph操作中默认为 真 。

mask操作返回一个包含输入图中所有的顶点和边的图。这可以用来和subgraph一起使用，以限制基于属性的另一个相关图。例如，我们用去掉顶点的图来运行联通分量，并且限制输出为合法的子图。

```
// Run Connected Components
val ccGraph = graph.connectedComponents() // No longer contains missing field
// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
// Restrict the answer to the valid subgraph
val validCCGraph = ccGraph.mask(validGraph)
```

该 groupEdges操作合并在多重图中的平行边（即重复顶点对之间的边）。在许多数值计算的应用中，平行的边缘可以 *加入* （他们的权重的会被汇总）为单条边从而降低了图形的大小。

1.7 Join 操作

在许多情况下，有必要从外部集合（RDDS）中加入图形数据。例如，我们可能有额外的用户属性，想要与现有的图形合并，或者我们可能需要从一个图选取一些顶点属性到另一个图。这些任务都可以使用来 *join*经操作完成。下面我们列出的关键联接运算符：

```
class Graph[VD, ED] {
  def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD)
    : Graph[VD, ED]
  def outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])(map: (VertexId, VD,
Option[U]) => VD2)
    : Graph[VD2, ED]
}
```

该 joinVertices运算符连接与输入RDD的顶点，并返回一个新的图，新图的顶点属性是通过用户自定义的 map功能作用在被连接的顶点上。没有匹配的RDD保留其原始值。

需要注意的是，如果RDD顶点包含多于一个的值，其中只有一个将会被使用。因此，建议在输入的RDD在初始为唯一的时候，使用下面的 *pre-index*所得到的值以加快后续join。

```
val nonUniqueCosts: RDD[(VertexID, Double)]
val uniqueCosts: VertexRDD[Double] =
  graph.vertices.aggregateUsingIndex(nonUnique, (a,b) => a + b)
val joinedGraph = graph.joinVertices(uniqueCosts)(
  (id, oldCost, extraCost) => oldCost + extraCost)
```

更一般 outerJoinVertices操作类似于 joinVertices，除了将用户定义的 map函数应用到所有的顶点，并且可以改变顶点的属性类型。因为不是所有的顶点可能会在输入匹配值RDD的 mpa 函数接受一个 Optin类型。例如，我们可以通过

用 outDegree 初始化顶点属性来设置一个图的 PageRank。

```
val outDegrees: VertexRDD[Int] = graph.outDegrees
val degreeGraph = graph.outerJoinVertices(outDegrees) { (id, oldAttr, outDegOpt)
=>
  outDegOpt match {
    case Some(outDeg) => outDeg
    case None => 0 // No outDegree means zero outDegree
  }
}
```

您可能已经注意到，在上面的例子中采用了多个参数列表的curried函数模式（例如，f(a)(b)）。虽然我们可以有同样写f(a)(b)为 f(a, b)，这将意味着该类型推断 b不依赖于 a。其结果是，用户将需要提供类型标注给用户自定义的函数：

```
val joinedGraph = graph.joinVertices(uniqueCosts,
  (id: VertexID, oldCost: Double, extraCost: Double) => oldCost + extraCost)
```

1.8 邻居聚集

图形计算的一个关键部分是聚集每个顶点的邻域信息。例如，我们可能想要知道每个用户追随者的数量或每个用户的追随者的平均年龄。许多图迭代算法（如 PageRank，最短路径，连通分量等）反复聚集邻居节点的属性，（例如，当前的 PageRank 值，到源节点的最短路径，最小可达顶点 ID）。

1.8.1 Map Reduce Triplets（mapReduceTriplets）

GraphX中核心（大量优化）聚集操作是 mapReduceTriplets操作：

```
class Graph[VD, ED] {
  def mapReduceTriplets[A](
      map: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
      reduce: (A, A) => A)
    : VertexRDD[A]
}
```

该 mapReduceTriplets运算符将用户定义的map函数作为输入，并且将map作用到每个triplet，并可以得到triplet上所有的顶点（或者两个，或者空）*的信息*。为了便于优化预聚合，我们目前仅支持发往triplet的源或目的地的顶点信息。用户定义的 reduce功能将合并所有目标顶点相同的信息。该 mapReduceTriplets操作返回 VertexRDD [A] ，包含所有以每个顶点作为目标节点集合消息（类型 A）。没有收到消息的顶点不包含在返回 VertexRDD。

需要注意的是 mapReduceTriplets需要一个附加的可选 activeSet（上面没有显示,请参见API文档的详细信息），这限制了 VertexRDD地图提供的邻接边的map阶段：

```
activeSetOpt: Option[(VertexRDD[_], EdgeDirection)] = None
```

该EdgeDirection指定了哪些和顶点相邻的边包含在map阶段。如果该方向是in，则用户定义的 mpa函数 将仅仅作用目标顶点在与活跃集中。如果方向是 out，则该 map函数将仅仅作用在那些源顶点在活跃集中的边。如果方向是 either，则 map 函数将仅在*任*一顶点在活动集中的边。如果方向是 both，则map函数将仅作用在两个顶点都在活跃集中。活跃集合必须来自图的顶点中。限制计算到相邻顶点的一个子集三胞胎是增量迭代计算中非常必要，而且是GraphX 实现Pregel中的关键。

在下面的例子中我们使用 mapReduceTriplets算子来计算高级用户追随者的平均年龄。

```
// Import random graph generation library
import org.apache.spark.graphx.util.GraphGenerators
// Create a graph with "age" as the vertex property.  Here we use a random graph
for simplicity.
val graph: Graph[Double, Int] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapVertices( (id, _) =>
id.toDouble )
// Compute the number of older followers and their total age
val olderFollowers: VertexRDD[(Int, Double)] = graph.mapReduceTriplets[(Int,
Double)](
  triplet => { // Map Function
    if (triplet.srcAttr > triplet.dstAttr) {
```

```
    // Send message to destination vertex containing counter and age
    Iterator((triplet.dstId, (1, triplet.srcAttr)))
   } else {
    // Don't send a message for this triplet
    Iterator.empty
   }
  },
  // Add counter and age
  (a, b) => (a._1 + b._1, a._2 + b._2) // Reduce Function
)
// Divide total age by number of older followers to get average age of older
followers
val avgAgeOfOlderFollowers: VertexRDD[Double] =
  olderFollowers.mapValues( (id, value) => value match { case (count, totalAge)
=> totalAge / count } )
// Display the results
avgAgeOfOlderFollowers.collect.foreach(println(_))
```

注意，当消息（和消息的总和）是固定尺寸的时候（例如，浮点运算和加法而不是列表和连接）时，mapReduceTriplets 操作执行。更精确地说，结果 mapReduceTriplets 最好是每个顶点度的次线性函数。

1.8.2 计算度信息

一个常见的聚合任务是计算每个顶点的度：每个顶点相邻边的数目。在有向图的情况下，往往需要知道入度，出度，以及总度。该 GraphOps类包含一系列的运算来计算每个顶点的度的集合。例如，在下面我们计算最大的入度，出度，总度：

```
// Define a reduce operation to compute the highest degree vertex
def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {
  if (a._2 > b._2) a else b
}
// Compute the max degrees
val maxInDegree: (VertexId, Int)  = graph.inDegrees.reduce(max)
val maxOutDegree: (VertexId, Int) = graph.outDegrees.reduce(max)
val maxDegrees: (VertexId, Int)   = graph.degrees.reduce(max)
```

1.8.3 收集邻居

在某些情况下可能更容易通过收集相邻顶点和它们的属性来表达在每个顶点表示的计算。这可以通过使用容易地实现 collectNeighborIds和 collectNeighbors运算。

```
class GraphOps[VD, ED] {
  def collectNeighborIds(edgeDirection: EdgeDirection):
VertexRDD[Array[VertexId]]
  def collectNeighbors(edgeDirection: EdgeDirection):
VertexRDD[ Array[(VertexId, VD)] ]
}
```

需要注意的是，这些运算计算代价非常高，因为他们包含重复信息，并且需要大量的通信。如果可能的话尽量直接使用 mapReduceTriplets。

1.9 缓存和清空缓存

在Spark中，RDDS默认并不保存在内存中。为了避免重复计算，当他们需要多次使用时，必须明确地使用缓存（见 Spark编程指南）。在GraphX 中Graphs行为方式相同。当需要多次使用图形时，一定要首先调用Graph.cache（ ）以在迭代计算，为了最佳性能，也可能需要 *清空缓存* 。默认情况下，缓存的RDDS和图表将保留在内存中，直到内存压力迫使他们按照LRU顺序被删除。对于迭代计算，之前的迭代的中间结果将填补缓存。虽然他们最终将被删除，内存中的不必要的数据会使垃圾收集机制变慢。一旦它们不再需要缓存，就立即清空中间结果的缓存，这将会更加有效。这涉及物化（缓存和强迫）图形或RDD每次迭代，清空所有其他数据集，并且只使用物化数据集在未来的迭代中。然而，由于图形是由多个RDDS的组成的 ,正确地持续化他们将非常困难。对于迭代计算 ,我们推荐使用Pregel API，它正确地unpersists中间结果。

Pregel 的 API

图本质上是递归的数据结构， 因为顶点的性质取决于它们的邻居，这反过来又依赖于邻居的属性。其结果是许多重要的图形算法迭代重新计算每个顶点的属性，直到定点条件满足为止。一系列图像并行方法已经被提出来表达这些迭代算法。GraphX 提供了类似与 Pregel 的操作，这是 Pregel 和 GraphLab 方法的融合。

从总体来看，Graphx 中的 Pregel 是一个批量同步并行消息传递抽象 约束到该图的拓扑结构。Pregel 运算符在一系列超步骤中 ,其中顶点收到从之前的步骤中流入消息的总和，计算出顶点属性的新值，然后在下一步中将消息发送到相邻的顶点。不同于 Pregel，而是

更像 GraphLab 消息被并行计算，并且作为 edge triplet，该消息的计算可以访问的源和目的地的顶点属性。没有收到消息的顶点在一个超级步跳过。当没有消息是，Pregel 停止迭代，并返回最终图形。

请注意，不像更标准的 Pregel 的实现，在 GraphX 中顶点只能将消息发送到邻近的顶点，并且信息构建是通过使用用户定义的消息函数并行执行。这些限制使得在 GraphX 有额外的优化。

以下是类型签名 Pregel，以及一个初始的实现（注调用 graph.cache 已被删除）中：

```
class GraphOps[VD, ED] {
  def pregel[A]
      (initialMsg: A,
       maxIter: Int = Int.MaxValue,
       activeDir: EdgeDirection = EdgeDirection.Out)
      (vprog: (VertexId, VD, A) => VD,
       sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
       mergeMsg: (A, A) => A)
    : Graph[VD, ED] = {
    // Receive the initial message at each vertex
    var g = mapVertices( (vid, vdata) => vprog(vid, vdata, initialMsg) ).cache()
    // compute the messages
    var messages = g.mapReduceTriplets(sendMsg, mergeMsg)
    var activeMessages = messages.count()
    // Loop until no messages remain or maxIterations is achieved
    var i = 0
    while (activeMessages > 0 && i < maxIterations) {
      //                Receive                the                messages:
-------------------------------------------------------------------------
      // Run the vertex program on all vertices that receive messages
      val newVerts = g.vertices.innerJoin(messages)(vprog).cache()
      // Merge the new vertex values back into the graph
      g    =    g.outerJoinVertices(newVerts)    {    (vid,    old,    newOpt)    =>
newOpt.getOrElse(old) }.cache()
      //                            Send                            Messages:
-------------------------------------------------------------------------------
--
      // Vertices that didn't receive a message above don't appear in newVerts
and therefore don't
      // get to send messages.  More precisely the map phase of mapReduceTriplets
is only invoked
      // on edges in the activeDir of vertices in newVerts
      messages    =    g.mapReduceTriplets(sendMsg,    mergeMsg,    Some((newVerts,
activeDir))).cache()
```

```
      activeMessages = messages.count()
      i += 1
    }
    g
  }
}
```

请注意，Pregel 需要两个参数列表（即 graph.pregel(list1)(list2)）。第一个参数列表中包含的配置参数包括初始信息，迭代的最大次数，以及发送消息（默认出边）的方向。第二个参数列表包含用于用户定义的接收消息（顶点程序 vprog），计算消息（sendMsg），并结合信息 mergeMsg。

我们可以使用 Pregel 运算符来表达计算，如在下面的例子中的单源最短路径。

```
import org.apache.spark.graphx._
// Import random graph generation library
import org.apache.spark.graphx.util.GraphGenerators
// A graph with edge attributes containing distances
val graph: Graph[Int, Double] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapEdges(e =>
e.attr.toDouble)
val sourceId: VertexId = 42 // The ultimate source
// Initialize the graph such that all vertices except the root have distance
infinity.
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), // Vertex Program
  triplet => {  // Send Message
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {
      Iterator.empty
    }
  },
  (a,b) => math.min(a,b) // Merge Message
  )
println(sssp.vertices.collect.mkString("\n"))
```

Graph Builder

GraphX 提供多种从RDD或者硬盘中的节点和边中构建图。默认情况下，没有哪种 Graph Builder会重新划分图的边; 相反，边会留在它们的默认分区（如原来的HDFS

块）。Graph.groupEdges需要的图形进行重新分区，因为它假设相同的边将被放在同一个分区同一位置，所以你必须在调用Graph.partitionBy之前调用groupEdges。

```
object GraphLoader {
  def edgeListFile(
      sc: SparkContext,
      path: String,
      canonicalOrientation: Boolean = false,
      minEdgePartitions: Int = 1)
    : Graph[Int, Int]
}
```

GraphLoader.edgeListFile提供了一种从磁盘上边的列表载入图的方式。它解析了一个以下形式的邻接列表（源顶点ID，目的地顶点ID）对，忽略以#开头的注释行：

```
# This is a comment
2 1
4 1
1 2
```

它从指定的边创建了一个图表，自动边中提到的任何顶点。所有顶点和边的属性默认为1。canonicalOrientation参数允许重新定向边的正方向（srcId <dstId），这是必需的 connected component算法。该minEdgePartitions参数指定边缘分区生成的最小数目;例如，在HDFS文件具有多个块，那么就有多个边的分割。

```
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]

}
```

Graph.apply允许从顶点和边的RDDS中创建的图。重复的顶点会任意选择，并在边RDD中存在的顶点， 但不是顶点RDD会被赋值为默认属性。

Graph.fromEdges允许从只有边的元组RDD创建的图，自动生成由边中存在的顶点，并且给这些顶点赋值为缺省值。

Graph.fromEdgeTuples允许从只有边的元组的RDD图中创建图，并将的边的值赋为1，并自动创建边中所存在的顶点，并设置为缺省值。它也支持删除重边; 进行删除重边时，传入 PartitionStrategy的Some 作为uniqueEdges参数（例如，uniqueEdges = Some（PartitionStrategy.RandomVertexCut））。分区策略是必要的，因为定位在同一分区相同的边，才能使他们能够进行重复删除。

顶点和边 RDDs

GraphX 公开了图中 RDD 顶点和边的视图。然而，因为 GraphX 将顶点和边保存在优化的数据结构，并且为这些数据结构提供额外的功能，顶点和边分别作为 VertexRDD 和 EdgeRDD 返回。在本节中，我们回顾一些这些类型的其他有用的功能。

1.11 VertexRDDs

该 VertexRDD [A]继承 RDD [(VertexID, A)]，并增加了一些额外的限制 ，每个 VertexID 只出现一次。此外，VertexRDD[A]表示一个顶点集合，其中每个顶点与类型的属性为 A。在内部，这是通过将顶点属性中存储在一个可重复使用的哈希表。因此，如果两个 VertexRDD s 继承自相同的基类 VertexRDD（例如，通过 filter 或 mapValues ），他们可以参加在常数时间内实现合并，而不需要重新计算 hash 值。要充分利用这个索引数据结构，VertexRDD 提供了以下附加功能：

```
class VertexRDD[VD] extends RDD[(VertexID, VD)] {
  // Filter the vertex set but preserves the internal index
  def filter(pred: Tuple2[VertexId, VD] => Boolean): VertexRDD[VD]
  // Transform the values without changing the ids (preserves the internal index)
  def mapValues[VD2](map: VD => VD2): VertexRDD[VD2]
  def mapValues[VD2](map: (VertexId, VD) => VD2): VertexRDD[VD2]
  // Remove vertices from this set that appear in the other set
  def diff(other: VertexRDD[VD]): VertexRDD[VD]
  // Join operators that take advantage of the internal indexing to accelerate
joins (substantially)
  def leftJoin[VD2, VD3](other: RDD[(VertexId, VD2)])(f: (VertexId, VD,
Option[VD2]) => VD3): VertexRDD[VD3]
  def innerJoin[U, VD2](other: RDD[(VertexId, U)])(f: (VertexId, VD, U) => VD2):
VertexRDD[VD2]
  // Use the index on this RDD to accelerate a `reduceByKey` operation on the
input RDD.
```

```
  def aggregateUsingIndex[VD2](other: RDD[(VertexId, VD2)], reduceFunc: (VD2,
VD2) => VD2): VertexRDD[VD2]
}
```

请注意 ,例如 如何 filter 操作符返回一个 VertexRDD。过滤器使用的是实际通过 BitSet 实现的，从而复用索引和保持能快速与其他 VertexRDD 实现连接功能。 类似地，mapValues 操作不允许 mapha 函数改变 VertexID ,从而可以复用统一 HashMap 中的数据结构。当两个 VertexRDD 派生自同一 HashMap，并且是通过线性少买而非代价昂贵的逐点查询时，无论是 leftJoin 和 innerJoin 连接时能够识别 VertexRDD 。

该aggregateUsingIndex操作是一种新的有效的从 RDD[(VertexID, A)]构建新的VertexRDD的方式。从概念上讲，如果我在一组顶点上构建了一个VertexRDD [B]， *这是一个* 在某些顶点RDD[(VertexID, A)]的超集，然后我可以重用该索引既聚集，随后为RDD[(VertexID, A)]建立索引。例如：

```
val setA: VertexRDD[Int] = VertexRDD(sc.parallelize(0L until 100L).map(id => (id,
1)))
val rddB: RDD[(VertexId, Double)] = sc.parallelize(0L until 100L).flatMap(id =>
List((id, 1.0), (id, 2.0)))
// There should be 200 entries in rddB
rddB.count
val setB: VertexRDD[Double] = setA.aggregateUsingIndex(rddB, _ + _)
// There should be 100 entries in setB
setB.count
// Joining A and B should now be fast!
val setC: VertexRDD[Double] = setA.innerJoin(setB)((id, a, b) => a + b)
```

EdgeRDDs

该EdgeRDD [ED ,VD] ,它继承RDD [Edge[ED],以各种分区策略 PartitionStrategy将边划分成不同的块。在每个分区中，边属性和邻接结构，分别存储，这使得更改属性值时，能够最大限度的复用。

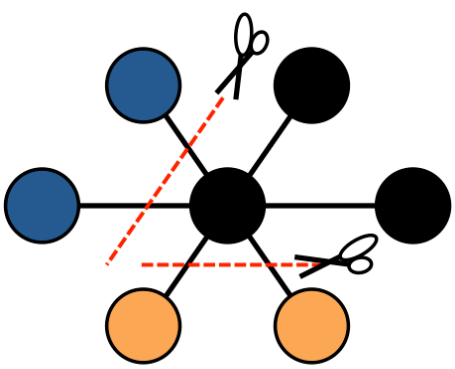
EdgeRDD 是提供的三个额外的函数：

```
// Transform the edge attributes while preserving the structure
def mapValues[ED2](f: Edge[ED] => ED2): EdgeRDD[ED2, VD]
// Revere the edges reusing both attributes and structure
def reverse: EdgeRDD[ED, VD]
// Join two `EdgeRDD`s partitioned using the same partitioning strategy.
def innerJoin[ED2, ED3](other: EdgeRDD[ED2, VD])(f: (VertexId, VertexId, ED, ED2)
=> ED3): EdgeRDD[ED3, VD]
```

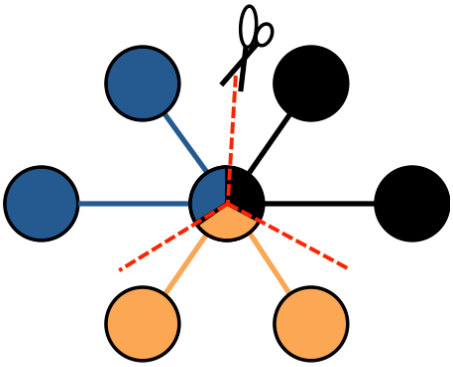
在大多数应用中，我们发现，在 EdgeRDD 中的操作是通过图形运算符来实现，或依靠在基类定义的 RDD 类操作。

优化图的表示

关于 GraphX 中如何表示分布式图结构的详细描述，这个话题超出了本指南的范围，一些高层次的理解可能有助于设计可扩展的算法设计以及 API 的最佳利用。GraphX 采用顶点切的方法来分发图划分：

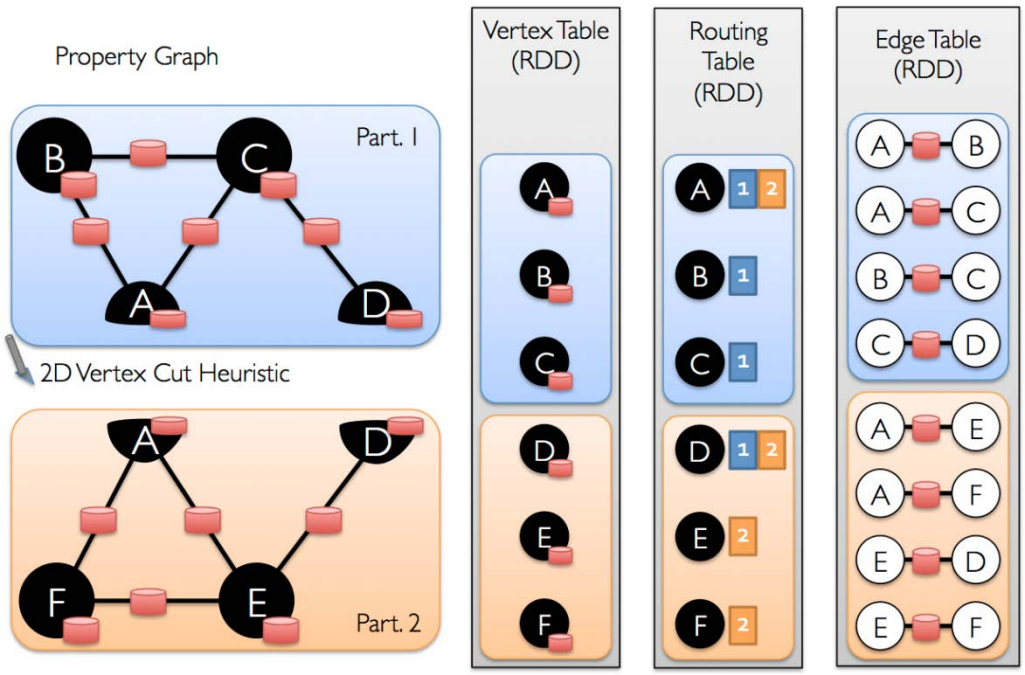


Edge Cut



Vertex Cut

不通过边划分图，GraphX 沿顶点来划分图，这样可以减少顶点之间的通信和存储开销。逻辑上，这对应于将边分配到不同的机器，并允许顶点跨越多个机器。分配边的确切方法取决于 PartitionStrategy 并有多个权衡各种试探法。用户可以通过重新分区图与不同的策略之间进行选择 Graph.partitionBy 操作。默认分区策略是按照图的构造，使用图中初始的边。但是，用户可以方便地切换到二维-分区或 GraphX 中其他启发式分区方法。



一旦边被划分，并行图计算的关键挑战在于有效的将每个顶点属性和边的属性连接起来。由于在现实世界中，边的数量多于顶点的数量，我们把顶点属性放在边中。因为不是所有的

分区将包含所有顶点相邻的边的信息，我们在内部维护一个路由表，这个表确定在哪里广播顶点信息，执行 triplet 和 mapReduceTriplets 的连接操作。

图算法

GraphX 包括一组图形算法来简化分析任务。该算法被包含于org.apache.spark.graphx.lib包中，并可直接通过 GraphOps而被Graph中的方法调用。本节介绍这些算法以及如何使用它们。

1.11 PageRank

PageRank记录了图中每个顶点的重要性，假设一条边从 *u*到 *v*，代表从u传递给 v的重要性。例如，如果一个Twitter用户有很多粉丝，用户排名将很高。

GraphX 自带的PageRank的静态和动态的实现，放在PageRank对象中。静态的PageRank运行的固定数量的迭代，而动态的PageRank运行，直到排名收敛（即当每个迭代和上一迭代的差值，在某个范围之内时停止迭代）。GraphOps允许Graph中的方法直接调用这些算法。

GraphX 还包括，我们可以将 PageRank 运行在社交网络数据集中。一组用户给出graphx/data/users.txt，以及一组用户之间的关系，给出了 graphx/data/followers.txt。我们可以按照如下方法来计算每个用户的网页级别：

```
// Load the edges as a graph
val graph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt")
// Run PageRank
val ranks = graph.pageRank(0.0001).vertices
// Join the ranks with the usernames
val users = sc.textFile("graphx/data/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ranksByUsername = users.join(ranks).map {
  case (id, (username, rank)) => (username, rank)
}
// Print the result
println(ranksByUsername.collect().mkString("\n"))
```

1.12 联通分量

连接分量算法标出了图中编号最低的顶点所联通的子集。例如，在社交网络中，连接分量类似集群。GraphX 包含在ConnectedComponents对象的算法，并且我们从该社交网络数据集中计算出连接组件的PageRank部分，如下所示：

```
// Load the graph as in the PageRank example
val graph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt")
// Find the connected components
val cc = graph.connectedComponents().vertices
// Join the connected components with the usernames
val users = sc.textFile("graphx/data/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ccByUsername = users.join(cc).map {
  case (id, (username, cc)) => (username, cc)
}
// Print the result
println(ccByUsername.collect().mkString("\n"))
```

1.13 三角计数

当顶点周围与有一个其他两个顶点有连线时，这个顶点是三角形的一部分。GraphX在TriangleCount对象实现了一个三角形计数算法，这个算法计算通过各顶点的三角形数目，从而提供集群的度。我们从PageRank部分计算社交网络数据集的三角形数量。注意TriangleCount要求边是规范的指向（srcId <dstId），并使用 *Graph.partitionBy* 来分割图形。

```
// Load the edges in canonical order and partition the graph for triangle count
val graph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt",
true).partitionBy(PartitionStrategy.RandomVertexCut)
// Find the triangle count for each vertex
val triCounts = graph.triangleCount().vertices
// Join the triangle counts with the usernames
val users = sc.textFile("graphx/data/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val triCountByUsername = users.join(triCounts).map { case (id, (username, tc))
=>
  (username, tc)
}
// Print the result
```

```
println(triCountByUsername.collect().mkString("\n"))
```

示例

假设我想从一些文本文件中构建图，只考虑图中重要关系和用户，在子图中运行的页面排名算法，然后终于返回与顶级用户相关的属性。我们可以在短短的几行 GraphX 代码中实现这一功能：

```
// Connect to the Spark cluster
val sc = new SparkContext("spark://master.amplab.org", "research")

// Load my user data and parse into tuples of user id and attribute list
val users = (sc.textFile("graphx/data/users.txt")
  .map(line => line.split(",")).map( parts => (parts.head.toLong, parts.tail) ))

// Parse the edge data which is already in userId -> userId format
val followerGraph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt")

// Attach the user attributes
val graph = followerGraph.outerJoinVertices(users) {
  case (uid, deg, Some(attrList)) => attrList
  // Some users may not have attributes so we set them as empty
  case (uid, deg, None) => Array.empty[String]
}

// Restrict the graph to users with usernames and names
val subgraph = graph.subgraph(vpred = (vid, attr) => attr.size == 2)

// Compute the PageRank
val pagerankGraph = subgraph.pageRank(0.001)

// Get the attributes of the top pagerank users
val userInfoWithPageRank = subgraph.outerJoinVertices(pagerankGraph.vertices)
{
  case (uid, attrList, Some(pr)) => (pr, attrList.toList)
  case (uid, attrList, None) => (0.0, attrList.toList)
}

println(userInfoWithPageRank.vertices.top(5)(Ordering.by(_._2._1)).mkString(
"\n"))
```

Spark 亚太研究院

Spark 亚太研究院是中国最专业的一站式大数据 Spark 解决方案供应商和高品质大数据企业级完整培训与服务供应商，以帮助企业规划、架构、部署、开发、培训和使用 Spark 为核心，同时提供 Spark 源码研究和应用技术训练。针对具体 Spark 项目，提供完整而彻底的解决方案。包括 Spark 一站式项目解决方案、Spark 一站式项目实施方案及 Spark 一体化顾问服务。

官网： www.sparkinchina.com

近期活动

- 2014 年亚太地区规格最高的 Spark 技术盛会！
- 面向大数据、云计算开发者、技术爱好者的饕餮盛宴！
- 云集国内外 Spark 技术领军人物及灵魂人物！
- 技术交流、应用分享、源码研究、商业案例探询！

时间：2014 年 12 月 6-7 日
地点：北京珠三角万豪酒店
Spark亚太峰会网址：http://www.sparkinchina.com/meeting/2014yt/default.asp

- 如果你是对 Spark 有浓厚兴趣的初学者，在这里你会有绝佳的入门和实践机会！
- 如果你是 Spark 的应用高手，在这里以“武”会友，和技术大牛们尽情切磋！
- 如果你是对 Spark 有深入独特见解的专家，在这里可以尽情展现你的才华！

比赛时间：

2014 年 9 月 30 日—12 月 3 日

Spark开发者大赛网址：http://www.sparkinchina.com/meeting/2014yt/dhhd.asp

视频课程：

《大数据 Spark 实战高手之路》 国内第一个 Spark 视频系列课程

从零起步，分阶段无任何障碍逐步掌握大数据统一计算平台 Spark，从 Spark 框架编写和开发语言 Scala 开始，到 Spark 企业级开发，再到 Spark 框架源码解析、Spark 与 Hadoop 的融合、商业案例和企业面试，一次性彻底掌握 Spark，成为云计算大数据时代的幸运儿和弄潮儿，笑傲大数据职场和人生！

- 第一阶段：熟练的掌握 Scala 语言
  课程学习地址：http://edu.51cto.com/pack/view/id-124.html
- 第二阶段：精通 Spark 平台本身提供给开发者 API
  课程学习地址：http://edu.51cto.com/pack/view/id-146.html
- 第三阶段：精通 Spark 内核
  课程学习地址：http://edu.51cto.com/pack/view/id-148.html
- 第四阶段：掌握基于 Spark 上的核心框架的使用
  课程学习地址：http://edu.51cto.com/pack/view/id-149.html
- 第五阶段：商业级别大数据中心黄金组合：Hadoop+ Spark
  课程学习地址：http://edu.51cto.com/pack/view/id-150.html
- 第六阶段：Spark 源码完整解析和系统定制
  课程学习地址：http://edu.51cto.com/pack/view/id-151.html

近期公开课：

《决胜大数据时代：Hadoop、Yarn、Spark 企业级最佳实践》

集大数据领域最核心三大技术：Hadoop 方向 50%：掌握生产环境下、源码级别下的 Hadoop 经验，解决性能、集群难点问题；Yarn 方向 20%：掌握最佳的分布式集群资源管理框架，能够轻松使用 Yarn 管理 Hadoop、Spark 等；Spark 方向 30%：未来统一的大数据框架平台，剖析 Spark 架构、内核等核心技术，对未来转向 SPARK 技术，做好技术储备。课程内容落地性强，即解决当下问题，又有助于驾驭未来。

开课时间： 2014 年 10 月 26-28 日北京、2014 年 11 月 1-3 日深圳

咨询电话：4006-998-758

QQ 交流群： 1 群：317540673 （已满）
2 群：297931500

微信公众号：spark-china