GraphX 介绍

目 录

Spark GraphX 介绍

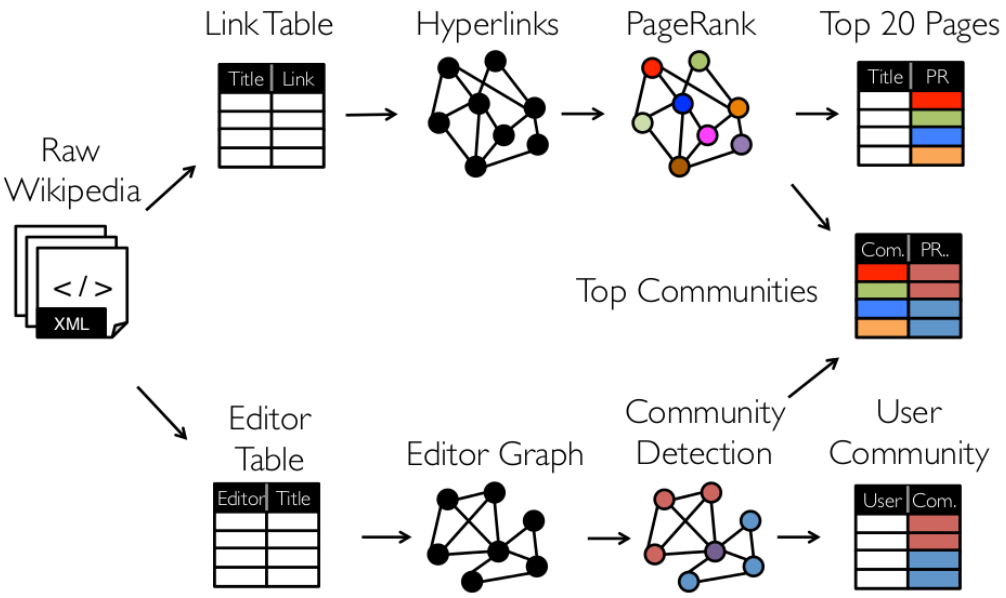
1 GraphX 介绍

1.1 GraphX 应用背景

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

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

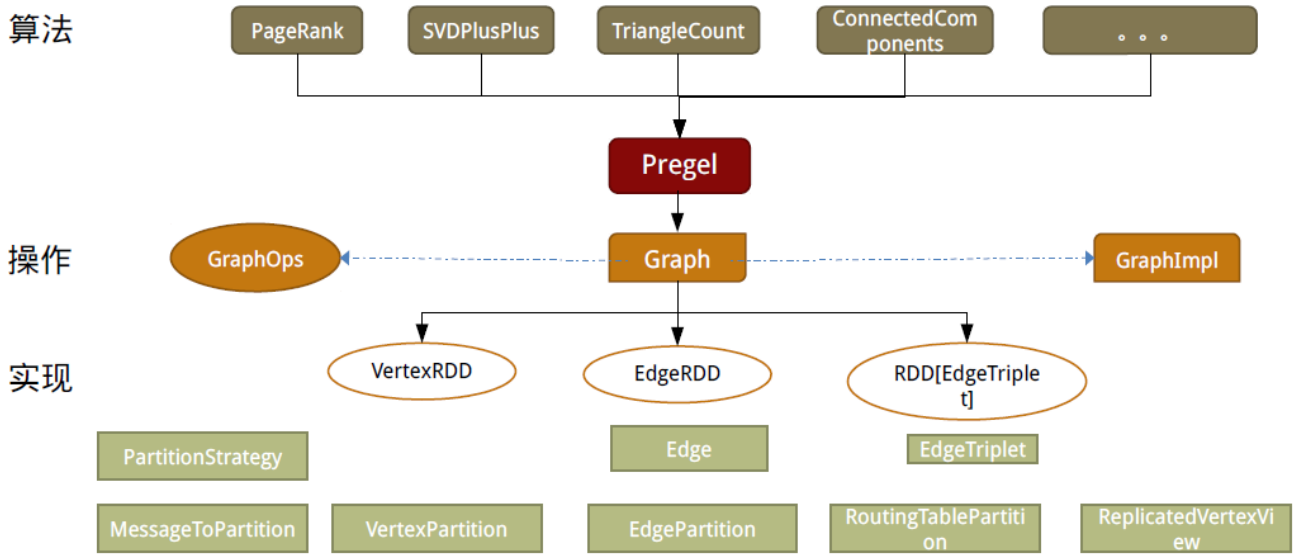
图的分布式或者并行处理其实是把图拆分成很多的子图，然后分别对这些子图进行计算，计算的时候可以分别迭代进行分阶段的计算，即对图进行并行计算。下面我们看一下图计算的简单示例：



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

1.2 GraphX 的框架

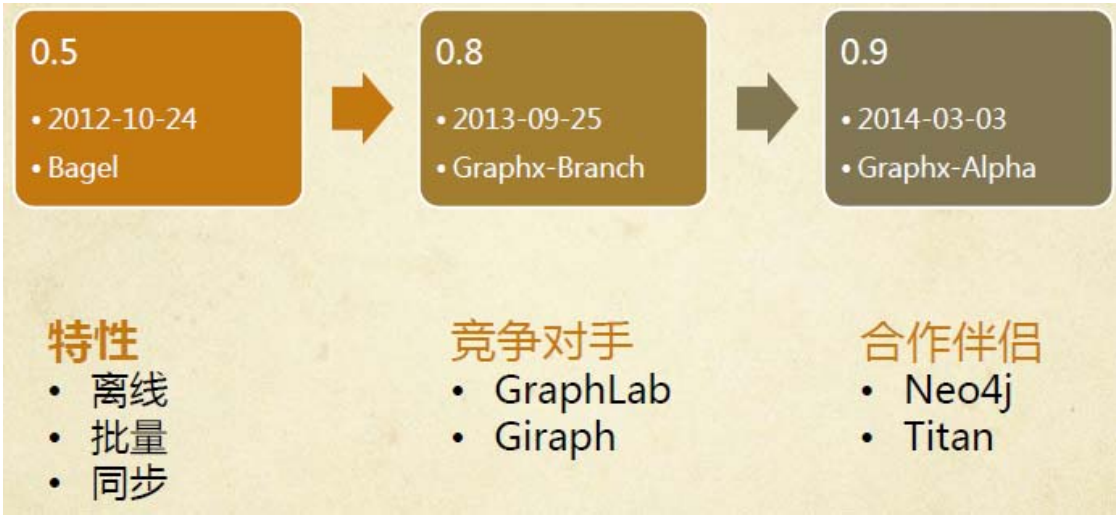
设计 GraphX 时，点分割和 GAS 都已成熟，在设计和编码中针对它们进行了优化，并在功能和性能之间寻找最佳的平衡点。如同 Spark 本身，每个子模块都有一个核心抽象。GraphX 的核心抽象是 Resilient Distributed Property Graph，一种点和边都带属性的有向多重图。它扩展了 Spark RDD 的抽象，有 Table 和 Graph 两种视图，而只需要一份物理存储。两种视图都有自己独有的操作符，从而获得了灵活操作和执行效率。



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

1.3 发展历程

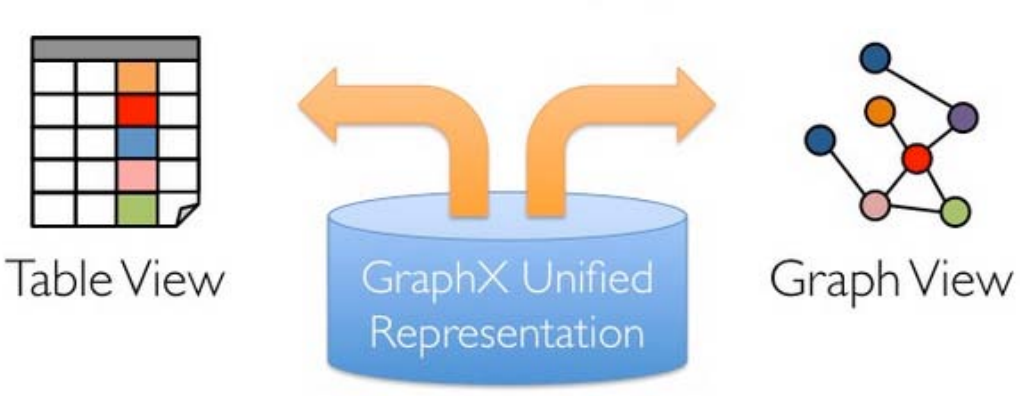
- 早在 0.5 版本，Spark 就带了一个小型的 Bagel 模块，提供了类似 Pregel 的功能。当然，这个版本还非常原始，性能和功能都比较弱，属于实验型产品。
- 到 0.8 版本时，鉴于业界对分布式图计算的需求日益见涨，Spark 开始独立一个分支 Graphx-Branch，作为独立的图计算模块，借鉴 GraphLab，开始设计开发 GraphX。
- 在 0.9 版本中，这个模块被正式集成到主干，虽然是 Alpha 版本，但已可以试用，小面包圈 Bagel 告别舞台。1.0 版本，GraphX 正式投入生产使用。



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

2 GraphX 实现分析

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

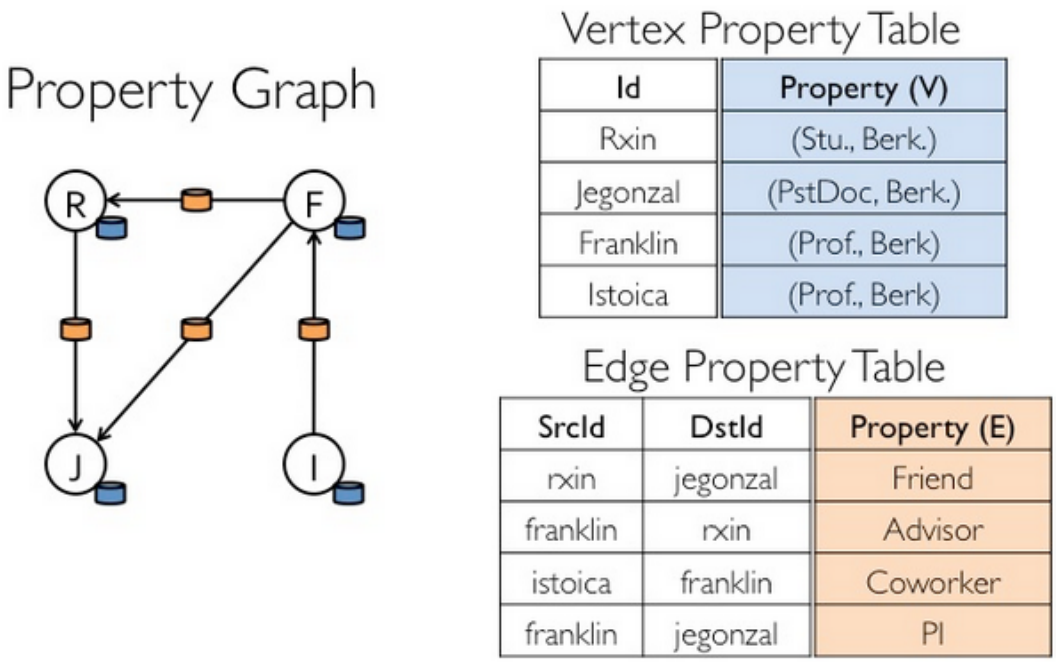


GraphX 的底层设计有以下几个关键点。

对 Graph 视图的所有操作，最终都会转换成其关联的 Table 视图的 RDD 操作来完成。这样对一个图的计算，最终在逻辑上，等价于一系列 RDD 的转换过程。因此，Graph 最终具备了 RDD 的 3 个关键特性：Immutable、Distributed 和 Fault-Tolerant，其中最关键的是 Immutable（不

变性）。逻辑上，所有图的转换和操作都产生了一个新图；物理上，GraphX 会有一定程度的不变顶点和边的复用优化，对用户透明。

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



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

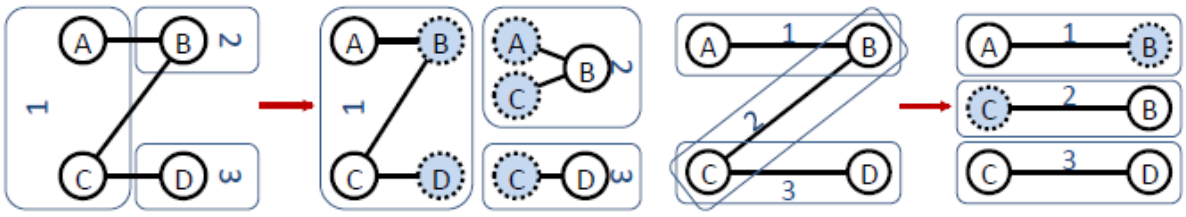
2.1 存储模式

2.1.1 图存储模式

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

- **边分割（Edge-Cut）**：每个顶点都存储一次，但有的边会被打断分到两台机器上。这样做的好处是节省存储空间；坏处是对图进行基于边的计算时，对于一条两个顶点被分到不同机器上的边来说，要跨机器通信传输数据，内网通信流量大。

- **点分割（Vertex-Cut）**：每条边只存储一次，都只会出现在一台机器上。邻居多的点会被复制到多台机器上，增加了存储开销，同时会引发数据同步问题。好处是可以大幅减少内网通信量。



(a) Edge-Cut　　(b) Vertex-Cut

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

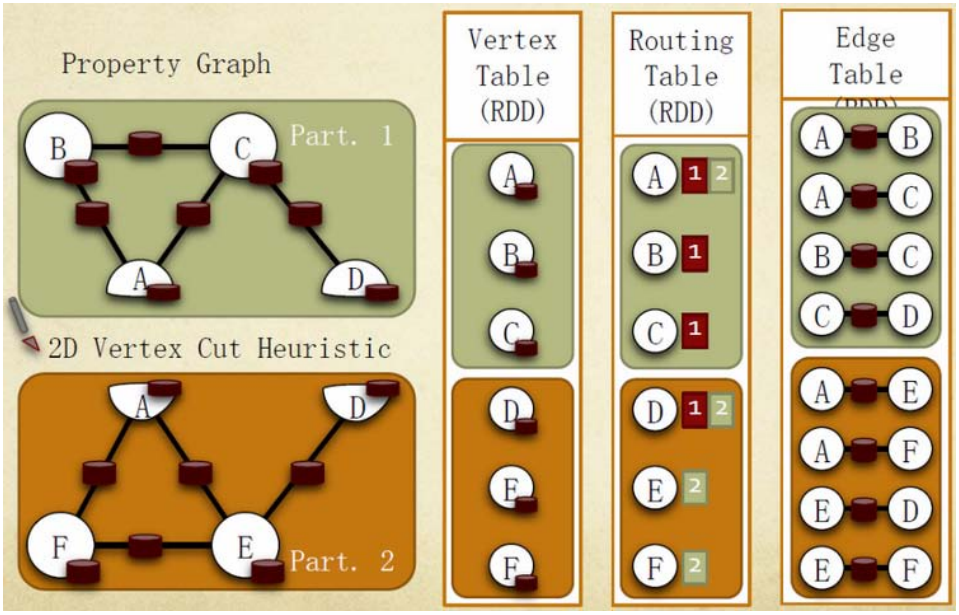
1. 磁盘价格下降，存储空间不再是问题，而内网的通信资源没有突破性进展，集群计算时内网带宽是宝贵的，时间比磁盘更珍贵。这点就类似于常见的空间换时间的策略。
2. 在当前的应用场景中，绝大多数网络都是“无尺度网络”，遵循幂律分布，不同点的邻居数量相差非常悬殊。而边分割会使那些多邻居的点所相连的边大多数被分到不同的机器上，这样的数据分布会使得内网带宽更加捉襟见肘，于是边分割存储方式被渐渐抛弃了。

2.1.2 GraphX 存储模式

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

- **VertexTable(id, data)**：id 为 Vertex id，data 为 Edge data
- **EdgeTable(pid, src, dst, data)**：pid 为 Partion id，src 为原定点 id，dst 为目的顶点 id
- **RoutingTable(id, pid)**：id 为 Vertex id，pid 为 Partion id

点分割存储实现如下图所示：

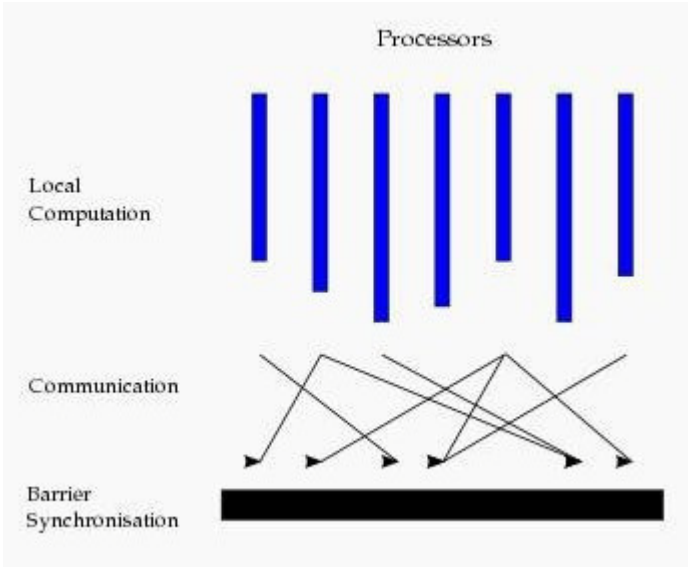


2.2 计算模式

2.2.1 图计算模式

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

Bulk Synchronous Parallell，即整体同步并行，它将计算分成一系列的超步（superstep）的迭代（iteration）。从纵向上看，它是一个串行模式，而从横向上看，它是一个并行的模式，每两个 superstep 之间设置一个栅栏（barrier），即整体同步点，确定所有并行的计算都完成后再启动下一轮 superstep。

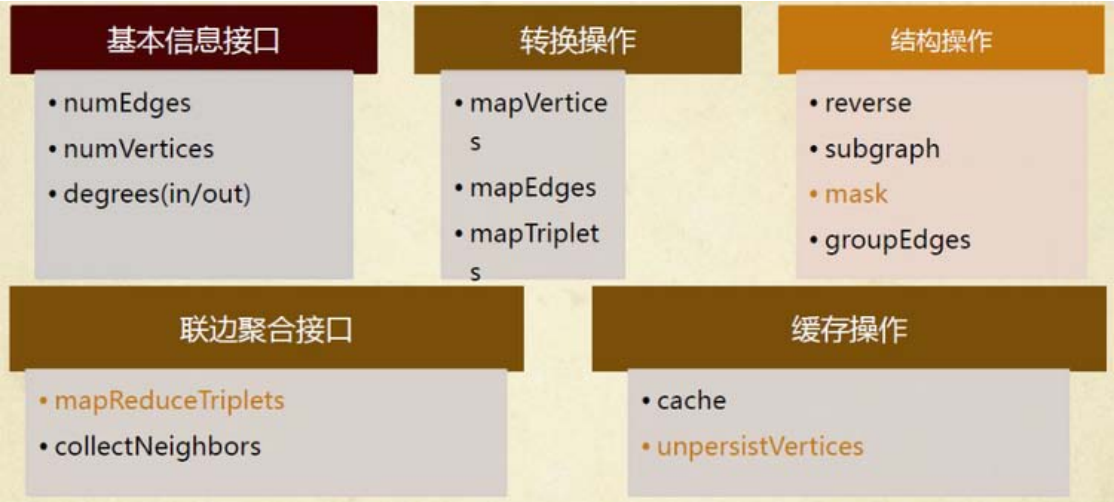


每一个超步（superstep）包含三部分内容：

1. **计算 compute**：每一个 processor 利用上一个 superstep 传过来的消息和本地的数据进行本地计算；
2. **消息传递**：每一个 processor 计算完毕后，将消息传递个与之关联的其它 processors
3. **整体同步点**：用于整体同步，确定所有的计算和消息传递都进行完毕后，进入下一个 superstep。

2.2.2 GraphX 计算模式

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



2.2.2.1 图的缓存

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

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

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

2.2.2.2 邻边聚合

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

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

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

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

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

1. **Caching for Iterative mrTriplets & Incremental Updates for Iterative mrTriplets**：在很多图分析算法中，不同点的收敛速度变化很大。在迭代后期，只有很少的点会有更新。因此，对于没有更新的点，下一次 mrTriplets 计算时 EdgeRDD 无需更新相应点值的本地缓存，大幅降低了通信开销。
2. **Indexing Active Edges**：没有更新的顶点在下一轮迭代时不需要向邻居重新发送消息。因此，mrTriplets 遍历边时，如果一条边的邻居点值在上一轮迭代时没有更新，则直接跳过，避免了大量无用的计算和通信。
3. **Join Elimination**：Triplet 是由一条边和其两个邻居点组成的三元组，操作 Triplet 的 map 函数常常只需访问其两个邻居点值中的一个。例如，在 PageRank 计算中，一个点值的更新只与其源顶点的值有关，而与其所指向的目的顶点的值无关。那么在 mrTriplets 计算中，就不需要 VertexRDD 和 EdgeRDD 的 3-way join，而只需要 2-way join。

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

2.2.2.3 进化的 Pregel 模式

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

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

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

常见的代码模板如下所示：

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

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

2.2.2.4 图算法工具包

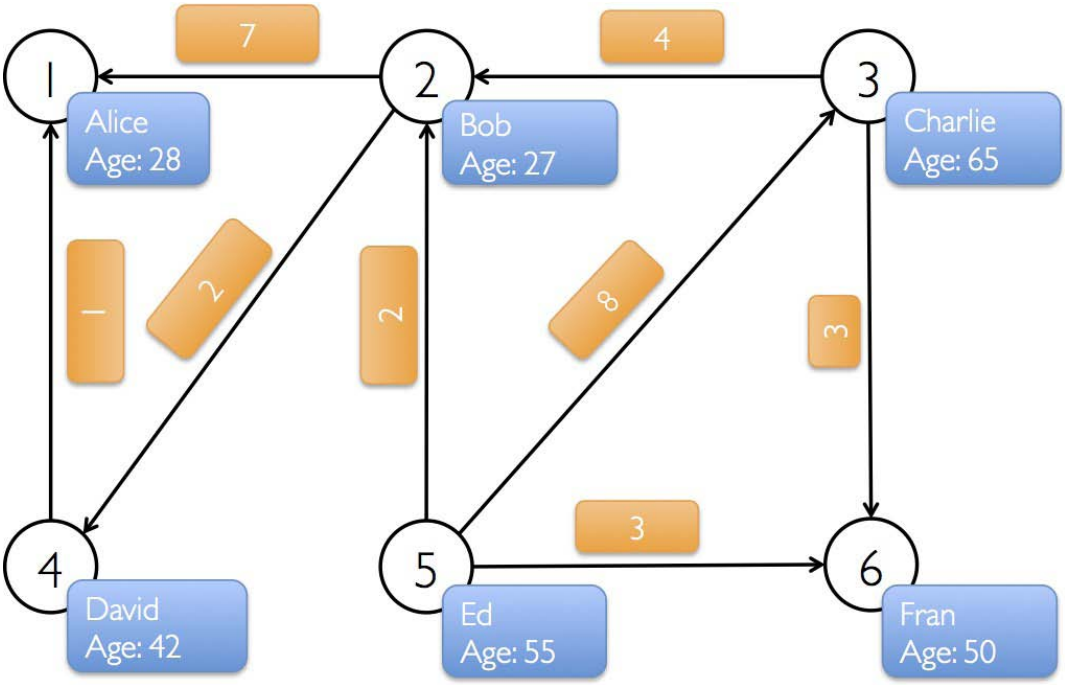
GraphX 也提供了一套图算法工具包，方便用户对图进行分析。目前最新版本已支持 PageRank、数三角形、最大连通图和最短路径等 6 种经典的图算法。这些算法的代码实现，目的和重点在于通用性。如果要获得最佳性能，可以参考其实现进行修改和扩展满足业务需求。另外，研读这些代码，也是理解 GraphX 编程最佳实践的好方法。

3 GraphX 实例

3.1 图例演示

3.1.1 例子介绍

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



3.1.2 程序代码

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

object GraphXExample {
  def main(args: Array[String]) {
    //屏蔽日志
    Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
    Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)
```

```
//设置运行环境
val conf = new SparkConf().setAppName("SimpleGraphX").setMaster("local")
val sc = new SparkContext(conf)

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

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

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

//******************************************************************************
//*************************  图的属性    ***************************************
```

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

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

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

//Degrees 操作
 println("找出图中最大的出度、入度、度数：")
 def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {
    if (a._2 > b._2) a else b
}
 println("max of outDegrees:" + graph.outDegrees.reduce(max) + " max of
inDegrees:" + graph.inDegrees.reduce(max) + " max of Degrees:" +
graph.degrees.reduce(max))
 println

//*************************************************************************************
//***************************  转换操作  ***********************************
//*************************************************************************************
```



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

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


//******************************************************************************
//*************************  连接操作    ***********************************
//******************************************************************************
println("**************************************************************")
println("连接操作")
println("**************************************************************")
val inDegrees: VertexRDD[Int] = graph.inDegrees
```



```
case class User(name: String, age: Int, inDeg: Int, outDeg: Int)

//创建一个新图，顶点 VD 的数据类型为 User，并从 graph 做类型转换
val initialUserGraph: Graph[User, Int] = graph.mapVertices { case (id, (name, age))
=> User(name, age, 0, 0)}

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

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

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

//*******************************************************************************
//************************  聚合操作  ************************************
//*******************************************************************************
println("*************************************************************")
println("聚合操作")
println("*************************************************************")
println("找出年纪最大的追求者：")
```

```
val oldestFollower: VertexRDD[(String, Int)] = userGraph.mapReduceTriplets[(String, Int)](
  // 将源顶点的属性发送给目标顶点，map 过程
  edge => Iterator((edge.dstId, (edge.srcAttr.name, edge.srcAttr.age))),
  // 得到最大追求者，reduce 过程
  (a, b) => if (a._2 > b._2) a else b
)

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

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

```
      (a,b) => math.min(a,b) // 最短距离
    )
    println(sssp.vertices.collect.mkString("\n"))

    sc.stop()
  }
}
```

3.1.3 运行结果

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

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

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

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

************************************************************
转换操作
************************************************************
顶点的转换操作，顶点 age + 10：
4 is (David,52)
```

1 is (Alice,38)
6 is (Fran,60)
3 is (Charlie,75)
5 is (Ed,65)
2 is (Bob,37)

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

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

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

****
连接操作
****
连接图的属性：

David inDeg: 1 outDeg: 1
Alice inDeg: 2 outDeg: 0
Fran inDeg: 2 outDeg: 0
Charlie inDeg: 1 outDeg: 2
Ed inDeg: 0 outDeg: 3
Bob inDeg: 2 outDeg: 2

出度和入读相同的人员：
David
Bob

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

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

```
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)
```

3.2 PageRank 演示

3.2.1 例子介绍

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

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

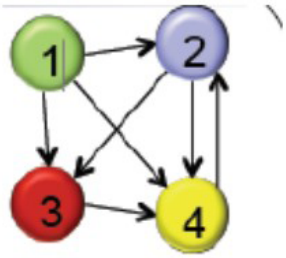
PageRank vector **q** is defined as $q = Gq$

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

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

Algorithm: Iterative powering for finding the first eigen-vector

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



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

3.2.2 测试数据

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

- 顶点为顶点编号和网页标题

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

- 边数据由两个顶点构成

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

3.2.3 程序代码

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

object PageRank {
  def main(args: Array[String]) {
    //屏蔽日志
    Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
    Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)

    //设置运行环境
    val conf = new SparkConf().setAppName("PageRank").setMaster("local")
    val sc = new SparkContext(conf)

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

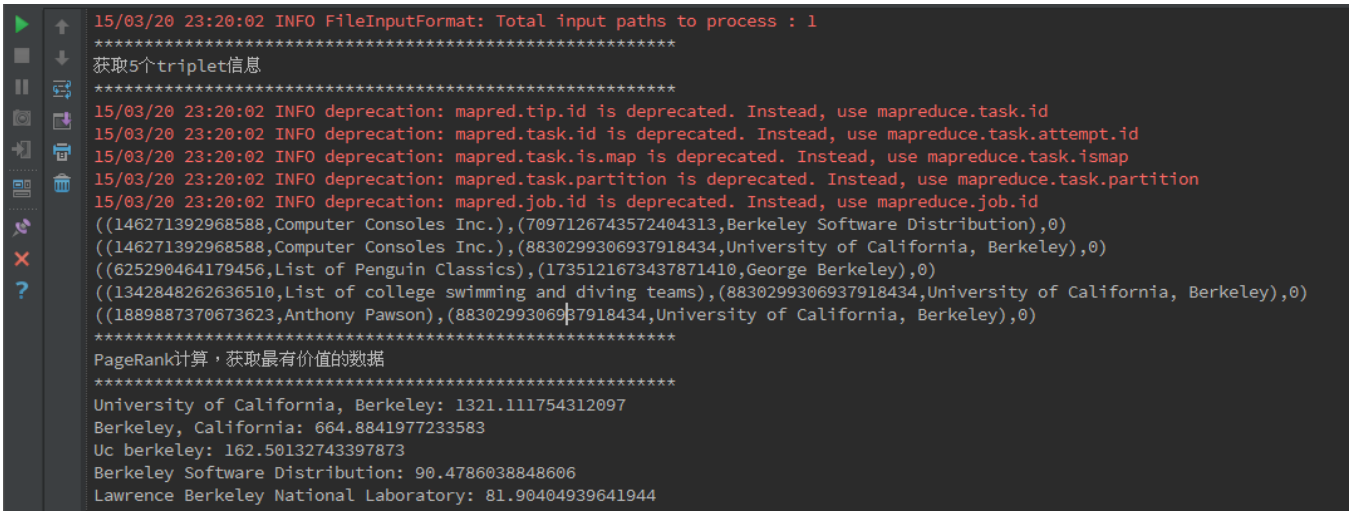
```
//装载顶点和边
val vertices = articles.map { line =>
  val fields = line.split('\t')
  (fields(0).toLong, fields(1))
}

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

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

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

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

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

titleAndPrGraph.vertices.top(10) {
  Ordering.by((entry: (VertexId, (Double, String))) => entry._2._1)
```

```
    }.foreach(t => println(t._2._2 + ": " + t._2._1))

    sc.stop()
  }
}
```

3.2.4 运行结果

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

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

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



4 参考资料

（1）《 GraphX: 基 于 Spark 的 弹 性 分 布 式 图 计 算 系 统 》 http://lidrema.blog.163.com/blog/static/20970214820147199643788/

（2）《快刀初试：Spark GraphX 在淘宝的实践》 http://www.csdn.net/article/2014-08-07/2821097