SparkGraphX 应用解析

教案

文档修订记录

文件状态： [√]草稿 [ ]正式发布	当前版本：	V1.1
	作　　者：	武玉飞
	审 核 人：	
	发布日期：	20170518

编制日期	版本	状态	简要说明	作者	审核者	审核日期
20170518	V1.1	A				

说明：

1. 按修改时间先后倒序排列，最新修改的排在第一行。
2. 版本栏中填入版本编号或者更改记录编号。
3. 状态分为三种状态：A——增加；M——修改；D——删除。
4. 在简要说明栏中填写变更的内容和变更的范围。
5. 表中所有日期格式为：YYYYMMDD。

目　录

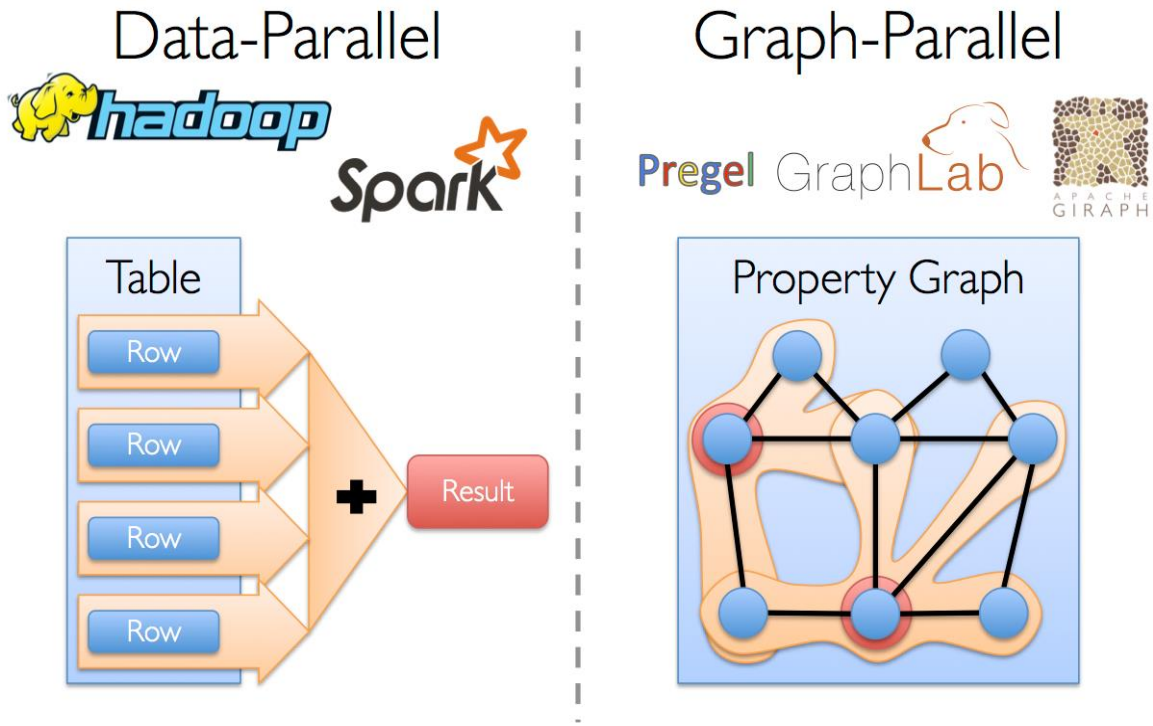
第1章 Spark GraphX 概述

1.1 什么是 Spark GraphX

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。那么什么是图，都计算些什么？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，数据中出现网状结构关系都需要图计算。

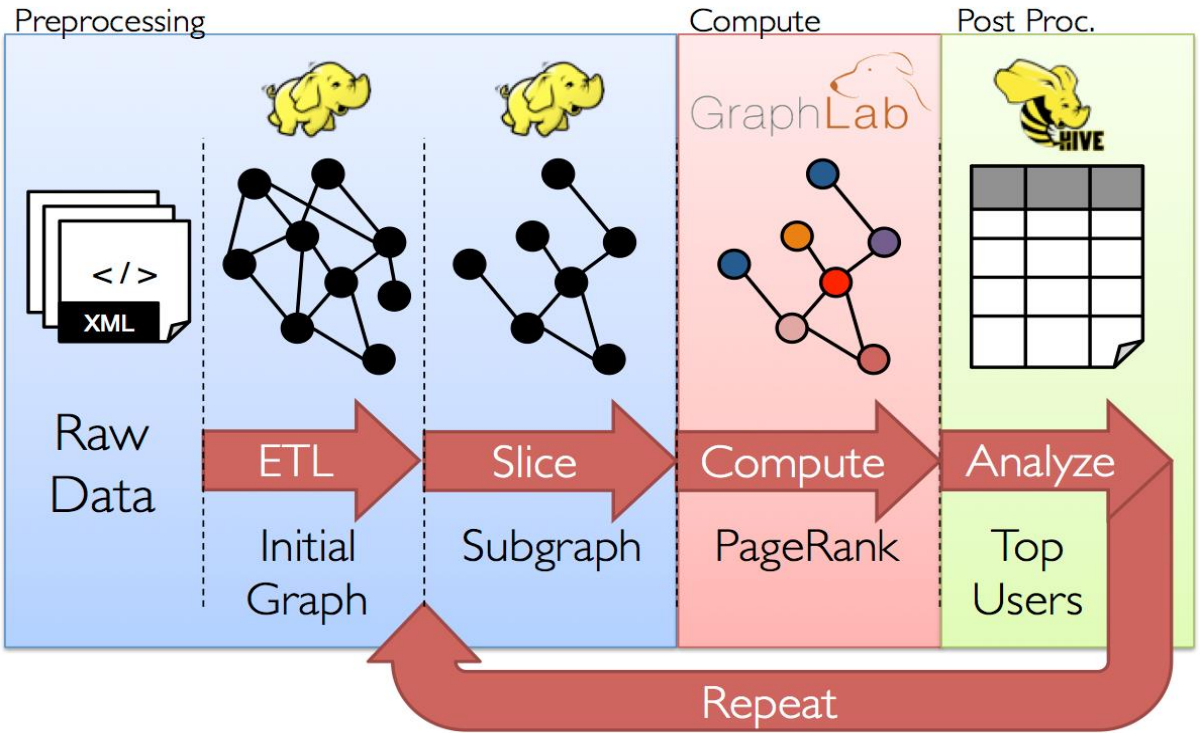
GraphX 是一个新的 Spark API，它用于图和分布式图(graph-parallel)的计算。GraphX 通过引入弹性分布式属性图（Resilient Distributed Property Graph）：顶点和边均有属性的有向多重图，来扩展 Spark RDD。为了支持图计算，GraphX 开发了一组基本的功能操作以及一个优化过的 Pregel API。另外，GraphX 也包含了一个快速增长的图算法和图 builders 的集合，用以简化图分析任务。

从社交网络到语言建模，不断增长的数据规模以及图形数据的重要性已经推动了许多新的分布式图系统的发展。通过限制计算类型以及引入新的技术来切分和分配图，这些系统可以高效地执行复杂的图形算法，比一般的分布式数据计算（data-parallel，如 spark、MapReduce）快很多。

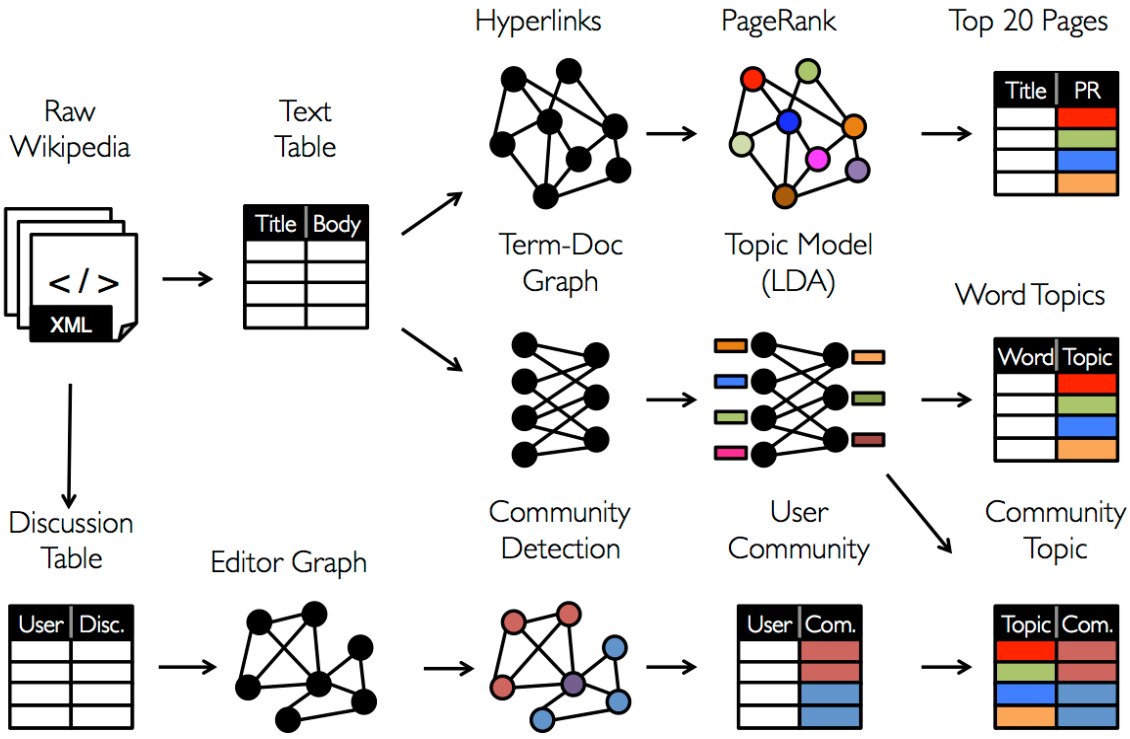


分布式图（graph-parallel）计算和分布式数据（data-parallel）计算类似，分布式数据计算采用了一种 record-centric（以记录为中心）的集合视图，而分布式图计算采用了一种 vertex-centric（以顶点为中心）的图视图。 分布式数据计算通过同时处理独立的数据来获得并发的目的，分布式图计算则是通过对图数据进行分区（即切分）来获得并发的目的。更准确的说，分布式图计算递归地定义特征的转换函数（这种转换函数作用于邻居特征），通过并发地执行这些转换函数来获得并发的目的。

分布式图计算比分布式数据计算更适合图的处理，但是在典型的图处理流水线中，它并不能很好地处理所有操作。例如，虽然分布式图系统可以很好的计算 PageRank 等算法，但是它们不适合从不同的数据源构建图或者跨过多个图计算特征。 更准确的说，分布式图系统提供的更窄的计算视图无法处理那些构建和转换图结构以及跨越多个图的需求。分布式图系统中无法提供的这些操作需要数据在图本体之上移动并且需要一个图层面而不是单独的顶点或边层面的计算视图。例如，我们可能想限制我们的分析到几个子图上，然后比较结果。 这不仅需要改变图结构，还需要跨多个图计算。



我们如何处理数据取决于我们的目标，有时同一原始数据可能会处理成许多不同表和图的视图，并且图和表之间经常需要能够相互移动。如下图所示：



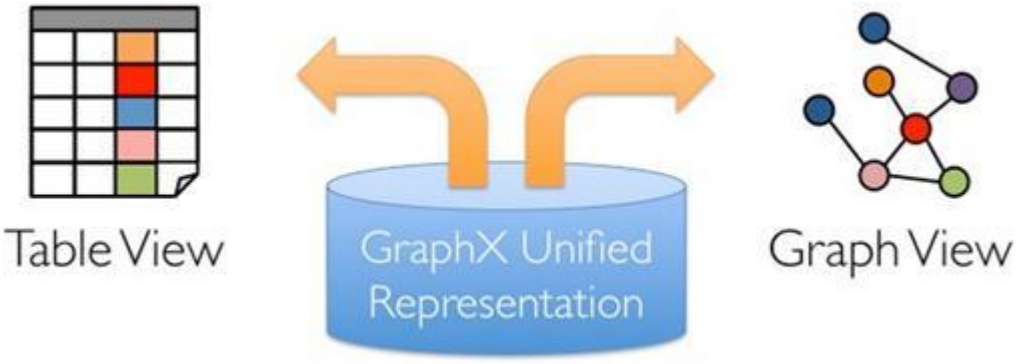
所以我们的图流水线必须通过组合 graph-parallel 和 data- parallel 来实现。但是这种组合必然会导致大量的数据移动以及数据复制，同时这样的系统也非常复杂。 例如，在传统的图计算流水线中，在 Table View 视图下，可能需要 Spark 或者 Hadoop 的支持，在 Graph View 这种视图下，可能需要 Prege 或者 GraphLab 的支持。也就是把图和表分在不同的系统中分别处理。 不同系统之间数据的移动和通信会成为很大的负担。

GraphX 项目将 graph-parallel 和 data-parallel 统一到一个系统中，并提供了一个唯一的组合 API。GraphX 允许用户把数据当做一个图和一个集合（RDD），而不需要数据移动或者复制。也就是说 GraphX 统一了 Graph View 和 Table View， 可以非常轻松的做 pipeline 操作。

1.2 弹性分布式属性图

GraphX 的核心抽象是弹性分布式属性图，它是一个有向多重图，带有连接到每个顶点和边的用户定义的对象。 有向多重图中多个并行的边共享相同

的源和目的顶点。支持并行边的能力简化了建模场景，相同的顶点可能存在多种关系(例如 co-worker 和 friend)。 每个顶点用一个唯一的 64 位长的标识符（VertexID）作为 key。GraphX 并没有对顶点标识强加任何排序。同样，边拥有相应的源和目的顶点标识符。



属性图扩展了 Spark RDD 的抽象，有 Table 和 Graph 两种视图，但是只需要一份物理存储。两种视图都有自己独有的操作符，从而使我们同时获得了操作的灵活性和执行的高效率。属性图以 vertex(VD)和 edge(ED)类型作为参数类型，这些类型分别是顶点和边相关联的对象的类型。

在某些情况下，在同样的图中，我们可能希望拥有不同属性类型的顶点。这可以通过继承完成。例如，将用户和产品建模成一个二分图，我们可以用如下方式：

```
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 RDD 一样，属性图是不可变的、分布式的、容错的。图的值或者结构的改变需要生成一个新的图来实现。注意，原始图中不受影响的部分都可以在新图中重用，用来减少存储的成本。 执行者使用一系列顶点分区方法来对图进行分区。如 RDD 一样，图的每个分区可以在发生故障的情况下被重新创建在不同的机器上。

逻辑上,属性图对应于一对类型化的集合(RDD),这个集合包含每一个顶点和边的属性。因此，图的类中包含访问图中顶点和边的成员变量。

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

VertexRDD[VD]和 EdgeRDD[ED]类是 RDD[(VertexID, VD)]和 RDD[Edge[ED]]的继承和优化版本。VertexRDD[VD]和 EdgeRDD[ED]都提供了额外的图计算功能并提供内部优化功能。

```
abstract class VertexRDD[VD](sc: SparkContext, deps: Seq[Dependency[_]])
extends RDD[(VertexId, VD)](sc, deps)

abstract class EdgeRDD[ED](sc: SparkContext, deps: Seq[Dependency[_]]) extends
RDD[Edge[ED]](sc, deps)
```

GraphX 的底层设计有以下几个关键点。

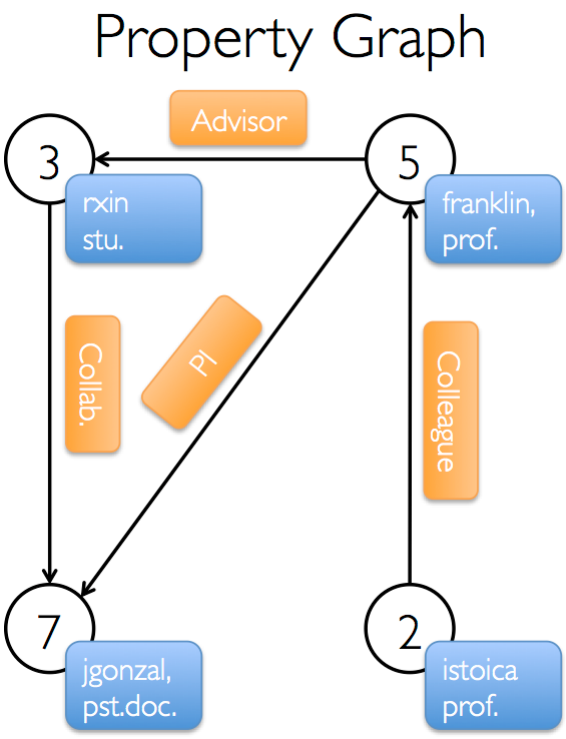
对 Graph 视图的所有操作，最终都会转换成其关联的 Table 视图的 RDD 操作来完成。这样对一个图的计算，最终在逻辑上，等价于一系列 RDD 的转换过程。因此，Graph 最终具备了 RDD 的 3 个关键特性：Immutable、Distributed 和 Fault-Tolerant，其中最关键的是 Immutable（不变性）。逻辑上，所有图的转换和操作都产生了一个新图；物理上，GraphX 会有一定程度的不变顶点和边的复用优化，对用户透明。

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。

1.3 运行图计算程序

假设我们想构造一个包括不同合作者的属性图。顶点属性可能包含用户名和职业。我们可以用描述合作者之间关系的字符串标注边缘。



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

开始的第一步是引入 Spark 和 GraphX 到你的项目中，如下面所示

```
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
```

如果你没有用到 Spark shell，你还将需要 SparkContext。

所得的图形将具有类型签名

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

有很多方式从一个原始文件、RDD 构造一个属性图。最一般的方法是利用 Graph object。下面的代码从 RDD 集合生成属性图。

```
// 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，我们用到了 Edge 样本类。边有一个 srcId 和 dstId 分别对应于源和目标顶点的标示符。另外，Edge 类有一个 attr 成员用来存储边属性。

我们可以分别用 graph.vertices 和 graph.edges 成员将一个图解构为相应的顶点和边。

```
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
```

注意，graph.vertices 返回一个 VertexRDD[(String, String)]，它继承于 RDD[(VertexID, (String, String))]。所以我们可以用 scala 的 case 表达式解构这个元组。另一方面，

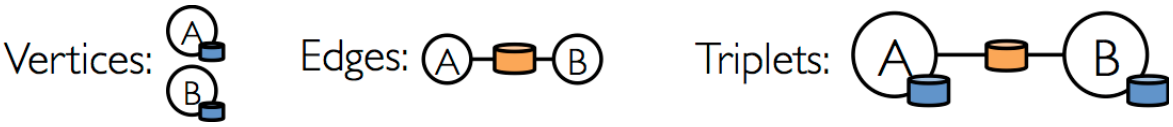
graph.edges 返回一个包含 Edge[String]对象的 EdgeRDD。我们也可以用到 case 类的类型构造器，如下例所示。

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

除了属性图的顶点和边视图，GraphX 也包含了一个三元组视图，三元视图逻辑上将顶点和边的属性保存为一个 RDD[EdgeTriplet[VD, ED]]，它包含 EdgeTriplet 类的实例。可以通过下面的 Sql 表达式表示这个连接。

```
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
```

或者通过下面的图来表示。



EdgeTriplet 类继承于 Edge 类，并且加入了 srcAttr 和 dstAttr 成员，这两个成员分别包含源和目的的属性。我们可以用一个三元组视图渲染字符串集合用来描述用户之间的关系。

```
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(_))
```

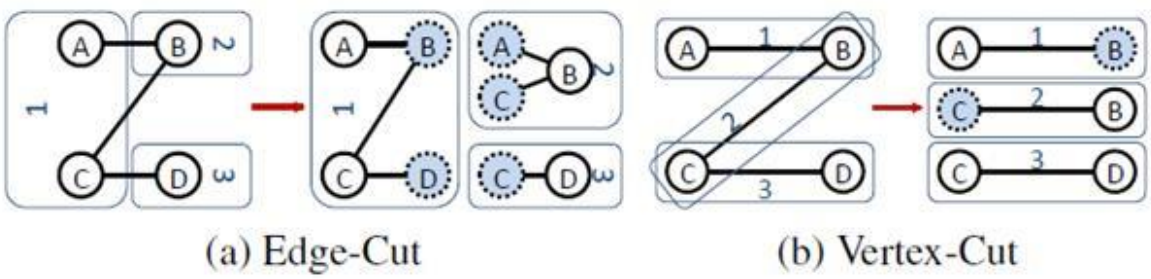
第2章 Spark GraphX 解析

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。

1) 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。
2) 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

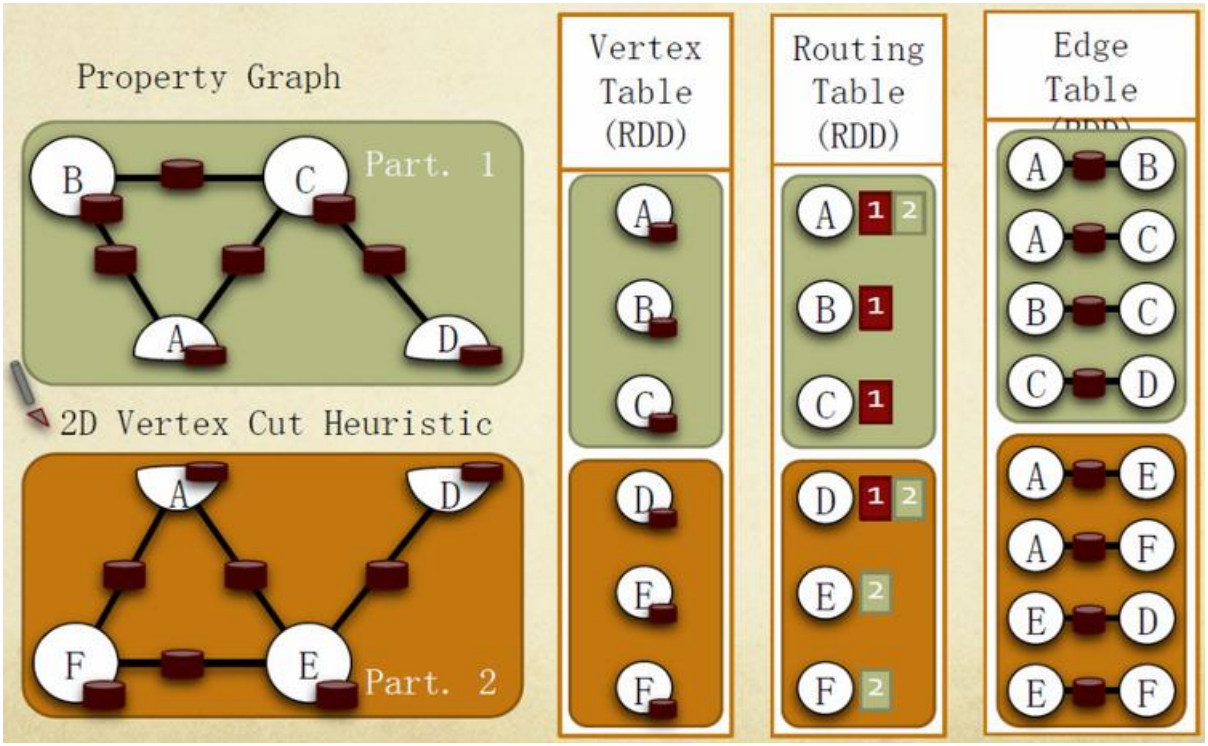
Graphx 借鉴 PowerGraph，使用的是 Vertex-Cut(点分割) 方式存储图，用三个 RDD 存储图数据信息：

VertexTable(id, data)：id 为顶点 id， data 为顶点属性

EdgeTable(pid, src, dst, data)：pid 为分区 id，src 为源顶点 id，dst 为目的顶点 id，data 为边属性

RoutingTable(id, pid)：id 为顶点 id，pid 为分区 id

点分割存储实现如下图所示：



GraphX 在进行图分割时，有几种不同的分区(partition)策略，它通过

PartitionStrategy 专门定义这些策略。在 PartitionStrategy 中，总共定义了 EdgePartition2D、EdgePartition1D、RandomVertexCut 以及 CanonicalRandomVertexCut 这四种不同的分区策略。下面分别介绍这几种策略。

2.1.2.1 RandomVertexCut

```
case object RandomVertexCut extends PartitionStrategy {
  override def getPartition(src: VertexId, dst: VertexId, numParts:
PartitionID): PartitionID = {
    math.abs((src, dst).hashCode()) % numParts
  }
}
```

这个方法比较简单，通过取源顶点和目标顶点 id 的哈希值来将边分配到不同的分区。这个方法会产生一个随机的边分割，两个顶点之间相同方向的边会分配到同一个分区。

2.1.2.2 CanonicalRandomVertexCut

```
case object CanonicalRandomVertexCut extends PartitionStrategy {
  override def getPartition(src: VertexId, dst: VertexId, numParts:
PartitionID): PartitionID = {
    if (src < dst) {
      math.abs((src, dst).hashCode()) % numParts
    } else {
      math.abs((dst, src).hashCode()) % numParts
    }
  }
}
```

这种分割方法和前一种方法没有本质的不同。不同的是，哈希值的产生带有确定的方向（即两个顶点中较小 id 的顶点在前）。两个顶点之间所有的边都会分配到同一个分区，而不管方向如何。

2.1.2.3 EdgePartition1D

```
case object EdgePartition1D extends PartitionStrategy {
  override def getPartition(src: VertexId, dst: VertexId, numParts:
PartitionID): PartitionID = {
    val mixingPrime: VertexId = 1125899906842597L
    (math.abs(src * mixingPrime) % numParts).toInt
  }
}
```

这种方法仅仅根据源顶点 id 来将边分配到不同的分区。有相同源顶点的边会分配到同一分区。

2.1.2.4 EdgePartition2D

```
case object EdgePartition2D extends PartitionStrategy {
  override def getPartition(src: VertexId, dst: VertexId, numParts:
PartitionID): PartitionID = {
    val ceilSqrtNumParts: PartitionID = math.ceil(math.sqrt(numParts)).toInt
    val mixingPrime: VertexId = 1125899906842597L
    if (numParts == ceilSqrtNumParts * ceilSqrtNumParts) {
      // Use old method for perfect squared to ensure we get same results
      val col: PartitionID = (math.abs(src * mixingPrime) %
ceilSqrtNumParts).toInt
      val row: PartitionID = (math.abs(dst * mixingPrime) %
ceilSqrtNumParts).toInt
      (col * ceilSqrtNumParts + row) % numParts
    } else {
      // Otherwise use new method
      val cols = ceilSqrtNumParts
      val rows = (numParts + cols - 1) / cols
      val lastColRows = numParts - rows * (cols - 1)
      val col = (math.abs(src * mixingPrime) % numParts / rows).toInt
      val row = (math.abs(dst * mixingPrime) % (if (col < cols - 1) rows else
lastColRows)).toInt
      col * rows + row
    }
  }
}
```

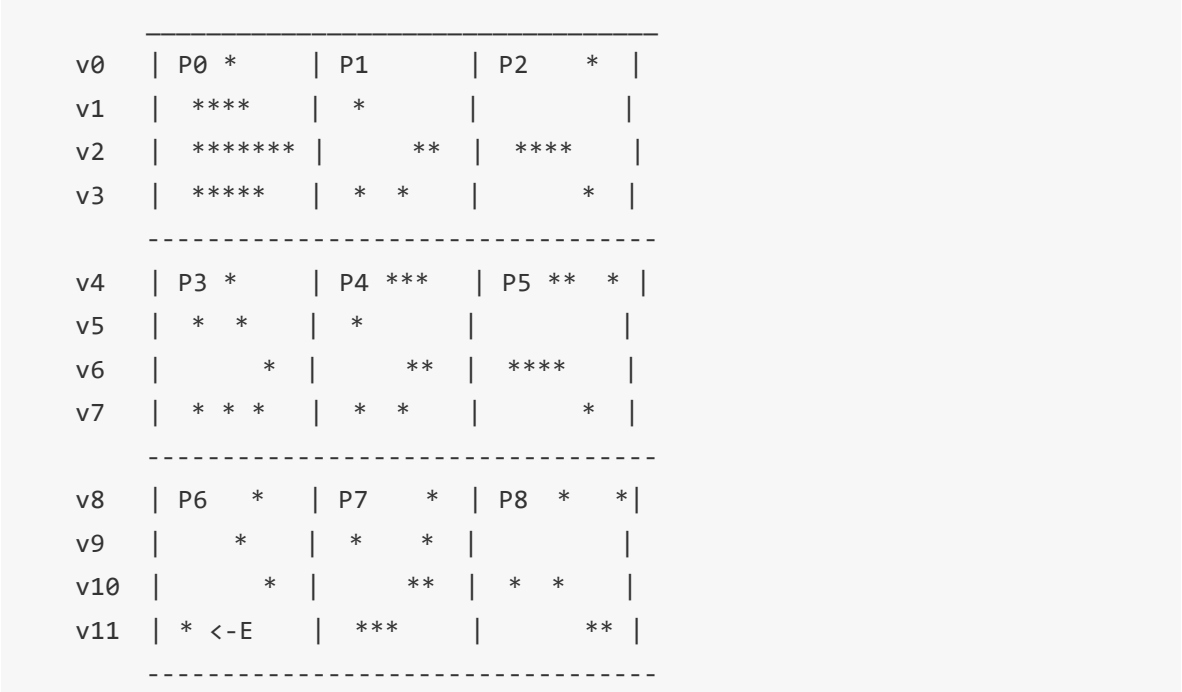
这种分割方法同时使用到了源顶点 id 和目的顶点 id。它使用稀疏边连接矩阵的 2 维区分来将边分配到不同的分区，从而保证顶点的备份数不大于 2 * sqrt(numParts)的限制。这里 numParts 表示分区数。 这个方法的实现分两种情况，即分区数能完全开方和不能完全开方两种情况。当分区数能完全开方时，采用下面的方法：

```
val col: PartitionID = (math.abs(src * mixingPrime) % ceilSqrtNumParts).toInt
val row: PartitionID = (math.abs(dst * mixingPrime) % ceilSqrtNumParts).toInt
(col * ceilSqrtNumParts + row) % numParts
```

当分区数不能完全开方时，采用下面的方法。这个方法的最后一列允许拥有不同的行数。

```
val cols = ceilSqrtNumParts
val rows = (numParts + cols - 1) / cols
//最后一列允许不同的行数
val lastColRows = numParts - rows * (cols - 1)
val col = (math.abs(src * mixingPrime) % numParts / rows).toInt
val row = (math.abs(dst * mixingPrime) % (if (col < cols - 1) rows else
lastColRows)).toInt
col * rows + row
```

下面举个例子来说明该方法。假设我们有一个拥有 12 个顶点的图，要把它切分到 9 台机器。我们可以用下面的稀疏矩阵来表示:



上面的例子中*表示分配到处理器上的边。E 表示连接顶点 v11 和 v1 的边，它被分配到了处理器 P6 上。为了获得边所在的处理器，我们将矩阵切分为 sqrt(numParts) * sqrt(numParts)块。 注意，上图中与顶点 v11 相连接的边只出现在第一列的块(P0,P3,P6)或者最后一行的块(P6,P7,P8)中，这保证了 V11 的副本数不会超过 2 * sqrt(numParts)份，在上例中即副本不能超过 6 份。

在上面的例子中，P0 里面存在很多边，这会造成工作的不均衡。为了提高均衡，我们首先用顶点 id 乘以一个大的素数，然后再 shuffle 顶点的位置。乘以一个大的素数本质上不能解决不平衡的问题，只是减少了不平衡的情况发生。

2.2 vertices、edges 以及 triplets

vertices、edges 以及 triplets 是 GraphX 中三个非常重要的概念。我们在前文 GraphX 介绍中对这三个概念有初步的了解。

2.2.1 vertices

在 GraphX 中，vertices 对应着名称为 VertexRDD 的 RDD。这个 RDD 有顶点 id 和顶点属性两个成员变量。它的源码如下所示：

```
abstract class VertexRDD[VD](sc: SparkContext, deps: Seq[Dependency[_]])
extends RDD[(VertexId, VD)](sc, deps)
```

从源码中我们可以看到，VertexRDD 继承自 RDD[(VertexId, VD)]，这里 VertexId 表示顶点 id，VD 表示顶点所带的属性的类别。这从另一个角度也说明 VertexRDD 拥有顶点 id 和顶点属性。

2.2.2 edges

在 GraphX 中，edges 对应着 EdgeRDD。这个 RDD 拥有三个成员变量，分别是源顶点 id、目标顶点 id 以及边属性。它的源码如下所示：

```
abstract class EdgeRDD[ED](sc: SparkContext, deps: Seq[Dependency[_]]) extends
RDD[Edge[ED]](sc, deps)
```

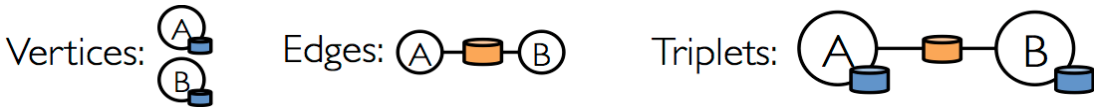
从源码中我们可以看到，EdgeRDD 继承自 RDD[Edge[ED]]，即类型为 Edge[ED]的 RDD。

2.2.3 triplets

在 GraphX 中，triplets 对应着 EdgeTriplet。它是一个三元组视图，这个视图逻辑上将顶点和边的属性保存为一个 RDD[EdgeTriplet[VD, ED]]。可以通过下面的 Sql 表达式表示这个三元视图的含义：

```
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
```

同样，也可以通过下面图解的形式来表示它的含义：



EdgeTriplet 的源代码如下所示：

```
class EdgeTriplet[VD, ED] extends Edge[ED] {
  //源顶点属性
  var srcAttr: VD = _ // nullValue[VD]
  //目标顶点属性
  var dstAttr: VD = _ // nullValue[VD]
  protected[spark] def set(other: Edge[ED]): EdgeTriplet[VD, ED] = {
    srcId = other.srcId
    dstId = other.dstId
    attr = other.attr
    this
  }
}
```

EdgeTriplet 类继承自 Edge 类，我们来看看这个父类：

```
case class Edge[@specialized(Char, Int, Boolean, Byte, Long, Float, Double)
ED] (var srcId: VertexId = 0, var dstId: VertexId = 0, var attr: ED =
null.asInstanceOf[ED]) extends Serializable
```

Edge 类中包含源顶点 id，目标顶点 id 以及边的属性。所以从源代码中我们可以知道，triplets 既包含了边属性也包含了源顶点的 id 和属性、目标顶点的 id 和属性。

2.3 图的构建

GraphX 的 Graph 对象是用户操作图的入口。前面的章节我们介绍过，它包含了边(edges)、顶点(vertices)以及 triplets 三部分，并且这三部分都包含相应的属性，可以携带额外的信息。

2.3.1 构建图的方法

构建图的入口方法有两种，分别是根据边构建和根据边的两个顶点构建。

2.3.1.1 根据边构建图(Graph.fromEdges)

```
def fromEdges[VD: ClassTag, ED: ClassTag](
      edges: RDD[Edge[ED]],
      defaultValue: VD,
      edgeStorageLevel: StorageLevel = StorageLevel.MEMORY_ONLY,
      vertexStorageLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD,
ED] = {
  GraphImpl(edges, defaultValue, edgeStorageLevel, vertexStorageLevel)
}
```

2.3.1.2 根据边的两个顶点数据构建(Graph.fromEdgeTuples)

```
def fromEdgeTuples[VD: ClassTag](
     rawEdges: RDD[(VertexId, VertexId)],
     defaultValue: VD,
     uniqueEdges: Option[PartitionStrategy] = None,
     edgeStorageLevel: StorageLevel = StorageLevel.MEMORY_ONLY,
     vertexStorageLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD,
Int] =
{
  val edges = rawEdges.map(p => Edge(p._1, p._2, 1))
  val graph = GraphImpl(edges, defaultValue, edgeStorageLevel,
vertexStorageLevel)
  uniqueEdges match {
    case Some(p) => graph.partitionBy(p).groupEdges((a, b) => a + b)
    case None => graph
  }
}
```

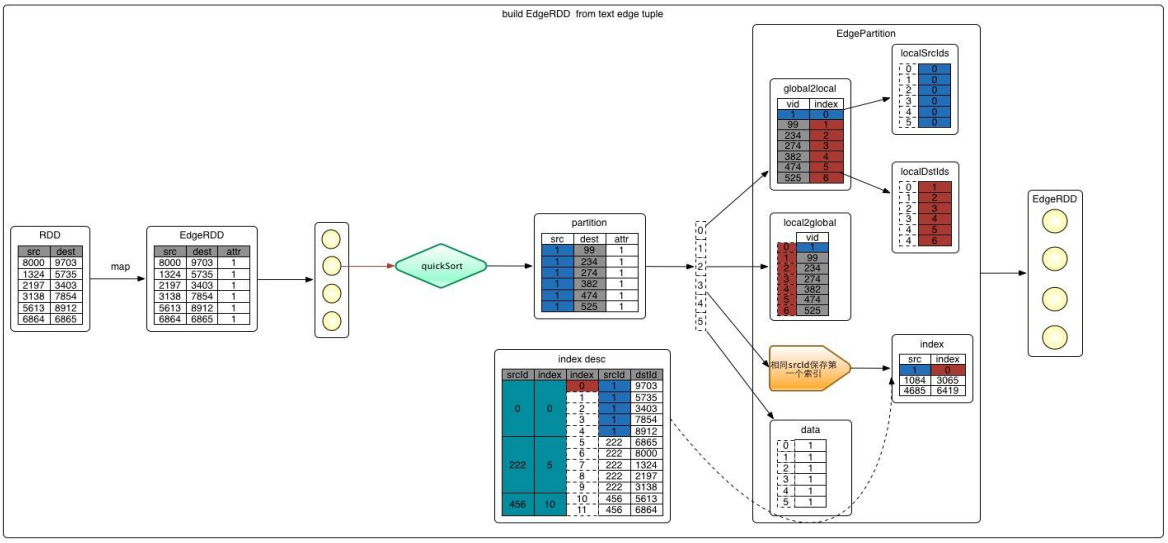
从上面的代码我们知道，不管是根据边构建图还是根据边的两个顶点数据构建，最终都是使用 GraphImpl 来构建的，即调用了 GraphImpl 的 apply 方法。

2.3.2 构建图的过程

构建图的过程很简单，分为三步，它们分别是构建边 EdgeRDD、构建顶点 VertexRDD、生成 Graph 对象。下面分别介绍这三个步骤。

2.3.2.1 构建边 EdgeRDD

从源代码看构建边 EdgeRDD 也分为三步，下图的例子详细说明了这些步骤。



- **1** 从文件中加载信息，转换成 tuple 的形式,即(srcId, dstId)

```
val rawEdgesRdd: RDD[(Long, Long)] =
  sc.textFile(input).filter(s => s != "0,0").repartition(partitionNum).map {
    case line =>
      val ss = line.split(",")
      val src = ss(0).toLong
      val dst = ss(1).toLong
      if (src < dst)
        (src, dst)
      else
        (dst, src)
  }.distinct()
```

- 2 入口，**调用 Graph.fromEdgeTuples(rawEdgesRdd)**

源数据为分割的两个点 ID，把源数据映射成 Edge(srcId, dstId, attr)对象，attr 默认为 1。这样元数据就构建成了 RDD[Edge[ED]],如下面的代码

```
val edges = rawEdges.map(p => Edge(p._1, p._2, 1))
```

- **3** 将 RDD[Edge[ED]]进一步转化成 EdgeRDDImpl[ED, VD]

第二步构建完 RDD[Edge[ED]]之后，GraphX 通过调用 GraphImpl 的 apply 方法来构建 Graph。

```
val graph = GraphImpl(edges, defaultValue, edgeStorageLevel,
vertexStorageLevel)
def apply[VD: ClassTag, ED: ClassTag](
      edges: RDD[Edge[ED]],
      defaultVertexAttr: VD,
      edgeStorageLevel: StorageLevel,
      vertexStorageLevel: StorageLevel): GraphImpl[VD, ED] = {
  fromEdgeRDD(EdgeRDD.fromEdges(edges), defaultVertexAttr, edgeStorageLevel,
vertexStorageLevel)
}
```

在 apply 调用 fromEdgeRDD 之前，代码会调用 EdgeRDD.fromEdges(edges)将 RDD[Edge[ED]]转化成 EdgeRDDImpl[ED, VD]。

```
def fromEdges[ED: ClassTag, VD: ClassTag](edges: RDD[Edge[ED]]):
EdgeRDDImpl[ED, VD] = {
  val edgePartitions = edges.mapPartitionsWithIndex { (pid, iter) =>
    val builder = new EdgePartitionBuilder[ED, VD]
    iter.foreach { e =>
      builder.add(e.srcId, e.dstId, e.attr)
    }
    Iterator((pid, builder.toEdgePartition))
  }
  EdgeRDD.fromEdgePartitions(edgePartitions)
}
```

程序遍历 RDD[Edge[ED]]的每个分区，并调用 builder.toEdgePartition 对分区内的边作相应的处理。

```
def toEdgePartition: EdgePartition[ED, VD] = {
  val edgeArray = edges.trim().array
  new Sorter(Edge.edgeArraySortDataFormat[ED])
    .sort(edgeArray, 0, edgeArray.length, Edge.lexicographicOrdering)
  val localSrcIds = new Array[Int](edgeArray.size)
  val localDstIds = new Array[Int](edgeArray.size)
  val data = new Array[ED](edgeArray.size)
  val index = new GraphXPrimitiveKeyOpenHashMap[VertexId, Int]
  val global2local = new GraphXPrimitiveKeyOpenHashMap[VertexId, Int]
  val local2global = new PrimitiveVector[VertexId]
```

```
  var vertexAttrs = Array.empty[VD]
  //采用列式存储的方式，节省了空间
  if (edgeArray.length > 0) {
    index.update(edgeArray(0).srcId, 0)
    var currSrcId: VertexId = edgeArray(0).srcId
    var currLocalId = -1
    var i = 0
    while (i < edgeArray.size) {
      val srcId = edgeArray(i).srcId
      val dstId = edgeArray(i).dstId
      localSrcIds(i) = global2local.changeValue(srcId,
        { currLocalId += 1; local2global += srcId; currLocalId }, identity)
      localDstIds(i) = global2local.changeValue(dstId,
        { currLocalId += 1; local2global += dstId; currLocalId }, identity)
      data(i) = edgeArray(i).attr
      //相同顶点 srcId 中第一个出现的 srcId 与其下标
      if (srcId != currSrcId) {
        currSrcId = srcId
        index.update(currSrcId, i)
      }
      i += 1
    }
    vertexAttrs = new Array[VD](currLocalId + 1)
  }
  new EdgePartition(
    localSrcIds, localDstIds, data, index, global2local,
local2global.trim().array, vertexAttrs,
    None)
}
```

- **toEdgePartition 的第一步就是对边进行排序。**

按照 srcId 从小到大排序。排序是为了遍历时顺序访问，加快访问速度。采用数组而不是 Map，是因为数组是连续的内存单元，具有原子性，避免了 Map 的 hash 问题，访问速度快。

- **toEdgePartition 的第二步就是填充 localSrcIds,localDstIds, data, index, global2local, local2global, vertexAttrs。**

数组 localSrcIds,localDstIds 中保存的是通过 global2local.changeValue(srcId/dstId)转换而成的分区本地索引。可以通过

localSrcIds、localDstIds 数组中保存的索引位从 local2global 中查到具体的 VertexId。

global2local 是一个简单的，key 值非负的快速 hash map：GraphXPrimitiveKeyOpenHashMap，保存 vertextId 和本地索引的映射关系。global2local 中包含当前 partition 所有 srcId、dstId 与本地索引的映射关系。

data 就是当前分区的 attr 属性数组。

我们知道相同的 srcId 可能对应不同的 dstId。按照 srcId 排序之后，相同的 srcId 会出现多行，如上图中的 index desc 部分。index 中记录的是相同 srcId 中第一个出现的 srcId 与其下标。

local2global 记录的是所有的 VertexId 信息的数组。形如：srcId,dstId,srcId,dstId,srcId,dstId,srcId,dstId。其中会包含相同的 srcId。即：当前分区所有 vertextId 的顺序实际值。

我们可以通过根据本地下标取 VertexId，也可以根据 VertexId 取本地下标，取相应的属性。

```
// 根据本地下标取 VertexId
localSrcIds/localDstIds -> index -> local2global -> VertexId
// 根据 VertexId 取本地下标，取属性
VertexId -> global2local -> index -> data -> attr object
```

2.3.2.2 构建顶点 VertexRDD

紧接着上面构建边 RDD 的代码，我们看看方法 fromEdgeRDD 的实现。

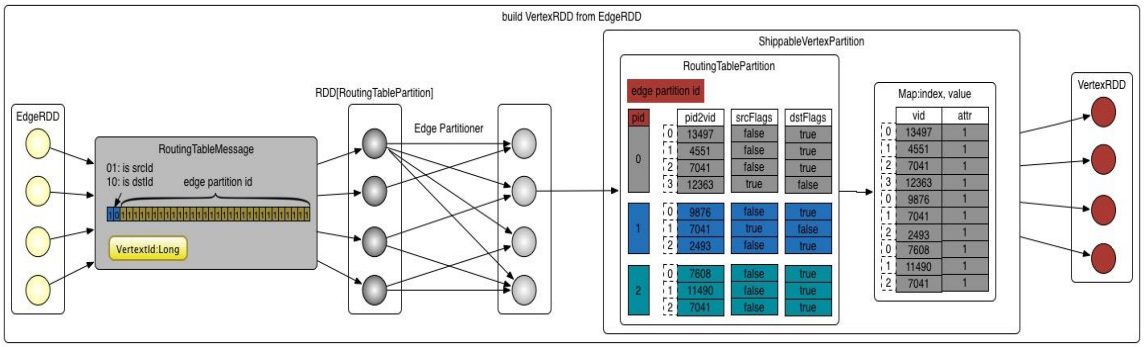
```
private def fromEdgeRDD[VD: ClassTag, ED: ClassTag](
        edges: EdgeRDDImpl[ED, VD],
        defaultVertexAttr: VD,
        edgeStorageLevel: StorageLevel,
        vertexStorageLevel: StorageLevel): GraphImpl[VD, ED] = {
  val edgesCached = edges.withTargetStorageLevel(edgeStorageLevel).cache()
  val vertices = VertexRDD.fromEdges(edgesCached, edgesCached.partitions.size,
defaultVertexAttr)
    .withTargetStorageLevel(vertexStorageLevel)
  fromExistingRDDs(vertices, edgesCached)
}
```

从上面的代码我们可以知道，GraphX 使用 VertexRDD.fromEdges 构建顶点 VertexRDD，当然我们把边 RDD 作为参数传入。

```
def fromEdges[VD: ClassTag](
                        edges: EdgeRDD[_], numPartitions: Int, defaultVal:
VD): VertexRDD[VD] = {
  //1 创建路由表
  val routingTables = createRoutingTables(edges, new
HashPartitioner(numPartitions))
```

```
  //2 根据路由表生成分区对象 vertexPartitions
  val vertexPartitions = routingTables.mapPartitions({ routingTableIter =>
    val routingTable =
      if (routingTableIter.hasNext) routingTableIter.next() else
RoutingTablePartition.empty
    Iterator(ShippableVertexPartition(Iterator.empty, routingTable,
defaultVal))
  }, preservesPartitioning = true)
  //3 创建 VertexRDDImpl 对象
  new VertexRDDImpl(vertexPartitions)
}
```

构建顶点 VertexRDD 的过程分为三步，如上代码中的注释。它的构建过程如下图所示：



- **创建路由表**

为了能通过点找到边，每个点需要保存点到边的信息，这些信息保存在 RoutingTablePartition 中。

```
private[graphx] def createRoutingTables(edges: EdgeRDD[_], vertexPartitioner:
Partitioner): RDD[RoutingTablePartition] = {
  // 将 edge partition 中的数据转换成 RoutingTableMessage 类型,
  val vid2pid = edges.partitionsRDD.mapPartitions(_.flatMap(
    Function.tupled(RoutingTablePartition.edgePartitionToMsgs)))
}
```

上述程序首先将边分区中的数据转换成 RoutingTableMessage 类型，即 tuple(VertexId,Int)类型。

```
def edgePartitionToMsgs(pid: PartitionID, edgePartition: EdgePartition[_, _])
: Iterator[RoutingTableMessage] = {
```

```
  val map = new GraphXPrimitiveKeyOpenHashMap[VertexId, Byte]
  edgePartition.iterator.foreach { e =>
    map.changeValue(e.srcId, 0x1, (b: Byte) => (b | 0x1).toByte)
    map.changeValue(e.dstId, 0x2, (b: Byte) => (b | 0x2).toByte)
  }
  map.iterator.map { vidAndPosition =>
    val vid = vidAndPosition._1
    val position = vidAndPosition._2
    toMessage(vid, pid, position)
  }
}
//`30-0`比特位表示边分区`ID`,`32-31`比特位表示标志位
private def toMessage(vid: VertexId, pid: PartitionID, position: Byte):
RoutingTableMessage = {
  val positionUpper2 = position << 30
  val pidLower30 = pid & 0x3FFFFFFF
  (vid, positionUpper2 | pidLower30)
}
```

根据代码，我们可以知道程序使用 int 的 32-31 比特位表示标志位，即 01: isSrcId ,10: isDstId。30-0 比特位表示边分区 ID。这样做可以节省内存。 RoutingTableMessage 表达的信息是：顶点 id 和它相关联的边的分区 id 是放在一起的,所以任何时候，我们都可以通过 RoutingTableMessage 找到顶点关联的边。

- **根据路由表生成分区对象**

```
private[graphx] def createRoutingTables(
                                edges: EdgeRDD[_], vertexPartitioner:
Partitioner): RDD[RoutingTablePartition] = {
  // 将 edge partition 中的数据转换成 RoutingTableMessage 类型,
  val numEdgePartitions = edges.partitions.size
  vid2pid.partitionBy(vertexPartitioner).mapPartitions(
    iter => Iterator(RoutingTablePartition.fromMsgs(numEdgePartitions, iter)),
    preservesPartitioning = true)
}
```

我们将第 1 步生成的 vid2pid 按照 HashPartitioner 重新分区。我们看看 RoutingTablePartition.fromMsgs 方法。

```
def fromMsgs(numEdgePartitions: Int, iter: Iterator[RoutingTableMessage])
: RoutingTablePartition = {
```

```
  val pid2vid = Array.fill(numEdgePartitions)(new PrimitiveVector[VertexId])
  val srcFlags = Array.fill(numEdgePartitions)(new PrimitiveVector[Boolean])
  val dstFlags = Array.fill(numEdgePartitions)(new PrimitiveVector[Boolean])
  for (msg <- iter) {
    val vid = vidFromMessage(msg)
    val pid = pidFromMessage(msg)
    val position = positionFromMessage(msg)
    pid2vid(pid) += vid
    srcFlags(pid) += (position & 0x1) != 0
    dstFlags(pid) += (position & 0x2) != 0
  }
  new RoutingTablePartition(pid2vid.zipWithIndex.map {
    case (vids, pid) => (vids.trim().array, toBitSet(srcFlags(pid)),
toBitSet(dstFlags(pid)))
  })
}
```

该方法从 RoutingTableMessage 获取数据，将 vid, 边 pid, isSrcId/isDstId 重新封装到 pid2vid，srcFlags，dstFlags 这三个数据结构中。它们表示当前顶点分区中的点在边分区的分布。 想象一下，重新分区后，新分区中的点可能来自于不同的边分区，所以一个点要找到边，就需要先确定边的分区号 pid, 然后在确定的边分区中确定是 srcId 还是 dstId, 这样就找到了边。 新分区中保存 vids.trim().array, toBitSet(srcFlags(pid)), toBitSet(dstFlags(pid))这样的记录。这里转换为 toBitSet 保存是为了节省空间。

根据上文生成的 routingTables,重新封装路由表里的数据结构为 ShippableVertexPartition。ShippableVertexPartition 会合并相同重复点的属性 attr 对象，补全缺失的 attr 对象。

```
def apply[VD: ClassTag](
                      iter: Iterator[(VertexId, VD)], routingTable:
RoutingTablePartition, defaultVal: VD,
                      mergeFunc: (VD, VD) => VD): ShippableVertexPartition[VD] =
{
  val map = new GraphXPrimitiveKeyOpenHashMap[VertexId, VD]
  // 合并顶点
  iter.foreach { pair =>
    map.setMerge(pair._1, pair._2, mergeFunc)
  }
  // 不全缺失的属性值
  routingTable.iterator.foreach { vid =>
    map.changeValue(vid, defaultVal, identity)
  }
```

```
  new ShippableVertexPartition(map.keySet, map._values, map.keySet.getBitSet,
routingTable)
}
//ShippableVertexPartition 定义
ShippableVertexPartition[VD: ClassTag](
val index: VertexIdToIndexMap,
val values: Array[VD],
val mask: BitSet,
val routingTable: RoutingTablePartition)
```

map 就是映射 vertexId->attr，index 就是顶点集合，values 就是顶点集对应的属性集，mask 指顶点集的 BitSet。

2.3.2.3 生成 Graph 对象

使用上述构建的 edgeRDD 和 vertexRDD，使用 new GraphImpl(vertices, new ReplicatedVertexView(edges.asInstanceOf[EdgeRDDImpl[ED, VD]])) 就可以生成 Graph 对象。ReplicatedVertexView 是点和边的视图，用来管理运送(shipping)顶点属性到 EdgeRDD 的分区。当顶点属性改变时，我们需要运送它们到边分区来更新保存在边分区的顶点属性。 注意，在 ReplicatedVertexView 中不要保存一个对边的引用，因为在属性运送等级升级后，这个引用可能会发生改变。

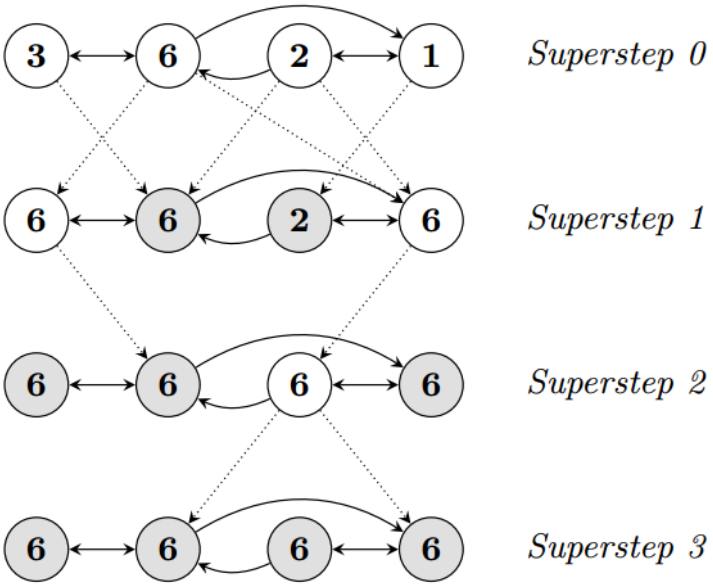
```
class ReplicatedVertexView[VD: ClassTag, ED: ClassTag](var edges:
EdgeRDDImpl[ED, VD], var hasSrcId: Boolean = false, var hasDstId: Boolean =
false)
```

2.4 计算模式

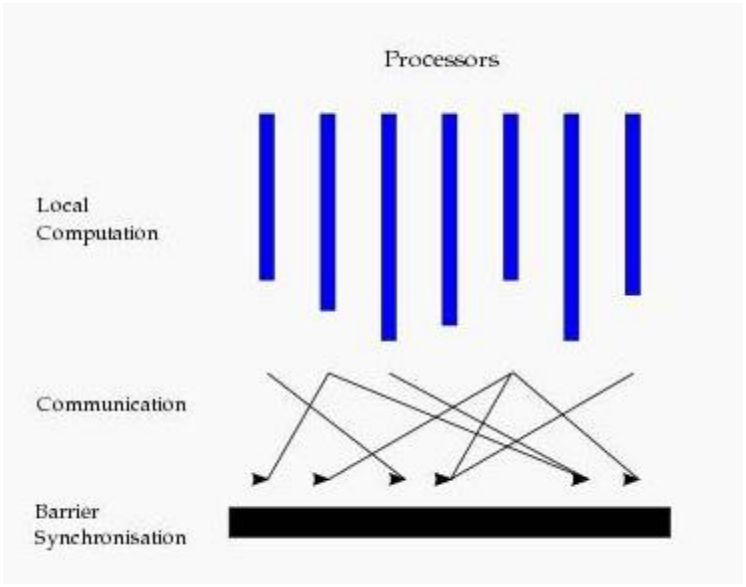
2.4.1 BSP 计算模式

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。

在 BSP 中，一次计算过程由一系列全局超步组成，每一个超步由并发计算、通信和同步三个步骤组成。同步完成，标志着这个超步的完成及下一个超步的开始。 BSP 模式的准则是批量同步(bulk synchrony)，其独特之处在于超步(superstep)概念的引入。一个 BSP 程序同时具有水平和垂直两个方面的结构。从垂直上看,一个 BSP 程序由一系列串行的超步(superstep)组成,如图所示:



从水平上看，在一个超步中，所有的进程并行执行局部计算。一个超步可分为三个阶段，如图所示:



- 本地计算阶段，每个处理器只对存储在本地内存中的数据进行本地计算。
- 全局通信阶段，对任何非本地数据进行操作。
- 栅栏同步阶段，等待所有通信行为的结束。

BSP 模型有如下几个特点：

1 将计算划分为一个一个的超步(superstep)，有效避免死锁;

2 将处理器和路由器分开，强调了计算任务和通信任务的分开，而路由器仅仅完成点到点的消息传递，不提供组合、复制和广播等功能，这样做既掩盖具体的互连网络拓扑，又简化了通信协议；

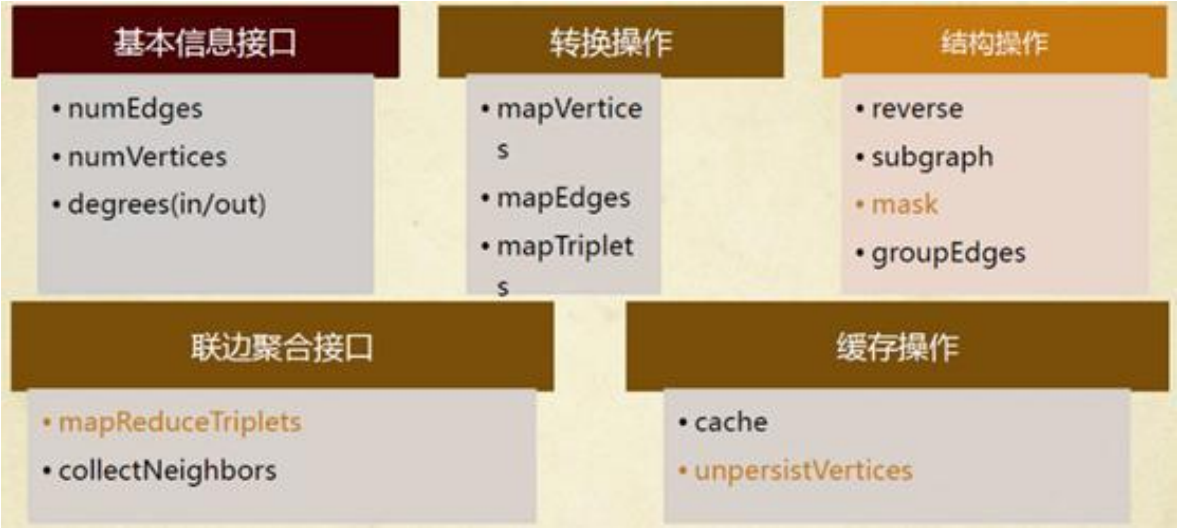
3 采用障碍同步的方式、以硬件实现的全局同步是可控的粗粒度级，提供了执行紧耦合同步式并行算法的有效方式

2.4.2 图操作一览

正如 RDDs 有基本的操作 map, filter 和 reduceByKey 一样，属性图也有基本的集合操作，这些操作采用用户自定义的函数并产生包含转换特征和结构的新图。定义在 Graph 中的核心操作是经过优化的实现。表示为核心操作的组合的便捷操作定义在 GraphOps 中。然而，因为有 Scala 的隐式转换，定义在 GraphOps 中的操作可以作为 Graph 的成员自动使用。例如，我们可以通过下面的方式计算每个顶点(定义在 GraphOps 中)的入度。

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

区分核心图操作和 GraphOps 的原因是为了在将来支持不同的图表示。每个图表示都必须提供核心操作的实现并重用很多定义在 GraphOps 中的有用操作。



2.4.3 操作一览

以下是定义在 Graph 和 GraphOps 中（为了简单起见，表现为图的成员）的功能的快速浏览。注意，某些函数签名已经简化（如默认参数和类型的限制已删除），一些更高级的功能已经被删除，所以请参阅 API 文档了解官方的操作列表。

```
/** 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]
  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 aggregateMessages[Msg: ClassTag](
                                  sendMsg: EdgeContext[VD, ED, Msg] => Unit,
                                  mergeMsg: (Msg, Msg) => Msg,
                                  tripletFields: TripletFields =
TripletFields.All)
  : 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]
}
```

2.4.4 转换操作

GraphX 中的转换操作主要有 mapVertices,mapEdges 和 mapTriplets 三个，它们在 Graph 文件中定义，在 GraphImpl 文件中实现。下面分别介绍这三个方法。

2.4.4.1 mapVertices

mapVertices 用来更新顶点属性。从图的构建那章我们知道，顶点属性保存在边分区中，所以我们需要改变的是边分区中的属性。

```
override def mapVertices[VD2: ClassTag]
(f: (VertexId, VD) => VD2)(implicit eq: VD =:= VD2 = null): Graph[VD2, ED] = {
  if (eq != null) {
    vertices.cache()
    // 使用方法 f 处理 vertices
    val newVerts = vertices.mapVertexPartitions(_.map(f)).cache()
    //获得两个不同 vertexRDD 的不同
    val changedVerts = vertices.asInstanceOf[VertexRDD[VD2]].diff(newVerts)
    //更新 ReplicatedVertexView
    val newReplicatedVertexView =
replicatedVertexView.asInstanceOf[ReplicatedVertexView[VD2, ED]]
      .updateVertices(changedVerts)
    new GraphImpl(newVerts, newReplicatedVertexView)
  } else {
    GraphImpl(vertices.mapVertexPartitions(_.map(f)),
replicatedVertexView.edges)
  }
}
```

上面的代码中，当 VD 和 VD2 类型相同时，我们可以重用没有发生变化的点，否则需要重新创建所有的点。我们分析 VD 和 VD2 相同的情况，分四步处理。

- 1 使用方法 f 处理 vertices,获得新的 VertexRDD
- 2 使用在 VertexRDD 中定义的 diff 方法求出新 VertexRDD 和源 VertexRDD 的不同

```
override def diff(other: VertexRDD[VD]): VertexRDD[VD] = {
  val otherPartition = other match {
    case other: VertexRDD[_] if this.partitioner == other.partitioner =>
      other.partitionsRDD
    case _ =>
      VertexRDD(other.partitionBy(this.partitioner.get)).partitionsRDD
  }
  val newPartitionsRDD = partitionsRDD.zipPartitions(
    otherPartition, preservesPartitioning = true
  ) { (thisIter, otherIter) =>
    val thisPart = thisIter.next()
    val otherPart = otherIter.next()
    Iterator(thisPart.diff(otherPart))
  }
  this.withPartitionsRDD(newPartitionsRDD)
}
```

这个方法首先处理新生成的 VertexRDD 的分区，如果它的分区和源 VertexRDD 的分区一致，那么直接取出它的 partitionsRDD,否则重新分区后取出它的 partitionsRDD。 针对新旧两个 VertexRDD 的所有分区，调用 VertexPartitionBaseOps 中的 diff 方法求得分区的不同。

```
def diff(other: Self[VD]): Self[VD] = {
  //首先判断
  if (self.index != other.index) {
    diff(createUsingIndex(other.iterator))
  } else {
    val newMask = self.mask & other.mask
    var i = newMask.nextSetBit(0)
    while (i >= 0) {
      if (self.values(i) == other.values(i)) {
        newMask.unset(i)
      }
      i = newMask.nextSetBit(i + 1)
    }
    this.withValues(other.values).withMask(newMask)
  }
}
```

该方法隐藏两个 VertexRDD 中相同的顶点信息，得到一个新的 VertexRDD。

- 3 更新 ReplicatedVertexView

```
def updateVertices(updates: VertexRDD[VD]): ReplicatedVertexView[VD, ED] = {
  //生成一个 VertexAttributeBlock
  val shippedVerts = updates.shipVertexAttributes(hasSrcId, hasDstId)
    .setName("ReplicatedVertexView.updateVertices - shippedVerts %s %s
(broadcast)".format(
      hasSrcId, hasDstId))
    .partitionBy(edges.partitioner.get)
  //生成新的边 RDD
  val newEdges =
edges.withPartitionsRDD(edges.partitionsRDD.zipPartitions(shippedVerts) {
    (ePartIter, shippedVertsIter) => ePartIter.map {
      case (pid, edgePartition) =>
        (pid,
edgePartition.updateVertices(shippedVertsIter.flatMap(_._2.iterator)))
    }
  })
  new ReplicatedVertexView(newEdges, hasSrcId, hasDstId)
}
```

updateVertices 方法返回一个新的 ReplicatedVertexView,它更新了边分区中包含的顶点属性。我们看看它的实现过程。首先看 shipVertexAttributes 方法的调用。 调用 shipVertexAttributes 方法会生成一个 VertexAttributeBlock，VertexAttributeBlock 包含当前分区的顶点属性，这些属性可以在特定的边分区使用。

```
def shipVertexAttributes(
                      shipSrc: Boolean, shipDst: Boolean):
Iterator[(PartitionID, VertexAttributeBlock[VD])] = {
  Iterator.tabulate(routingTable.numEdgePartitions) { pid =>
    val initialSize = if (shipSrc && shipDst) routingTable.partitionSize(pid)
else 64
    val vids = new PrimitiveVector[VertexId](initialSize)
    val attrs = new PrimitiveVector[VD](initialSize)
    var i = 0
    routingTable.foreachWithinEdgePartition(pid, shipSrc, shipDst) { vid =>
      if (isDefined(vid)) {
        vids += vid
        attrs += this(vid)
      }
```

```
      i += 1
    }
    //（边分区 id，VertexAttributeBlock（顶点 id，属性））
    (pid, new VertexAttributeBlock(vids.trim().array, attrs.trim().array))
  }
}
```

获得新的顶点属性之后，我们就可以调用 updateVertices 更新边中顶点的属性了，如下面代码所示：

```
edgePartition.updateVertices(shippedVertsIter.flatMap(_._2.iterator))
//更新 EdgePartition 的属性
def updateVertices(iter: Iterator[(VertexId, VD)]): EdgePartition[ED, VD] = {
  val newVertexAttrs = new Array[VD](vertexAttrs.length)
  System.arraycopy(vertexAttrs, 0, newVertexAttrs, 0, vertexAttrs.length)
  while (iter.hasNext) {
    val kv = iter.next()
    //global2local 获得顶点的本地 index
    newVertexAttrs(global2local(kv._1)) = kv._2
  }
  new EdgePartition(
    localSrcIds, localDstIds, data, index, global2local, local2global,
newVertexAttrs,
    activeSet)
}
```

2.4.4.2 mapEdges

mapEdges 用来更新边属性。

```
override def mapEdges[ED2: ClassTag](
                                f: (PartitionID, Iterator[Edge[ED]]) =>
Iterator[ED2]): Graph[VD, ED2] = {
  val newEdges = replicatedVertexView.edges
    .mapEdgePartitions((pid, part) => part.map(f(pid, part.iterator)))
  new GraphImpl(vertices, replicatedVertexView.withEdges(newEdges))
}
```

相比于 mapVertices，mapEdges 显然要简单得多，它只需要根据方法 f 生成新的 EdgeRDD,然后再初始化即可。

2.4.4.3 mapTriplets

mapTriplets 用来更新边属性。

```
override def mapTriplets[ED2: ClassTag](
    f: (PartitionID, Iterator[EdgeTriplet[VD, ED]]) => Iterator[ED2],
    tripletFields: TripletFields): Graph[VD, ED2] = {
  vertices.cache()
  replicatedVertexView.upgrade(vertices, tripletFields.useSrc,
tripletFields.useDst)
  val newEdges = replicatedVertexView.edges.mapEdgePartitions { (pid, part) =>
    part.map(f(pid, part.tripletIterator(tripletFields.useSrc,
tripletFields.useDst)))
  }
  new GraphImpl(vertices, replicatedVertexView.withEdges(newEdges))
}
```

这段代码中，replicatedVertexView 调用 upgrade 方法修改当前的 ReplicatedVertexView，使调用者可以访问到指定级别的边信息（如仅仅可以读源顶点的属性）。

```
def upgrade(vertices: VertexRDD[VD], includeSrc: Boolean, includeDst: Boolean)
{
  //判断传递级别
  val shipSrc = includeSrc && !hasSrcId
  val shipDst = includeDst && !hasDstId
  if (shipSrc || shipDst) {
    val shippedVerts: RDD[(Int, VertexAttributeBlock[VD])] =
      vertices.shipVertexAttributes(shipSrc, shipDst)
        .setName("ReplicatedVertexView.upgrade(%s, %s) - shippedVerts %s %s
(broadcast)".format(
          includeSrc, includeDst, shipSrc, shipDst))
        .partitionBy(edges.partitioner.get)
    val newEdges =
edges.withPartitionsRDD(edges.partitionsRDD.zipPartitions(shippedVerts) {
      (ePartIter, shippedVertsIter) => ePartIter.map {
        case (pid, edgePartition) =>
          (pid,
edgePartition.updateVertices(shippedVertsIter.flatMap(_._2.iterator)))
      }
    })
    edges = newEdges
    hasSrcId = includeSrc
    hasDstId = includeDst
  }
}
```

最后，用 f 处理边，生成新的 RDD，最后用新的数据初始化图。

2.4.5 结构操作

当前的 GraphX 仅仅支持一组简单的常用结构性操作。下面是基本的结构性操作列表。

```
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]
}
```

下面分别介绍这四种函数的原理。

2.4.5.1 reverse

reverse 操作返回一个新的图，这个图的边的方向都是反转的。例如，这个操作可以用来计算反转的 PageRank。因为反转操作没有修改顶点或者边的属性或者改变边的数量，所以我们可以 在不移动或者复制数据的情况下有效地实现它。

```
override def reverse: Graph[VD, ED] = {
  new GraphImpl(vertices.reverseRoutingTables(),
replicatedVertexView.reverse())
}
def reverse(): ReplicatedVertexView[VD, ED] = {
  val newEdges = edges.mapEdgePartitions((pid, part) => part.reverse)
  new ReplicatedVertexView(newEdges, hasDstId, hasSrcId)
}
//EdgePartition 中的 reverse
def reverse: EdgePartition[ED, VD] = {
  val builder = new ExistingEdgePartitionBuilder[ED, VD](
    global2local, local2global, vertexAttrs, activeSet, size)
  var i = 0
  while (i < size) {
    val localSrcId = localSrcIds(i)
    val localDstId = localDstIds(i)
    val srcId = local2global(localSrcId)
```

```
    val dstId = local2global(localDstId)
    val attr = data(i)
    //将源顶点和目标顶点换位置
    builder.add(dstId, srcId, localDstId, localSrcId, attr)
    i += 1
  }
  builder.toEdgePartition
}
```

2.4.5.2 subgraph

subgraph 操作利用顶点和边的判断式（predicates），返回的图仅仅包含满足顶点判断式的顶点、满足边判断式的边以及满足顶点判断式的 triple。subgraph 操作可以用于很多场景，如获取 感兴趣的顶点和边组成的图或者获取清除断开连接后的图。

```
override def subgraph(
                    epred: EdgeTriplet[VD, ED] => Boolean = x => true,
                    vpred: (VertexId, VD) => Boolean = (a, b) => true):
Graph[VD, ED] = {
  vertices.cache()
  // 过滤 vertices，重用 partitioner 和索引
  val newVerts = vertices.mapVertexPartitions(_.filter(vpred))
  // 过滤 triplets
  replicatedVertexView.upgrade(vertices, true, true)
  val newEdges = replicatedVertexView.edges.filter(epred, vpred)
  new GraphImpl(newVerts, replicatedVertexView.withEdges(newEdges))
}
  // 该代码显示，subgraph 方法的实现分两步：先过滤 VertexRDD，然后再过滤 EdgeRDD。如上，过滤 VertexRDD 比较简单，我们重点看过滤 EdgeRDD 的过程。
def filter(
          epred: EdgeTriplet[VD, ED] => Boolean,
          vpred: (VertexId, VD) => Boolean): EdgeRDDImpl[ED, VD] = {
  mapEdgePartitions((pid, part) => part.filter(epred, vpred))
}
//EdgePartition 中的 filter 方法
def filter(
          epred: EdgeTriplet[VD, ED] => Boolean,
          vpred: (VertexId, VD) => Boolean): EdgePartition[ED, VD] = {
  val builder = new ExistingEdgePartitionBuilder[ED, VD](
    global2local, local2global, vertexAttrs, activeSet)
```

```
  var i = 0
  while (i < size) {
    // The user sees the EdgeTriplet, so we can't reuse it and must create one
per edge.
    val localSrcId = localSrcIds(i)
    val localDstId = localDstIds(i)
    val et = new EdgeTriplet[VD, ED]
    et.srcId = local2global(localSrcId)
    et.dstId = local2global(localDstId)
    et.srcAttr = vertexAttrs(localSrcId)
    et.dstAttr = vertexAttrs(localDstId)
    et.attr = data(i)
    if (vpred(et.srcId, et.srcAttr) && vpred(et.dstId, et.dstAttr) &&
epred(et)) {
      builder.add(et.srcId, et.dstId, localSrcId, localDstId, et.attr)
    }
    i += 1
  }
  builder.toEdgePartition
}
```

因为用户可以看到 EdgeTriplet 的信息，所以我们不能重用 EdgeTriplet，需要重新创建一个，然后在用 epred 函数处理。

2.4.5.3 mask

mask 操作构造一个子图，这个子图包含输入图中包含的顶点和边。它的实现很简单，顶点和边均做 inner join 操作即可。这个操作可以和 subgraph 操作相结合，基于另外一个相关图的特征去约束一个图。

```
override def mask[VD2: ClassTag, ED2: ClassTag] (
     other: Graph[VD2, ED2]): Graph[VD, ED] = {
  val newVerts = vertices.innerJoin(other.vertices) { (vid, v, w) => v }
  val newEdges = replicatedVertexView.edges.innerJoin(other.edges) { (src, dst,
v, w) => v }
  new GraphImpl(newVerts, replicatedVertexView.withEdges(newEdges))
}
```

2.4.5.4 groupEdges

groupEdges 操作合并多重图中的并行边(如顶点对之间重复的边)。在大量的应用程序中，并行的边可以合并（它们的权重合并）为一条边从而降低图的大小。

```
override def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED] = {
  val newEdges = replicatedVertexView.edges.mapEdgePartitions(
    (pid, part) => part.groupEdges(merge))
  new GraphImpl(vertices, replicatedVertexView.withEdges(newEdges))
}
def groupEdges(merge: (ED, ED) => ED): EdgePartition[ED, VD] = {
  val builder = new ExistingEdgePartitionBuilder[ED, VD](
    global2local, local2global, vertexAttrs, activeSet)
  var currSrcId: VertexId = null.asInstanceOf[VertexId]
  var currDstId: VertexId = null.asInstanceOf[VertexId]
  var currLocalSrcId = -1
  var currLocalDstId = -1
  var currAttr: ED = null.asInstanceOf[ED]
  // 迭代处理所有的边
  var i = 0
  while (i < size) {
    //如果源顶点和目的顶点都相同
    if (i > 0 && currSrcId == srcIds(i) && currDstId == dstIds(i)) {
      // 合并属性
      currAttr = merge(currAttr, data(i))
    } else {
      // This edge starts a new run of edges
      if (i > 0) {
        // 添加到 builder 中
        builder.add(currSrcId, currDstId, currLocalSrcId, currLocalDstId,
currAttr)
      }
      // Then start accumulating for a new run
      currSrcId = srcIds(i)
      currDstId = dstIds(i)
      currLocalSrcId = localSrcIds(i)
      currLocalDstId = localDstIds(i)
      currAttr = data(i)
    }
    i += 1
  }
```

```
  if (size > 0) {
    builder.add(currSrcId, currDstId, currLocalSrcId, currLocalDstId, currAttr)
  }
  builder.toEdgePartition
}
```

在图构建那章我们说明过，存储的边按照源顶点 id 排过序，所以上面的代码可以通过一次迭代完成对所有相同边的处理。

2.4.5.5 应用举例

```
// 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(_))

/ 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)
```

2.4.6 关联操作

在许多情况下，有必要将外部数据加入到图中。例如，我们可能有额外的用户属性需要合并到已有的图中或者我们可能想从一个图中取出顶点特征加入到另外一个图中。这些任务可以用 join 操作完成。 主要的 join 操作如下所示。

```
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]
}
```

joinVertices 操作 join 输入 RDD 和顶点，返回一个新的带有顶点特征的图。这些特征是通过在连接顶点的结果上使用用户定义的 map 函数获得的。没有匹配的顶点保留其原始值。 下面详细地来分析这两个函数。

2.4.6.1 joinVertices

```
def joinVertices[U: ClassTag](table: RDD[(VertexId, U)])(mapFunc: (VertexId,
VD, U) => VD)
: Graph[VD, ED] = {
  val uf = (id: VertexId, data: VD, o: Option[U]) => {
    o match {
      case Some(u) => mapFunc(id, data, u)
      case None => data
    }
  }
  graph.outerJoinVertices(table)(uf)
}
```

我们可以看到，joinVertices 的实现是通过 outerJoinVertices 来实现的。这是因为 join 本来就是 outer join 的一种特例。

2.4.6.2 outerJoinVertices

```
override def outerJoinVertices[U: ClassTag, VD2: ClassTag]
(other: RDD[(VertexId, U)])
(updateF: (VertexId, VD, Option[U]) => VD2)
(implicit eq: VD =:= VD2 = null): Graph[VD2, ED] = {
  if (eq != null) {
    vertices.cache()
    // updateF preserves type, so we can use incremental replication
    val newVerts = vertices.leftJoin(other)(updateF).cache()
    val changedVerts = vertices.asInstanceOf[VertexRDD[VD2]].diff(newVerts)
    val newReplicatedVertexView =
replicatedVertexView.asInstanceOf[ReplicatedVertexView[VD2, ED]]
      .updateVertices(changedVerts)
    new GraphImpl(newVerts, newReplicatedVertexView)
  } else {
    // updateF does not preserve type, so we must re-replicate all vertices
    val newVerts = vertices.leftJoin(other)(updateF)
    GraphImpl(newVerts, replicatedVertexView.edges)
  }
}
```

通过以上的代码我们可以看到，如果 updateF 不改变类型，我们只需要创建改变的顶点即可，否则我们要重新创建所有的顶点。我们讨论不改变类型的情况。 这种情况分三步。

- 1 修改顶点属性值

```
val newVerts = vertices.leftJoin(other)(updateF).cache()
```

这一步会用顶点 RDD join 传入的 RDD，然后用 updateF 作用 joinRDD 中的所有顶点，改变它们的值。

- 2 找到发生改变的顶点

```
val changedVerts = vertices.asInstanceOf[VertexRDD[VD2]].diff(newVerts)
```

- 3 更新 newReplicatedVertexView 中边分区中的顶点属性

```
val newReplicatedVertexView =
replicatedVertexView.asInstanceOf[ReplicatedVertexView[VD2, ED]]
  .updateVertices(changedVerts)
```

2.4.7 聚合操作

GraphX 中提供的聚合操作有 aggregateMessages、collectNeighborIds 和 collectNeighbors 三个，其中 aggregateMessages 在 GraphImpl 中实现，collectNeighborIds 和 collectNeighbors 在 GraphOps 中实现。下面分别介绍这几个方法。

2.4.7.1 aggregateMessages

2.4.7.1.1 aggregateMessages 接口

aggregateMessages 是 GraphX 最重要的 API，用于替换 mapReduceTriplets。目前 mapReduceTriplets 最终也是通过 aggregateMessages 来实现的。它主要功能是向邻边发消息，合并邻边收到的消息，返回 messageRDD。 aggregateMessages 的接口如下：

```
def aggregateMessages[A: ClassTag](
     sendMsg: EdgeContext[VD, ED, A] => Unit,
     mergeMsg: (A, A) => A,
     tripletFields: TripletFields = TripletFields.All)
: VertexRDD[A] = {
  aggregateMessagesWithActiveSet(sendMsg, mergeMsg, tripletFields, None)
}
```

该接口有三个参数，分别为发消息函数，合并消息函数以及发消息的方向。

- sendMsg： 发消息函数

```
private def sendMsg(ctx: EdgeContext[KCoreVertex, Int, Map[Int, Int]]): Unit =
{
  ctx.sendToDst(Map(ctx.srcAttr.preKCore -> -1, ctx.srcAttr.curKCore -> 1))
  ctx.sendToSrc(Map(ctx.dstAttr.preKCore -> -1, ctx.dstAttr.curKCore -> 1))
}
```

- mergeMsg：合并消息函数

该函数用于在 Map 阶段每个 edge 分区中每个点收到的消息合并，并且它还用于 reduce 阶段，合并不同分区的消息。合并 vertexId 相同的消息。

- tripletFields：定义发消息的方向

2.4.7.1.2 aggregateMessages 处理流程

aggregateMessages 方法分为 Map 和 Reduce 两个阶段，下面我们分别就这两个阶段说明。

2.4.7.1.2.1 Map 阶段

从入口函数进入 aggregateMessagesWithActiveSet 函数，该函数首先使用 VertexRDD[VD]更新 replicatedVertexView，只更新其中 vertexRDD 中 attr 对象。如构建图中介绍的，replicatedVertexView 是点和边的视图，点的属性有变化，要更新边中包含的点的 attr。

```
replicatedVertexView.upgrade(vertices, tripletFields.useSrc,
tripletFields.useDst)
val view = activeSetOpt match {
  case Some((activeSet, _)) =>
    //返回只包含活跃顶点的 replicatedVertexView
    replicatedVertexView.withActiveSet(activeSet)
  case None =>
    replicatedVertexView
}
```

程序然后会对 replicatedVertexView 的 edgeRDD 做 mapPartitions 操作，所有的操作都在每个边分区的迭代中完成，如下面的代码：

```
val preAgg = view.edges.partitionsRDD.mapPartitions(_.flatMap {
  case (pid, edgePartition) =>
    // 选择 scan 方法
    val activeFraction = edgePartition.numActives.getOrElse(0) /
edgePartition.indexSize.toFloat
    activeDirectionOpt match {
      case Some(EdgeDirection.Both) =>
        if (activeFraction < 0.8) {
          edgePartition.aggregateMessagesIndexScan(sendMsg, mergeMsg,
tripletFields,
            EdgeActiveness.Both)
        } else {
          edgePartition.aggregateMessagesEdgeScan(sendMsg, mergeMsg,
tripletFields,
            EdgeActiveness.Both)
        }
      case Some(EdgeDirection.Either) =>
        edgePartition.aggregateMessagesEdgeScan(sendMsg, mergeMsg,
tripletFields,
          EdgeActiveness.Either)
      case Some(EdgeDirection.Out) =>
        if (activeFraction < 0.8) {
          edgePartition.aggregateMessagesIndexScan(sendMsg, mergeMsg,
tripletFields,
            EdgeActiveness.SrcOnly)
        } else {
          edgePartition.aggregateMessagesEdgeScan(sendMsg, mergeMsg,
tripletFields,
            EdgeActiveness.SrcOnly)
        }
      case Some(EdgeDirection.In) =>
        edgePartition.aggregateMessagesEdgeScan(sendMsg, mergeMsg,
tripletFields,
          EdgeActiveness.DstOnly)
      case _ => // None
        edgePartition.aggregateMessagesEdgeScan(sendMsg, mergeMsg,
tripletFields,
          EdgeActiveness.Neither)
```

```
    }
})
```

在分区内，根据 activeFraction 的大小选择是进入 aggregateMessagesEdgeScan 还是 aggregateMessagesIndexScan 处理。aggregateMessagesEdgeScan 会顺序地扫描所有的边， 而 aggregateMessagesIndexScan 会先过滤源顶点索引，然后在扫描。我们重点去分析 aggregateMessagesEdgeScan。

```
def aggregateMessagesEdgeScan[A: ClassTag](
     sendMsg: EdgeContext[VD, ED, A] => Unit,
     mergeMsg: (A, A) => A,
     tripletFields: TripletFields,
     activeness: EdgeActiveness): Iterator[(VertexId, A)] = {
  var ctx = new AggregatingEdgeContext[VD, ED, A](mergeMsg, aggregates, bitset)
  var i = 0
  while (i < size) {
    val localSrcId = localSrcIds(i)
    val srcId = local2global(localSrcId)
    val localDstId = localDstIds(i)
    val dstId = local2global(localDstId)
    val srcAttr = if (tripletFields.useSrc) vertexAttrs(localSrcId) else
null.asInstanceOf[VD]
    val dstAttr = if (tripletFields.useDst) vertexAttrs(localDstId) else
null.asInstanceOf[VD]
    ctx.set(srcId, dstId, localSrcId, localDstId, srcAttr, dstAttr, data(i))
    sendMsg(ctx)
    i += 1
  }
}
```

该方法由两步组成，分别是获得顶点相关信息，以及发送消息。

- 获取顶点相关信息

在前文介绍 edge partition 时，我们知道它包含 localSrcIds,localDstIds, data, index, global2local, local2global, vertexAttrs 这几个重要的数据结构。其中 localSrcIds,localDstIds 分别表示源顶点、目的顶点在当前分区中的索引。 所以我们可以遍历 localSrcIds,根据其下标去 localSrcIds 中拿到 srcId 在全局 local2global 中的索引，最后拿到 srcId。通过 vertexAttrs 拿到顶点属性。通过 data 拿到边属性。

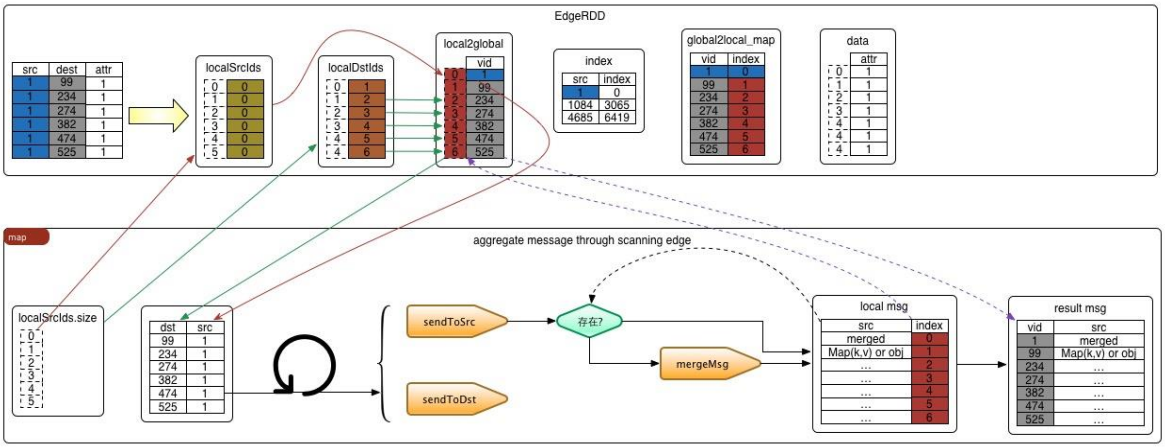
- 发送消息

发消息前会根据接口中定义的 tripletFields，拿到发消息的方向。发消息的过程就是遍历到一条边，向 localSrcIds/localDstIds 中添加数据，如果 localSrcIds/localDstIds 中已经存在该数据，则执行合并函数 mergeMsg。

```
override def sendToSrc(msg: A) {
  send(_localSrcId, msg)
}
override def sendToDst(msg: A) {
  send(_localDstId, msg)
}
@inline private def send(localId: Int, msg: A) {
  if (bitset.get(localId)) {
    aggregates(localId) = mergeMsg(aggregates(localId), msg)
  } else {
    aggregates(localId) = msg
    bitset.set(localId)
  }
}
```

每个点之间在发消息的时候是独立的，即：点单纯根据方向，向以相邻点的以 localId 为下标的数组中插数据，互相独立，可以并行运行。Map 阶段最后返回消息 RDD messages: RDD[(VertexId, VD2)]

Map 阶段的执行流程如下例所示：



2.4.7.1.2.2 Reduce 阶段

Reduce 阶段的实现就是调用下面的代码

```
vertices.aggregateUsingIndex(preAgg, mergeMsg)
override def aggregateUsingIndex[VD2: ClassTag](
                                        messages: RDD[(VertexId, VD2)],
reduceFunc: (VD2, VD2) => VD2): VertexRDD[VD2] = {
  val shuffled = messages.partitionBy(this.partitioner.get)
  val parts = partitionsRDD.zipPartitions(shuffled, true) { (thisIter, msgIter)
=>
    thisIter.map(_.aggregateUsingIndex(msgIter, reduceFunc))
  }
  this.withPartitionsRDD[VD2](parts)
}
```

上面的代码通过两步实现。

- 1 对 `messages` 重新分区，分区器使用 `VertexRDD` 的 `partitioner`。然后使用 `zipPartitions` 合并两个分区。
- 2 对等合并 `attr`，聚合函数使用传入的 `mergeMsg` 函数

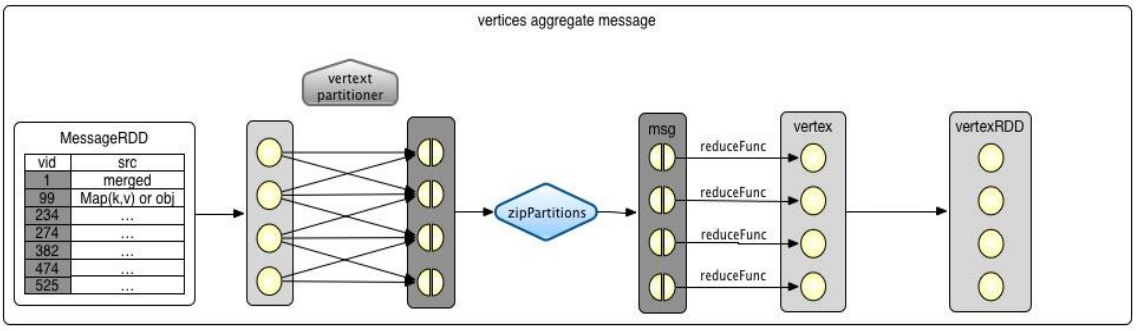
```
def aggregateUsingIndex[VD2: ClassTag](
              iter: Iterator[Product2[VertexId, VD2]],
               reduceFunc: (VD2, VD2) => VD2): Self[VD2] = {
  val newMask = new BitSet(self.capacity)
  val newValues = new Array[VD2](self.capacity)
```

```
  iter.foreach { product =>
    val vid = product._1
    val vdata = product._2
    val pos = self.index.getPos(vid)
    if (pos >= 0) {
      if (newMask.get(pos)) {
        newValues(pos) = reduceFunc(newValues(pos), vdata)
      } else { // otherwise just store the new value
        newMask.set(pos)
        newValues(pos) = vdata
      }
    }
  }
  this.withValues(newValues).withMask(newMask)
}
```

根据传参，我们知道上面的代码迭代的是 messagePartition，并不是每个节点都会收到消息，所以 messagePartition 集合最小，迭代速度会快。

这段代码表示，我们根据 vetexId 从 index 中取到其下标 pos,再根据下标，从 values 中取到 attr，存在 attr 就用 mergeMsg 合并 attr，不存在就直接赋值。

Reduce 阶段的过程如下图所示：



2.4.7.1.3 举例

下面的例子计算比用户年龄大的追随者（即 followers）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.aggregateMessages[(Int,
Double)](
  triplet => { // Map Function
    if (triplet.srcAttr > triplet.dstAttr) {
      // Send message to destination vertex containing counter and age
      triplet.sendToDst(1, triplet.srcAttr)
    }
  },
  // 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(_))
```

2.4.7.2 collectNeighbors

该方法的作用是收集每个顶点的邻居顶点的顶点 id 和顶点属性。

```
def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[Array[(VertexId,
VD)]] = {
  val nbrs = edgeDirection match {
    case EdgeDirection.Either =>
      graph.aggregateMessages[Array[(VertexId, VD)]](
        ctx => {
          ctx.sendToSrc(Array((ctx.dstId, ctx.dstAttr)))
          ctx.sendToDst(Array((ctx.srcId, ctx.srcAttr)))
        },
        (a, b) => a ++ b, TripletFields.All)
    case EdgeDirection.In =>
      graph.aggregateMessages[Array[(VertexId, VD)]](
        ctx => ctx.sendToDst(Array((ctx.srcId, ctx.srcAttr))),
```

```
        (a, b) => a ++ b, TripletFields.Src)
    case EdgeDirection.Out =>
      graph.aggregateMessages[Array[(VertexId, VD)]](
        ctx => ctx.sendToSrc(Array((ctx.dstId, ctx.dstAttr))),
        (a, b) => a ++ b, TripletFields.Dst)
    case EdgeDirection.Both =>
      throw new SparkException("collectEdges does not support
EdgeDirection.Both. Use" +
        "EdgeDirection.Either instead.")
  }
  graph.vertices.leftJoin(nbrs) { (vid, vdata, nbrsOpt) =>
    nbrsOpt.getOrElse(Array.empty[(VertexId, VD)])
  }
}
```

从上面的代码中，第一步是根据 EdgeDirection 来确定调用哪个 aggregateMessages 实现聚合操作。我们用满足条件 EdgeDirection.Either 的情况来说明。可以看到 aggregateMessages 的方式消息的函数为：

```
ctx => {
  ctx.sendToSrc(Array((ctx.dstId, ctx.dstAttr)))
  ctx.sendToDst(Array((ctx.srcId, ctx.srcAttr)))
},
```

这个函数在处理每条边时都会同时向源顶点和目的顶点发送消息，消息内容分别为（目的顶点 id，目的顶点属性）、（源顶点 id，源顶点属性）。为什么会这样处理呢？ 我们知道，每条边都由两个顶点组成，对于这个边，我需要向源顶点发送目的顶点的信息来记录它们之间的邻居关系，同理向目的顶点发送源顶点的信息来记录它们之间的邻居关系。

Merge 函数是一个集合合并操作，它合并同同一个顶点对应的所有目的顶点的信息。如下所示：

```
(a, b) => a ++ b
```

通过 aggregateMessages 获得包含邻居关系信息的 VertexRDD 后，把它和现有的 vertices 作 join 操作，得到每个顶点的邻居消息。

2.4.7.3 collectNeighborIds

该方法的作用是收集每个顶点的邻居顶点的顶点 id。它的实现和 collectNeighbors 非常相同。

```
def collectNeighborIds(edgeDirection: EdgeDirection):
VertexRDD[Array[VertexId]] = {
  val nbrs =
    if (edgeDirection == EdgeDirection.Either) {
      graph.aggregateMessages[Array[VertexId]](
        ctx => { ctx.sendToSrc(Array(ctx.dstId));
ctx.sendToDst(Array(ctx.srcId)) },
        _ ++ _, TripletFields.None)
    } else if (edgeDirection == EdgeDirection.Out) {
      graph.aggregateMessages[Array[VertexId]](
        ctx => ctx.sendToSrc(Array(ctx.dstId)),
        _ ++ _, TripletFields.None)
    } else if (edgeDirection == EdgeDirection.In) {
      graph.aggregateMessages[Array[VertexId]](
        ctx => ctx.sendToDst(Array(ctx.srcId)),
        _ ++ _, TripletFields.None)
    } else {
      throw new SparkException("It doesn't make sense to collect neighbor ids
without a " +
        "direction. (EdgeDirection.Both is not supported; use
EdgeDirection.Either instead.)")
    }
  graph.vertices.leftZipJoin(nbrs) { (vid, vdata, nbrsOpt) =>
    nbrsOpt.getOrElse(Array.empty[VertexId])
  }
}
```

和 collectNeighbors 的实现不同的是，aggregateMessages 函数中的 sendMsg 函数只发送顶点 Id 到源顶点和目的顶点。其它的实现基本一致。

```
ctx => { ctx.sendToSrc(Array(ctx.dstId)); ctx.sendToDst(Array(ctx.srcId)) }
```

2.4.8 缓存操作

在 Spark 中，RDD 默认是不缓存的。为了避免重复计算，当需要多次利用它们时，我们必须显示地缓存它们。GraphX 中的图也有相同的方式。当利用到图多次时，确保首先访问 Graph.cache()方法。

在迭代计算中，为了获得最佳的性能，不缓存可能是必须的。默认情况下，缓存的 RDD 和图会一直保留在内存中直到因为内存压力迫使它们以 LRU 的顺序删除。对于迭代计算，先前的迭代的中间结果将填充到缓存 中。虽然它们最终会被删除，但是保存在内存中的不需要的数据将会减慢垃圾回收。只有中间结果不需要，不缓存它们是更高效的。然而，因为图是由多个 RDD 组成的，正确的不持久化它们是困难的。对于迭代计算，我们建议使用 Pregel API，它可以正确的不持久化中间结果。

GraphX 中的缓存操作有 cache,persist,unpersist 和 unpersistVertices。它们的接口分别是：

```
def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD, ED]
def cache(): Graph[VD, ED]
def unpersist(blocking: Boolean = true): Graph[VD, ED]
def unpersistVertices(blocking: Boolean = true): Graph[VD, ED]
```

2.5 Pregel API

图本身是递归数据结构，顶点的属性依赖于它们邻居的属性，这些邻居的属性又依赖于自己邻居的属性。所以许多重要的图算法都是迭代的重新计算每个顶点的属性，直到满足某个确定的条件。 一系列的图并发(graph-parallel)抽象已经被提出来用来表达这些迭代算法。GraphX 公开了一个类似 Pregel 的操作，它是广泛使用的 Pregel 和 GraphLab 抽象的一个融合。

GraphX 中实现的这个更高级的 Pregel 操作是一个约束到图拓扑的批量同步（bulk-synchronous）并行消息抽象。Pregel 操作者执行一系列的超步（super

steps），在这些步骤中，顶点从 之前的超步中接收进入(inbound)消息的总和，为顶点属性计算一个新的值，然后在以后的超步中发送消息到邻居顶点。不像 Pregel 而更像 GraphLab，消息通过边 triplet 的一个函数被并行计算， 消息的计算既会访问源顶点特征也会访问目的顶点特征。在超步中，没有收到消息的顶点会被跳过。当没有消息遗留时，Pregel 操作停止迭代并返回最终的图。

注意，与标准的 Pregel 实现不同的是，GraphX 中的顶点仅仅能发送信息给邻居顶点，并且可以利用用户自定义的消息函数并行地构造消息。这些限制允许对 GraphX 进行额外的优化。

下面的代码是 pregel 的具体实现。

```
def apply[VD: ClassTag, ED: ClassTag, A: ClassTag]
  (graph: Graph[VD, ED],
   initialMsg: A,
   maxIterations: Int = Int.MaxValue,
   activeDirection: EdgeDirection = EdgeDirection.Either)
  (vprog: (VertexId, VD, A) => VD,
   sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
   mergeMsg: (A, A) => A)
  : Graph[VD, ED] =
{
  var g = graph.mapVertices((vid, vdata) => vprog(vid, vdata,
initialMsg)).cache()
  // 计算消息
  var messages = g.mapReduceTriplets(sendMsg, mergeMsg)
  var activeMessages = messages.count()
  // 迭代
  var prevG: Graph[VD, ED] = null
  var i = 0
  while (activeMessages > 0 && i < maxIterations) {
    // 接收消息并更新顶点
    prevG = g
    g = g.joinVertices(messages)(vprog).cache()
    val oldMessages = messages
    // 发送新消息
    messages = g.mapReduceTriplets(
      sendMsg, mergeMsg, Some((oldMessages, activeDirection))).cache()
    activeMessages = messages.count()
```

```
    i += 1
  }
  g
}
```

2.5.1 1 pregel 计算模型

Pregel 计算模型中有三个重要的函数，分别是 vertexProgram、sendMessage 和 messageCombiner。

- vertexProgram：用户定义的顶点运行程序。它作用于每一个顶点，负责接收进来的信息，并计算新的顶点值。
- sendMsg：发送消息
- mergeMsg：合并消息

我们具体分析它的实现。根据代码可以知道，这个实现是一个迭代的过程。在开始迭代之前，先完成一些初始化操作：

```
var g = graph.mapVertices((vid, vdata) => vprog(vid, vdata,
initialMsg)).cache()
// 计算消息
var messages = g.mapReduceTriplets(sendMsg, mergeMsg)
var activeMessages = messages.count()
```

程序首先用 vprog 函数处理图中所有的顶点，生成新的图。然后用生成的图调用聚合操作（mapReduceTriplets，实际的实现是我们前面章节讲到的 aggregateMessagesWithActiveSet 函数）获取聚合后的消息。 activeMessages 指 messages 这个 VertexRDD 中的顶点数。

下面就开始迭代操作了。在迭代内部，分为二步。

- 1 接收消息，并更新顶点

```
g = g.joinVertices(messages)(vprog).cache()
//joinVertices 的定义
def joinVertices[U: ClassTag](table: RDD[(VertexId, U)])(mapFunc: (VertexId,
VD, U) => VD)
: Graph[VD, ED] = {
  val uf = (id: VertexId, data: VD, o: Option[U]) => {
    o match {
      case Some(u) => mapFunc(id, data, u)
      case None => data
    }
  }
  graph.outerJoinVertices(table)(uf)
}
```

这一步实际上是使用 outerJoinVertices 来更新顶点属性。

outerJoinVertices 在关联操作中有详细介绍。

- 2 发送新消息

```
messages = g.mapReduceTriplets(
      sendMsg, mergeMsg, Some((oldMessages, activeDirection))).cache()
```

注意，在上面的代码中，mapReduceTriplets 多了一个参数

Some((oldMessages, activeDirection))。这个参数的作用是：它使我们在发送新的消息时，会忽略掉那些两端都没有接收到消息的边，减少计算量。

2.5.2 pregel 实现最短路径

```
import org.apache.spark.graphx._
import org.apache.spark.graphx.util.GraphGenerators
val graph: Graph[Long, Double] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapEdges(e =>
e.attr.toDouble)
val sourceId: VertexId = 42 // The ultimate source
// 初始化图
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"))
```

上面的例子中，`Vertex Program` 函数定义如下：

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

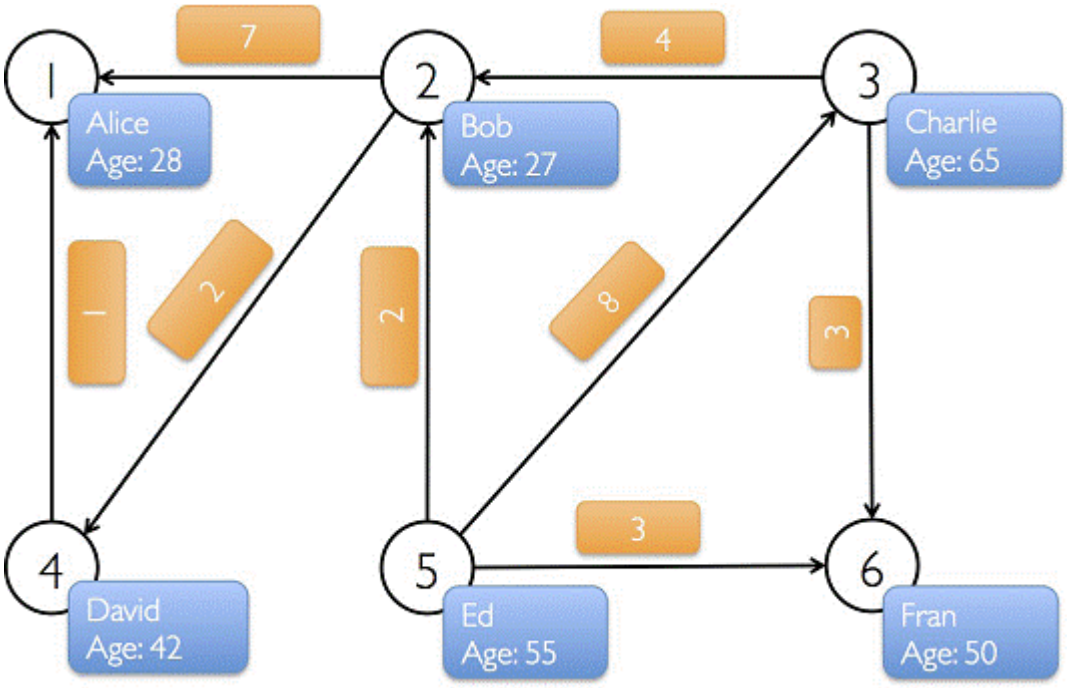
这个函数的定义显而易见，当两个消息来的时候，取它们当中路径的最小值。同理 `Merge Message` 函数也是同样的含义。

`Send Message` 函数中，会首先比较 `triplet.srcAttr + triplet.attr` 和 `triplet.dstAttr`，即比较加上边的属性后，这个值是否小于目的节点的属性，如果小于，则发送消息到目的顶点。

2.6 GraphX 实例

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。



程序代码如下：

```
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)),
```

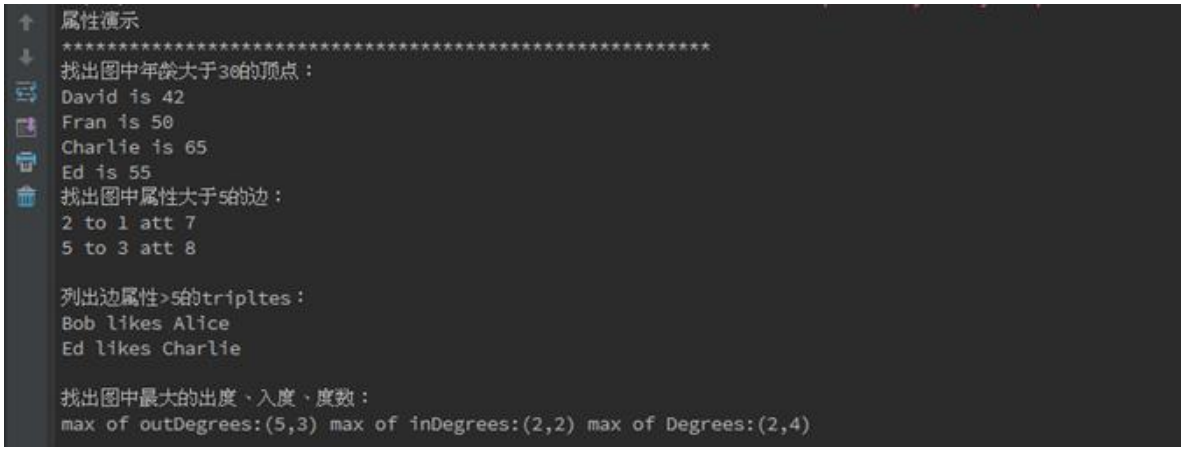
```
      (4L, ("David", 42)),
      (5L, ("Ed", 55)),
      (6L, ("Fran", 50))
    )
    //边的数据类型 ED:Int
    val edgeArray = Array(
      Edge(2L, 1L, 7),
      Edge(2L, 4L, 2),
      Edge(3L, 2L, 4),
      Edge(3L, 6L, 3),
      Edge(4L, 1L, 1),
      Edge(5L, 2L, 2),
      Edge(5L, 3L, 8),
      Edge(5L, 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()
  }
}
```



运行结果如下：

```
************************************************************
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)
```

第3章 图算法

3.1 PageRank 排名算法

3.1.1 算法概述

PageRank,即**网页排名**，又称**网页级别**、**Google 左侧排名**或**佩奇排名**。

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**，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另网站排名获得提升，从而提高搜索结果的相关性和质量。

3.1.2 从入链数量到 PageRank

PageRank 的计算基于以下两个基本假设：

λ 数量假设：在 Web 图模型中，如果一个页面节点接收到的其他网页指向的入链数量越多，那么这个页面越重要。

λ 质量假设：指向页面 A 的入链质量不同，质量高的页面会通过链接向其他页面传递更多的权重。所以越是质量高的页面指向页面 A，则页面 A 越重要。

利用以上两个假设，PageRank 算法刚开始赋予每个网页相同的重要性得分，通过迭代递归计算来更新每个页面节点的 PageRank 得分，直到得分稳定为止。 PageRank 计算得出的结果是网页的重要性评价，这和用户输入的查询是没有任何关系的，即算法是主题无关的。

3.1.3 PageRank 算法原理

PageRank 的计算充分利用了两个假设：数量假设和质量假设。步骤如下：

1）在初始阶段：网页通过链接关系构建起 Web 图，每个页面设置相同的 PageRank 值，通过若干轮的计算，会得到每个页面所获得的最终 PageRank 值。随着每一轮的计算进行，网页当前的 PageRank 值会不断得到更新。

2）在一轮中更新页面 PageRank 得分的计算方法：在一轮更新页面 PageRank 得分的计算中，每个页面将其当前的 PageRank 值平均分配到本页面包含的出链上，这样每个链接即获得了相应的权值。而每个页面将所有指向本页面的入链所传入的权值求和，即可得到新的 PageRank 得分。当每个页面都获得了更新后的 PageRank 值，就完成了一轮 PageRank 计算。

3.1.3.1 基本思想：

如果网页 T 存在一个指向网页 A 的连接，则表明 T 的所有者认为 A 比较重要，从而把 T 的一部分重要性得分赋予 A。这个重要性得分值为：PR（T）/L(T)

其中 PR（T）为 T 的 PageRank 值，L(T)为 T 的出链数

则 A 的 PageRank 值为一系列类似于 T 的页面重要性得分值的累加。

即一个页面的得票数由所有链向它的页面的重要性来决定，到一个页面的超链接相当于对该页投一票。一个页面的 PageRank 是由所有链向它的页面（链入页面）的重要性经过递归算法得到的。一个有较多链入的页面会有较高的等级，相反如果一个页面没有任何链入页面，那么它没有等级。

我们设向量 B 为第一、第二...第 N 个网页的网页排名

$$B = (b_1, b_2, \ldots, b_N)^T$$

矩阵 A 代表网页之间的权重输出关系，其中 a_{mn} 代表第 m 个网页向第 n 个网页的输出权重。

$$A=\begin{bmatrix} a_{11} & \cdots & a_{1n} & \cdots & a_{1M} \\ \cdots & & & & \cdots \\ a_{m1} & \cdots & a_{mn} & \cdots & a_{mM} \\ \cdots & & & & \cdots \\ a_{M1} & \cdots & a_{Mn} & \cdots & a_{MM} \end{bmatrix}$$

输出权重计算较为简单：假设 m 一共有 10 个出链，指向 n 的一共有 2 个，那么 m 向 n 输出的权重就为 2/10。

现在问题变为：A 是已知的，我们要通过计算得到 B。

假设 B_i 是第 i 次迭代的结果，那么

$$B_i = A \times B_{i-1}$$

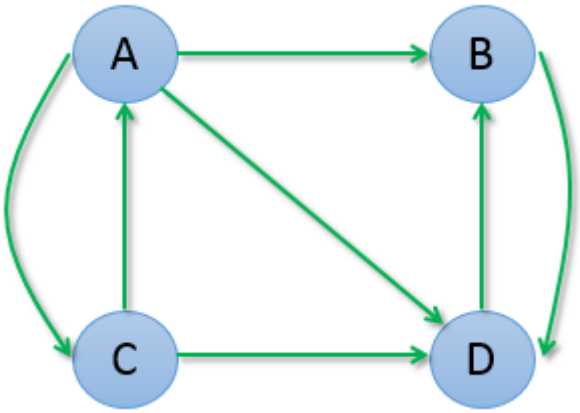
初始假设所有网页的排名都是 1/N （N 为网页总数量），即

$$B_0 = (\frac{1}{N}, \frac{1}{N}, \frac{1}{N}, \ldots, \frac{1}{N})^T$$

通过上述迭代计算，最终 B_i 会收敛，即 B_i 无限趋近于 B，此时 B = B × A。

3.1.3.2 具体示例

假设有网页 A、B、C、D，它们之间的链接关系如下图所示



计算 B_1 如下：

$$B_1 = A \times B_0 = \begin{bmatrix} 0 & 0 & 1/2 & 0 \\ 1/3 & 0 & 0 & 1/1 \\ 1/3 & 0 & 0 & 0 \\ 1/3 & 1/1 & 1/2 & 0 \end{bmatrix} \times \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} = \begin{bmatrix} 1/8 \\ 1/3 \\ 1/12 \\ 11/24 \end{bmatrix}$$

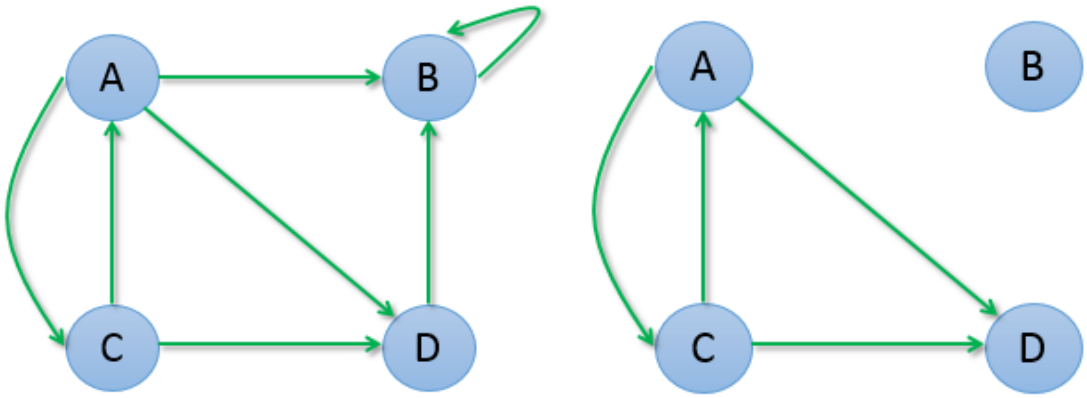
不断迭代，计算结果如下：

```
第 1 次迭代: 0.125, 0.333, 0.083, 0.458
第 2 次迭代: 0.042, 0.500, 0.042, 0.417
第 3 次迭代: 0.021, 0.431, 0.014, 0.535
第 4 次迭代: 0.007, 0.542, 0.007, 0.444
第 5 次迭代: 0.003, 0.447, 0.002, 0.547
第 6 次迭代: 0.001, 0.549, 0.001, 0.449
第 7 次迭代: 0.001, 0.449, 0.000, 0.550
第 8 次迭代: 0.000, 0.550, 0.000, 0.450
第 9 次迭代: 0.000, 0.450, 0.000, 0.550
第 10 次迭代: 0.000, 0.550, 0.000, 0.450
... ...
```

我们可以发现，A 和 C 的权重变为 0，而 B 和 D 的权重也趋于在 0.5 附近摆动。从图中也可以观察出：A 和 C 之间有互相链接，但它们又把权重输出给了 B 和 D，而 B 和 D 之间互相链接，并不向 A 或 C 输出任何权重，所以久而久之权重就都转移到 B 和 D 了。

3.1.3.3 PageRank 的改进

上面是最简单正常的情况，考虑一下两种特殊情况：



第一种情况是，B 存在导向自己的链接，迭代计算过程是：

```
第 1 次迭代: 0.125, 0.583, 0.083, 0.208
第 2 次迭代: 0.042, 0.833, 0.042, 0.083
```

```
第 3 次迭代: 0.021, 0.931, 0.014, 0.035
第 4 次迭代: 0.007, 0.972, 0.007, 0.014
第 5 次迭代: 0.003, 0.988, 0.002, 0.006
第 6 次迭代: 0.001, 0.995, 0.001, 0.002
第 7 次迭代: 0.001, 0.998, 0.000, 0.001
第 8 次迭代: 0.000, 0.999, 0.000, 0.000
第 9 次迭代: 0.000, 1.000, 0.000, 0.000
第 10 次迭代: 0.000, 1.000, 0.000, 0.000
... ...
```

我们发现最终 B 权重变为 1，其它所有网页的权重都变为了 0 =。=!

第二种情况是 B 是孤立于其它网页的，既没有入链也没有出链，迭代计算过程是：

```
第 1 次迭代: 0.125, 0.000, 0.125, 0.250
第 2 次迭代: 0.063, 0.000, 0.063, 0.125
第 3 次迭代: 0.031, 0.000, 0.031, 0.063
第 4 次迭代: 0.016, 0.000, 0.016, 0.031
第 5 次迭代: 0.008, 0.000, 0.008, 0.016
第 6 次迭代: 0.004, 0.000, 0.004, 0.008
第 7 次迭代: 0.002, 0.000, 0.002, 0.004
第 8 次迭代: 0.001, 0.000, 0.001, 0.002
第 9 次迭代: 0.000, 0.000, 0.000, 0.001
第 10 次迭代: 0.000, 0.000, 0.000, 0.000
... ...
```

我们发现所有网页权重都变为了 0 =。=!

出现这种情况是因为上面的数学模型出现了问题，该模型认为上网者从一个网页浏览下一个网页都是通过页面的超链接。想象一下正常的上网情景，其实我们在看完一个网页后，可能直接在浏览器输入一个网址，而不通过上一个页面的超链接。

我们假设每个网页被用户通过直接访问方式的概率是相等的，即 1/N，N 为网页总数，设矩阵 e 如下：

$$e=(\frac{1}{N},\frac{1}{N},\frac{1}{N},...,\frac{1}{N})^T$$

设用户通过页面超链接浏览下一网页的概率为 α，则直接访问的方式浏览下一个网页的概率为 1 - α，改进上一节的迭代公式为：

$$B_i=\alpha\times A\times B_{i-1}+(1-\alpha)\times e$$

通常情况下设 α 为 0.8，上一节”具体示例”的计算变为如下：

$$B_1 = \alpha \times A \times B_0 + (1-\alpha) \times e = 0.8 \times \begin{bmatrix} 0 & 0 & 1/2 & 0 \\ 1/3 & 0 & 0 & 1/1 \\ 1/3 & 0 & 0 & 0 \\ 1/3 & 1/1 & 1/2 & 0 \end{bmatrix} \times \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} + 0.2 \times \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}$$

迭代过程如下：

```
第 1 次迭代: 0.150, 0.317, 0.117, 0.417
第 2 次迭代: 0.097, 0.423, 0.090, 0.390
第 3 次迭代: 0.086, 0.388, 0.076, 0.450
第 4 次迭代: 0.080, 0.433, 0.073, 0.413
第 5 次迭代: 0.079, 0.402, 0.071, 0.447
第 6 次迭代: 0.079, 0.429, 0.071, 0.421
第 7 次迭代: 0.078, 0.408, 0.071, 0.443
第 8 次迭代: 0.078, 0.425, 0.071, 0.426
第 9 次迭代: 0.078, 0.412, 0.071, 0.439
第 10 次迭代: 0.078, 0.422, 0.071, 0.428
第 11 次迭代: 0.078, 0.414, 0.071, 0.437
第 12 次迭代: 0.078, 0.421, 0.071, 0.430
第 13 次迭代: 0.078, 0.415, 0.071, 0.436
第 14 次迭代: 0.078, 0.419, 0.071, 0.431
第 15 次迭代: 0.078, 0.416, 0.071, 0.435
第 16 次迭代: 0.078, 0.419, 0.071, 0.432
第 17 次迭代: 0.078, 0.416, 0.071, 0.434
第 18 次迭代: 0.078, 0.418, 0.071, 0.432
第 19 次迭代: 0.078, 0.417, 0.071, 0.434
第 20 次迭代: 0.078, 0.418, 0.071, 0.433
... ...
```

3.1.3.4 修正 PageRank 计算公式：

由于存在一些出链为 0，也就是那些不链接任何其他网页的网， 也称为孤立网页，使得很多网页能被访问到。因此需要对 PageRank 公式进行修正，即在简单公式的基础上增加了**阻尼系数（damping factor）**q， q 一般取值 q=0.85。

其意义是，在任意时刻，用户到达某页面后并继续向后浏览的概率。 1- q= 0.15 就是用户停止点击，随机跳到新 URL 的概率）的算法被用到了所有页面上，估算页面可能被上网者放入书签的概率。

最后，即所有这些被换算为一个百分比再乘上一个系数 q。由于下面的算法，没有页面的 PageRank 会是 0。所以，Google 通过数学系统给了每个页面一个最小值。

$$PR(A) = \left(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \cdots \right) q + 1 - q$$

这个公式就是.S Brin 和 L. Page 在《The Anatomy of a Large- scale Hypertextual Web Search Engine Computer Networks and ISDN Systems 》定义的公式。

所以一个页面的 PageRank 是由其他页面的 PageRank 计算得到。Google 不断的重复计算每个页面的 PageRank。如果给每个页面一个随机 PageRank 值（非 0），那么经过不断的重复计算，这些页面的 PR 值会趋向于正常和稳定。这就是搜索引擎使用它的原因。

首先求完整的公式：

Arvind Arasu 在《Junghoo Cho Hector Garcia - Molina, Andreas Paepcke, Sriram Raghavan. Searching the Web》 更加准确的表达为：

$$\mathrm{PageRank}(p_i) = \frac{1-q}{N} + q \sum_{p_j} \frac{\mathrm{PageRank}(p_j)}{L(p_j)}$$

$p_1, p_2, ..., p_N$是被研究的页面，$M(p_i)$是p_i链入页面的数量，$L(p_j)$是p_j链出页面的数量，而 *N* 是所有页面的数量。

PageRank 值是一个特殊矩阵中的特征向量。这个特征向量为：

$$\mathbf{R} = \begin{bmatrix} \mathrm{PageRank}(p_1) \\ \mathrm{PageRank}(p_2) \\ \vdots \\ \mathrm{PageRank}(p_N) \end{bmatrix}$$

R 是如下等式的一个解：

$$\mathbf{R} = \begin{bmatrix} (1-q)/N \\ (1-q)/N \\ \vdots \\ (1-q)/N \end{bmatrix} + q \begin{bmatrix} \ell(p_1,p_1) & \ell(p_1,p_2) & \cdots & \ell(p_1,p_N) \\ \ell(p_2,p_1) & \ddots & & \\ \vdots & & \ell(p_i,p_j) & \\ \ell(p_N,p_1) & & & \ell(p_N,p_N) \end{bmatrix} \mathbf{R}$$

如果网页 i 有指向网页 j 的一个链接，则

$$\sum_{i=1}^{N} \ell(p_i,p_j) = 1,$$

否则$\ell(p_i,p_j)=0$。

3.1.4 Spark GraphX 实现

```
import org.apache.spark.graphx.GraphLoader

// Load the edges as a graph
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")
// Run PageRank
val ranks = graph.pageRank(0.0001).vertices
// Join the ranks with the usernames
val users = sc.textFile("data/graphx/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"))
```

3.2 广度优先遍历(参考)

```
val graph = GraphLoader.edgeListFile(sc, "graphx/data/test_graph.txt")

val root: VertexId = 1
```

```
val initialGraph = graph.mapVertices((id, _) => if (id == root) 0.0 else
  Double.PositiveInfinity)

val vprog = { (id: VertexId, attr: Double, msg: Double) =>
math.min(attr,msg) }

val sendMessage = { (triplet: EdgeTriplet[Double, Int]) =>
  var iter:Iterator[(VertexId, Double)] = Iterator.empty
  val isSrcMarked = triplet.srcAttr != Double.PositiveInfinity
  val isDstMarked = triplet.dstAttr != Double.PositiveInfinity
  if(!(isSrcMarked && isDstMarked)){
    if(isSrcMarked){
      iter = Iterator((triplet.dstId,triplet.srcAttr+1))
    }else{
      iter = Iterator((triplet.srcId,triplet.dstAttr+1))
    }
  }
  iter
}

val reduceMessage = { (a: Double, b: Double) => math.min(a,b) }

val bfs = initialGraph.pregel(Double.PositiveInfinity, 20)(vprog, sendMessage,
reduceMessage)

println(bfs.vertices.collect.mkString("\n"))
```

3.3 单源最短路径(参考)

```
import scala.reflect.ClassTag

import org.apache.spark.graphx._

/**
  * Computes shortest paths to the given set of landmark vertices, returning a
graph where each
  * vertex attribute is a map containing the shortest-path distance to each
reachable landmark.
  */
```

```
object ShortestPaths {
  /** Stores a map from the vertex id of a landmark to the distance to that
landmark. */
  type SPMap = Map[VertexId, Int]

  private def makeMap(x: (VertexId, Int)*) = Map(x: _*)

  private def incrementMap(spmap: SPMap): SPMap = spmap.map { case (v, d) => v
-> (d + 1) }

  private def addMaps(spmap1: SPMap, spmap2: SPMap): SPMap =
    (spmap1.keySet ++ spmap2.keySet).map {
      k => k -> math.min(spmap1.getOrElse(k, Int.MaxValue), spmap2.getOrElse(k,
Int.MaxValue))
    }.toMap

  /**
    * Computes shortest paths to the given set of landmark vertices.
    *
    * @tparam ED the edge attribute type (not used in the computation)
    *
    * @param graph the graph for which to compute the shortest paths
    * @param landmarks the list of landmark vertex ids. Shortest paths will be
computed to each
    * landmark.
    *
    * @return a graph where each vertex attribute is a map containing the
shortest-path distance to
    * each reachable landmark vertex.
    */
  def run[VD, ED: ClassTag](graph: Graph[VD, ED], landmarks: Seq[VertexId]):
Graph[SPMap, ED] = {
    val spGraph = graph.mapVertices { (vid, attr) =>
      if (landmarks.contains(vid)) makeMap(vid -> 0) else makeMap()
    }

    val initialMessage = makeMap()

    def vertexProgram(id: VertexId, attr: SPMap, msg: SPMap): SPMap = {
      addMaps(attr, msg)
    }
```

```
    def sendMessage(edge: EdgeTriplet[SPMap, _]): Iterator[(VertexId, SPMap)] =
{
      val newAttr = incrementMap(edge.dstAttr)
      if (edge.srcAttr != addMaps(newAttr, edge.srcAttr)) Iterator((edge.srcId,
newAttr))
      else Iterator.empty
    }

    Pregel(spGraph, initialMessage)(vertexProgram, sendMessage, addMaps)
  }
}
```

3.4 连通图(参考)

```
import scala.reflect.ClassTag

import org.apache.spark.graphx._

/** Connected components algorithm. */
object ConnectedComponents {
  /**
    * Compute the connected component membership of each vertex and return a
graph with the vertex
    * value containing the lowest vertex id in the connected component
containing that vertex.
    *
    * @tparam VD the vertex attribute type (discarded in the computation)
    * @tparam ED the edge attribute type (preserved in the computation)
    * @param graph the graph for which to compute the connected components
    * @param maxIterations the maximum number of iterations to run for
    * @return a graph with vertex attributes containing the smallest vertex in
each
    *         connected component
    */
  def run[VD: ClassTag, ED: ClassTag](graph: Graph[VD, ED],
                                      maxIterations: Int): Graph[VertexId, ED] = {
    require(maxIterations > 0, s"Maximum of iterations must be greater than 0,"
+
```

```
      s" but got ${maxIterations}")

    val ccGraph = graph.mapVertices { case (vid, _) => vid }
    def sendMessage(edge: EdgeTriplet[VertexId, ED]): Iterator[(VertexId,
VertexId)] = {
      if (edge.srcAttr < edge.dstAttr) {
        Iterator((edge.dstId, edge.srcAttr))
      } else if (edge.srcAttr > edge.dstAttr) {
        Iterator((edge.srcId, edge.dstAttr))
      } else {
        Iterator.empty
      }
    }
    val initialMessage = Long.MaxValue
    val pregelGraph = Pregel(ccGraph, initialMessage,
      maxIterations, EdgeDirection.Either)(
      vprog = (id, attr, msg) => math.min(attr, msg),
      sendMsg = sendMessage,
      mergeMsg = (a, b) => math.min(a, b))
    ccGraph.unpersist()
    pregelGraph
  } // end of connectedComponents

  /**
    * Compute the connected component membership of each vertex and return a
graph with the vertex
    * value containing the lowest vertex id in the connected component
containing that vertex.
    *
    * @tparam VD the vertex attribute type (discarded in the computation)
    * @tparam ED the edge attribute type (preserved in the computation)
    * @param graph the graph for which to compute the connected components
    * @return a graph with vertex attributes containing the smallest vertex in
each
    *         connected component
    */
  def run[VD: ClassTag, ED: ClassTag](graph: Graph[VD, ED]): Graph[VertexId,
ED] = {
    run(graph, Int.MaxValue)
  }
}
```

3.5 三角计数(参考)

```
import scala.reflect.ClassTag

import org.apache.spark.graphx._

/**
 * Compute the number of triangles passing through each vertex.
 *
 * The algorithm is relatively straightforward and can be computed in three
steps:
 *
 * <ul>
 * <li> Compute the set of neighbors for each vertex</li>
 * <li> For each edge compute the intersection of the sets and send the count
to both vertices.</li>
 * <li> Compute the sum at each vertex and divide by two since each triangle
is counted twice.</li>
 * </ul>
 *
 * There are two implementations.  The default `TriangleCount.run`
implementation first removes
 * self cycles and canonicalizes the graph to ensure that the following
conditions hold:
 * <ul>
 * <li> There are no self edges</li>
 * <li> All edges are oriented src > dst</li>
 * <li> There are no duplicate edges</li>
 * </ul>
 * However, the canonicalization procedure is costly as it requires
repartitioning the graph.
 * If the input data is already in "canonical form" with self cycles removed
then the
 * `TriangleCount.runPreCanonicalized` should be used instead.
 *
 * {{{
 * val canonicalGraph = graph.mapEdges(e =>
1).removeSelfEdges().canonicalizeEdges()
 * val counts = TriangleCount.runPreCanonicalized(canonicalGraph).vertices
```

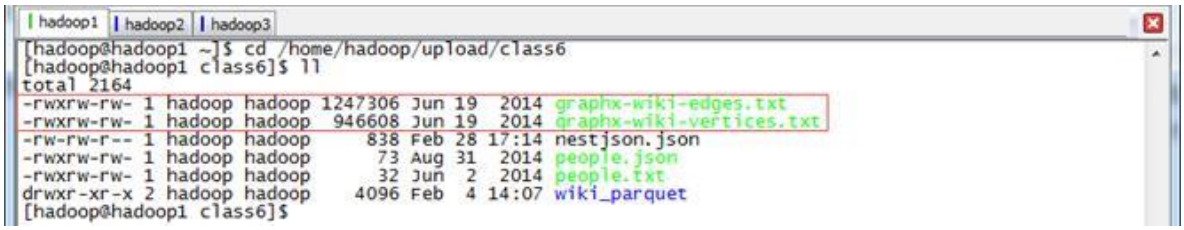
```
 * }}}
 *
 */
object TriangleCount {

  def run[VD: ClassTag, ED: ClassTag](graph: Graph[VD, ED]): Graph[Int, ED] = {
    // Transform the edge data something cheap to shuffle and then canonicalize
    val canonicalGraph = graph.mapEdges(e =>
true).removeSelfEdges().convertToCanonicalEdges()
    // Get the triangle counts
    val counters = runPreCanonicalized(canonicalGraph).vertices
    // Join them bath with the original graph
    graph.outerJoinVertices(counters) { (vid, _, optCounter: Option[Int]) =>
      optCounter.getOrElse(0)
    }
  }


  def runPreCanonicalized[VD: ClassTag, ED: ClassTag](graph: Graph[VD, ED]):
Graph[Int, ED] = {
    // Construct set representations of the neighborhoods
    val nbrSets: VertexRDD[VertexSet] =
      graph.collectNeighborIds(EdgeDirection.Either).mapValues { (vid, nbrs) =>
        val set = new VertexSet(nbrs.length)
        var i = 0
        while (i < nbrs.length) {
          // prevent self cycle
          if (nbrs(i) != vid) {
            set.add(nbrs(i))
          }
          i += 1
        }
        set
      }

    // join the sets with the graph
    val setGraph: Graph[VertexSet, ED] = graph.outerJoinVertices(nbrSets) {
      (vid, _, optSet) => optSet.getOrElse(null)
    }

    // Edge function computes intersection of smaller vertex with larger vertex
    def edgeFunc(ctx: EdgeContext[VertexSet, ED, Int]) {
```

```
      val (smallSet, largeSet) = if (ctx.srcAttr.size < ctx.dstAttr.size) {
        (ctx.srcAttr, ctx.dstAttr)
      } else {
        (ctx.dstAttr, ctx.srcAttr)
      }
      val iter = smallSet.iterator
      var counter: Int = 0
      while (iter.hasNext) {
        val vid = iter.next()
        if (vid != ctx.srcId && vid != ctx.dstId && largeSet.contains(vid)) {
          counter += 1
        }
      }
      ctx.sendToSrc(counter)
      ctx.sendToDst(counter)
    }

    // compute the intersection along edges
    val counters: VertexRDD[Int] = setGraph.aggregateMessages(edgeFunc, _ + _)
    // Merge counters with the graph and divide by two since each triangle is
counted twice
    graph.outerJoinVertices(counters) { (_, _, optCounter: Option[Int]) =>
      val dblCount = optCounter.getOrElse(0)
      // This algorithm double counts each triangle so the final count should
be even
      require(dblCount % 2 == 0, "Triangle count resulted in an invalid number
of triangles.")
      dblCount / 2
    }
  }
}
```

第4章 PageRank 实例

采用的数据是 wiki 数据中含有 Berkeley 标题的网页之间连接关系，数据为两个文件：graphx-wiki-vertices.txt 和 graphx-wiki-edges.txt，可以分别用于图计算的顶点和边。

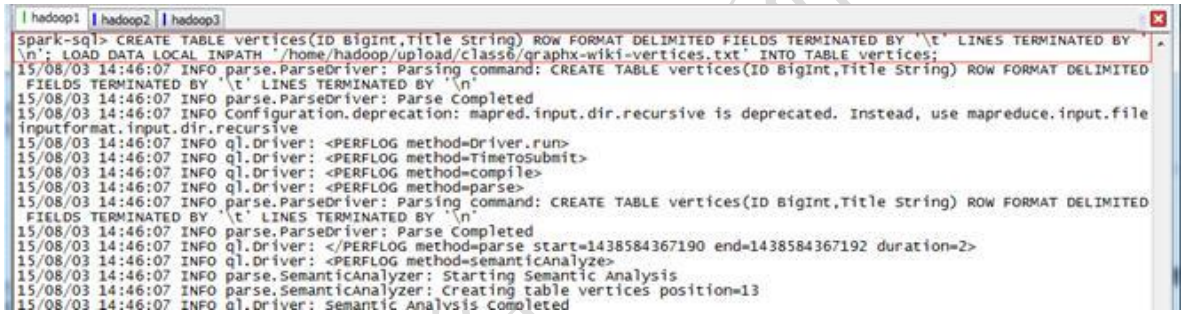
第一步 上传数据



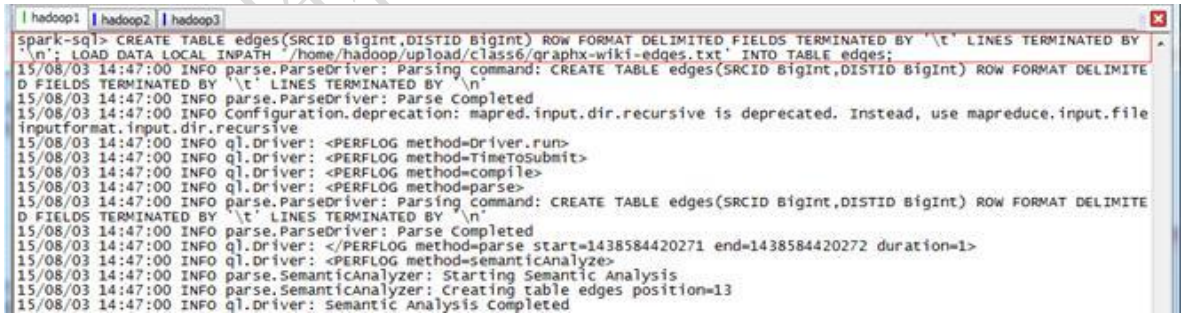
第二步　定义表并加载数据

创建 vertices 和 edges 两个表并加载数据：

spark-sql>show databases;
spark-sql>use hive;
spark-sql>CREATE TABLE vertices(ID BigInt,Title String) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' LINES TERMINATED BY '\n'; LOAD DATA LOCAL INPATH '/home/bigdata/hadoop/graphx-wiki-vertices.txt' INTO TABLE vertices;

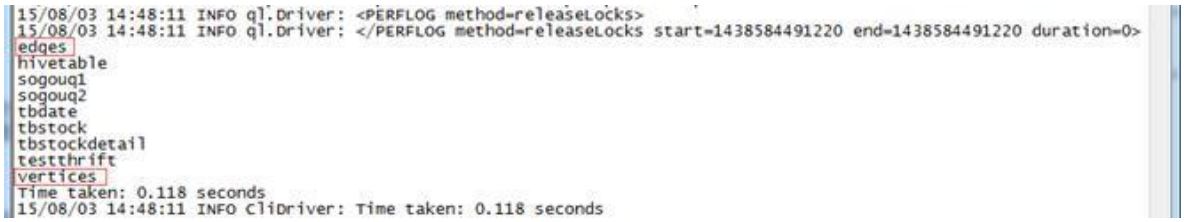


spark-sql>CREATE TABLE edges(SRCID BigInt,DISTID BigInt) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' LINES TERMINATED BY '\n'; LOAD DATA LOCAL INPATH '/home/bigdata/hadoop/graphx-wiki-edges.txt' INTO TABLE edges;



查看创建结果

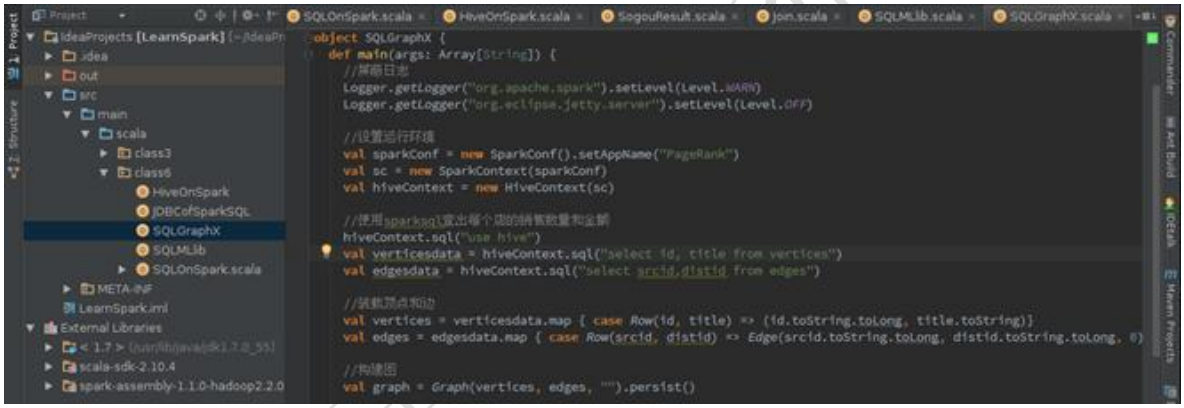
spark-sql>show tables;



4.1.1 实现代码

```
import org.apache.log4j.{Level, Logger}
import org.apache.spark.sql.hive.HiveContext
import org.apache.spark.{SparkContext, SparkConf}
import org.apache.spark.graphx._
import org.apache.spark.sql.catalyst.expressions.Row

object SQLGraphX {
    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 sparkConf = new SparkConf().setAppName("PageRank")
        val sc = new SparkContext(sparkConf)
        val hiveContext = new HiveContext(sc)

        //使用 sparksql 查出每个店的销售数量和金额
        hiveContext.sql("use hive")
        val verticesdata = hiveContext.sql("select id, title from vertices")
        val edgesdata = hiveContext.sql("select srcid,distid from edges")

        //装载顶点和边
        val vertices = verticesdata.map { case Row(id, title) =>
(id.toString.toLong, title.toString)}
        val edges = edgesdata.map { case Row(srcid, distid) =>
Edge(srcid.toString.toLong, distid.toString.toLong, 0)}

        //构建图
        val graph = Graph(vertices, edges, "").persist()

        //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")
```

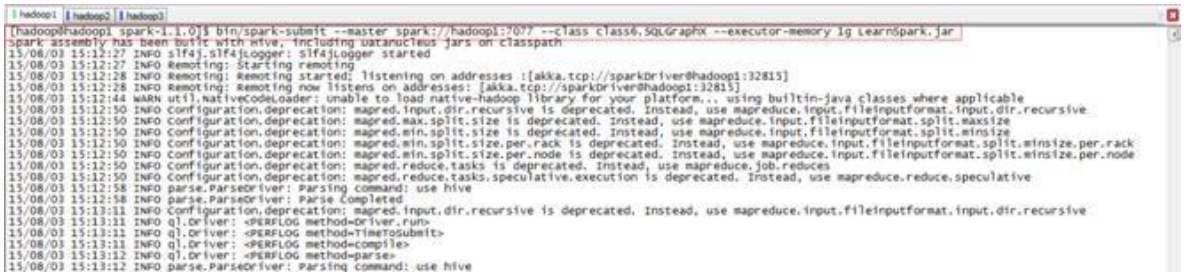
```
        println("*******************************************************")
        val prGraph = graph.pageRank(0.001).cache()

        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()
      }
}
```



4.1.2 运行查看结果

通过如下命令调用打包中的 SQLOnSpark 方法：

cd /app/hadoop/spark-2.1.1
bin/spark-submit --master spark://master01:7077 --class com.atguigu.SQLGraphX --executor-memory 1g SQLGraphX.jar



运行结果：

```
15/08/03 15:30:11 INFO parse.ParseDriver: Parsing command: select id, title from vertices
15/08/03 15:30:11 INFO parse.ParseDriver: Parse Completed
15/08/03 15:30:11 INFO parse.ParseDriver: Parsing command: select srcid,distid from edges
15/08/03 15:30:11 INFO parse.ParseDriver: Parse Completed
15/08/03 15:30:12 INFO Configuration.deprecation: mapred.map.tasks is deprecated. Instead, use mapreduce.job.maps
15/08/03 15:30:15 INFO mapred.FileInputFormat: Total input paths to process : 1
15/08/03 15:30:16 INFO mapred.FileInputFormat: Total input paths to process : 1
*************************************************************
PageRank??,?????????
*************************************************************
University of California, Berkeley: 1321.1117543121227
Berkeley, California: 664.8841977233989
Uc berkeley: 162.5013274339786
Berkeley Software Distribution: 90.47860388486127
Lawrence Berkeley National Laboratory: 81.90404939642022
George Berkeley: 81.85226118458043
Busby Berkeley: 47.87199821801991
Berkeley Hills: 44.76406979519929
Xander Berkeley: 30.32407534728813
Berkeley County, South Carolina: 28.908336483710315
```