赶路人

梦想需要一步一步脚踏实地的去实现.....

Spark GraphX

2016-10-24 13:01 699人阅读 评论(0) 收藏 举报

分类：

spark (6)

目录(?) [+]

6、 Spark GraphX

6.1 概述

GraphX是spark的一个新组件用于图和并行图计算。在一个高水平，GraphX通过引进一个新的图抽象扩展了spark RDD：带有顶点和边属性的有向多重图。为了支持图计算，GraphX 提供了很多基本的操作（像 subgraph, joinVertices, and aggregateMessages ）和pregel的一个优化变种。除此之外，GraphX 包含了一个正在增长的图算法和图构造的集合来简化图的分析任务。

6.1.1 从spark1.1 迁移

GraphX 在spark 1.3.1改变了部分用户正在使用api：

- 为了改进性能，引入了一个新版的 mapReduceTriplets 称为aggregateMessages，它取先前返回信息从 mapReduceTriplets 通过一个回调 EdgeContext 而不是通过返回值。我们正在遗弃 mapReduceTriplets，鼓励用户查阅过度指南。
- 在spark1.0和1.1，EdgeRDD的签名切换从 EdgeRDD[ED] 到 EdgeRDD[ED, VD]来进行一些缓存优化。我们已经发现了一个更加优雅的解决方案，恢复了签名到更加自然地 EdgeRDD[ED]类型。

6.2 开始

开始spark GraphX，你首先需要将spark和GraphX导入你的工程，如下：

```
[java]
01. import org.apache.spark._
02. import org.apache.spark.graphx._
03. // To make some of the examples work we will also need RDD
04. import org.apache.spark.rdd.RDD
```

如果你没有使用spark shell你需要一个SparkContext。

6.3 属性图

属性图是一个有向的多重图，用户为每一个顶点（vertex）和边（edge）定义对象。一个有向多重图是一个有向图，潜在的多重平行边共享相同的源和目的顶点（vertex）。支持平行边的能力简化了相同顶点间有多重关系（例如，同时和朋友）的建模场景。每一个顶点以64位长度标识

其他杂七杂八 (5)

文章存档

2017年12月 (1)
2017年11月 (2)
2017年10月 (2)
2017年09月 (9)
2017年08月 (1)

展开

阅读排行

包含mysql 递归查询父节点 和... (11322)
HttpClient4.X的代理添加实现 (6930)
Oozie安装总结 (6556)
spark streaming从指定offset... (2592)
网络舆情监控系统爬虫子系统... (2553)
搭建:canal部署与实例运行和解.. (2209)
oozie 客户端常用命令 (2058)
java.io.IOException: No FileS... (2017)
mysql按日期分组（group by... (1778)
Sqoop导入关系数据库到Hive (1721)

评论排行

HttpClient4.X的代理添加实现 (2)
spark streaming从指定offset... (2)
包含mysql 递归查询父节点 和... (1)
Azkaban简介和使用 (1)
Oozie安装总结 (0)
hadoop异常 java.io.IOExcep... (0)
sqoop中文开发手册 (0)
flume学习（五）：Flume Ch... (0)
flume学习（四）：Flume Int... (0)
flume学习（九）：自定义拦截.. (0)

推荐文章

* 【2017年11月27日】CSDN博客更新周报
* 【CSDN】邀请您来GitChat赚钱啦！
* 【GitChat】精选——JavaScript进阶指南
* 改做人工智能之前，90%的人都没能给自己定位
* TensorFlow 人脸识别网络与对抗网络搭建
* Vue 移动端项目生产环境优化
* 面试必考的计算机网络知识点梳理

最新评论

HttpClient4.X的代理添加实现
huochai1989 ：你这个需要看详细代码，尤其是运行main我才可以确认错误

HttpClient4.X的代理添加实现
沿途看风景 ：400 Bad Request400 Bad Requestnginx/1.10.2访问https网...

Azkaban简介和使用
徒步到天鸣 ：大神你好，我编译安装后进入控制台执行job提示我加载不到job的控件demotype=command...

（vertexId）作为键。GraphX没有对顶点标识符强加一个排序限制。同样地，边有对应的源和目的顶点标识符。

属性图通过顶点（VD）和边（ED）类型参数化。这些类型分别指与顶点和边相关的对象。

GraphX优化了顶点和边类型表示，当它们使用原始数据类型（像 int，double等），使用特殊数组存储它们降低了内存使用。

在一些情况下，同一个图中顶点使用不同的属性类型进行描述。这能通过继承实现。例如，将用户和产品建模为一个二分图，可以用如下方式：

```java
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
```

像RDDs，属性图是不变的、分布式的和容错的。图的值或者结构的改变通过产生一个期望改变的新图来完成。注意，原始图的本质部分（不影响结构、属性和索引）都可以在新图中重用，用来减少这种固有的功能**数据结构**的成本。图被分区通过executors使用一个范围的顶点进行启发式分区。像RDDs一样，当发生故障时，图的每一个分区能被重新创建在不同的机器上。

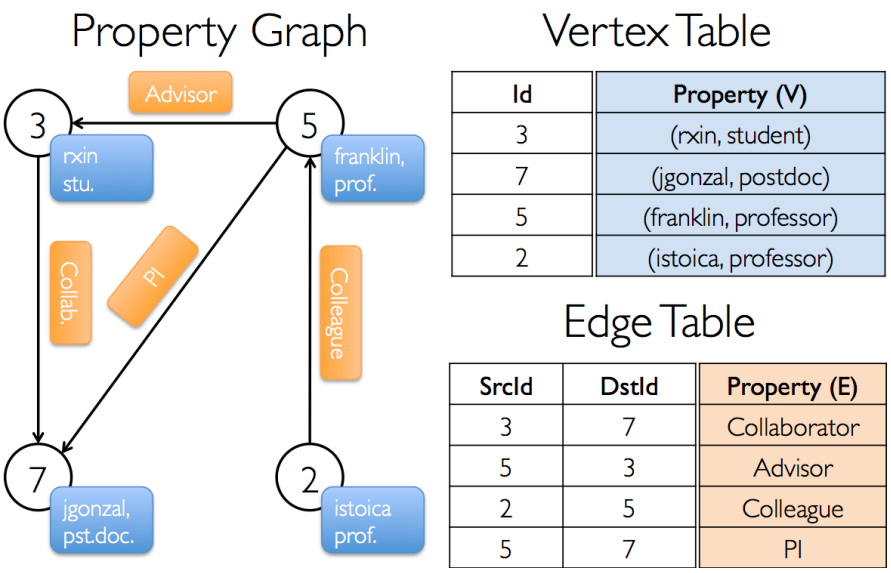
逻辑上属性图对应一对类型化的RDDs集合，其编码每一个顶点和边的属性。因此，图类包含图的顶点和边成员：

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

VertexRDD[VD]和 EdgeRDD[ED]分别对应RDD[(VertexID, VD)]和RDD[Edge[ED]]版本的扩展和优化。VertexRDD[VD] 和 EdgeRDD[ED]提供了额外的功能在图计算中，同时进行了内部优化。讨论 VertexRDD 和 EdgeRDD API细节在vertex和edgeRDDs小节，现在暂且认为简单RDDs形式：RDD[(VertexID, VD)] 和 RDD[Edge[ED]]

6.3.1 属性图属性图实例

假设我们想构建一个包含不同合作者的属性图在图工程中。顶点属性可能包含用户名和职业。我们注释边使用字符串描述合作者之间的关系。



结果图有如下类型签名：

```
[java]
01. val userGraph: Graph[(String, String), String]
```

有很多种方式构建一个属性图从原始文件、RDDs、甚至合成生成器，这些在graph builders节将详细介绍。或许最基本的方法是使用图对象。例如，下面代码展示了使用一系列RDDs集合构建一个图：

```
[java]
01. // Assume the SparkContext has already been constructed
02. val sc: SparkContext
03. // Create an RDD for the vertices
04. val users: RDD[(VertexId, (String, String))] =
05.   sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
06.                       (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
07. // Create an RDD for edges
08. val relationships: RDD[Edge[String]] =
09.   sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
10.                       Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))
11. // Define a default user in case there are relationship with missing user
12. val defaultUser = ("John Doe", "Missing")
13. // Build the initial Graph
14. val graph = Graph(users, relationships, defaultUser)
```

在上面的实例中，我们用到了Edge样本类。Edges 有一个srcId 和 dstId 对应原顶点和目的顶点标识符。除此之外，Edge类有一个attr 成员存储边属性。

我们可以使用graph.vertices和graph.edges解构出一个图对应的顶点和边。

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

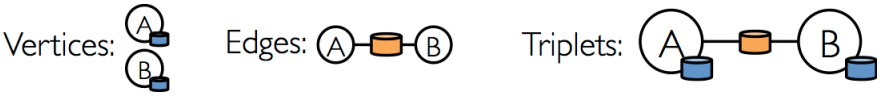
注意：graph.vertices返回一个VertexRDD[(String, String)]，其扩展自RDD[(VertexID, (String, String))]，这样我们可以使用**Scala** case表达式来解构元祖。在另一方面，graph.edges返回一个EdgeRDD 包含Edge[String]对象。我们也可以使用case类类型的构造器，如下所示：

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

除了属性图的顶点和边视图。GraphX 也暴露了一个triplet 视图。triplet视图逻辑上连接了顶点和边属性产生一个 RDD[EdgeTriplet[VD, ED]]，其包含EdgeTriplet类。join可以表达在下面SQL表达式：

```java
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
```

或者生动的表示为：



EdgeTriplet类扩展了Edge类通过增加srcAttr 和dstAttr 成员，它们包含源和目的顶点属性。我们可以使用一个图的 triplet 视图来提供一些字符串描述用户之间的关系。

```java
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(_))
```

6.4 图操作

像RDDs有基本的操作，如 map、filter和reduceByKey，属性图也有一些基本的操作，这些操作采用用户自定义函数，产生转换属性和解构的新图。在Graph中定义的核心操作是已经被优化的实现，组合核心操作的便捷操作定义在GraphOps中。然而，由于Scala的隐士转换在GraphOps中的操作可在Graph的成员中自动获得。例如，我们可以计算每一个顶点的入度（定义在GraphOps），如下所示：

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

区别核心graph操作和GraphOps的原因是在将来支持不同的图表述。每一个图表述必须提供核心操作实现，重复使用在GraphOps中有用的一些操作。

6.4.1 操作列表概要

以下是一个定义在 Graph 和 GraphOps函数快速摘要，为简单起见都作为 Graph 的成员。注意：一些函数签名已经被简化（像默认参数和类型约束被移除），一些高级的函数没有列出，如果需要请参考api文档。

```java
/** 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]
}
```

6.4.2 属性操作

像RDD map操作，属性图包括下面操作：

```
[java]

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

这里每一个操作产生一个新图，其顶点和边被用户定义的map函数修改了。

注意：在每一个实例图结构不受影响。这是这些操作的关键特征，这允许结果图重复利用原始图的结构索引。下面的代码片段逻辑上是等同的，但是第一个没有保存结构索引，其不会从GraphX系统优化中获益：

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

代替，使用mapVertices保护结构索引：

```
[java]
01. val newGraph = graph.mapVertices((id, attr) => mapUdf(id, attr))
```

这些操作经常用来初始化图为了进行特殊计算或者排除不需要的属性。例如，给定一个图，它的出度作为顶点属性（之后描述如何构建这样一个图），我们初始化它为PageRank：

```
[java]
01. // Given a graph where the vertex property is the out degree
02. val inputGraph: Graph[Int, String] =
03.   graph.outerJoinVertices(graph.outDegrees)((vid, _, degOpt) => degOpt.getOrElse(0))
04. // Construct a graph where each edge contains the weight
05. // and each vertex is the initial PageRank
06. val outputGraph: Graph[Double, Double] =
07.   inputGraph.mapTriplets(triplet => 1.0 / triplet.srcAttr).mapVertices((id, _) => 1.0)
```

6.4.3 结构操作

当前，GraphX仅仅支持一个简单的常用结构操作，将来会不断完善。下面是基本结构操作列表：

```
[java]
01. class Graph[VD, ED] {
02.   def reverse: Graph[VD, ED]
03.   def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
04.                vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
05.   def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
06.   def groupEdges(merge: (ED, ED) => ED): Graph[VD,ED]
07. }
```

reverse操作返回一个新图，其所有的边方向反向。有时这是有用的，例如，尝试计算反转的PageRank。因为反转操作没有修改顶点或者边属性或者改变边数量，这能够高效的实现没有数据移动或者复制。

subgraph操作利用顶点和边判断，返回图包含满足判断的顶点，满足边判断的顶点，满足顶点判断的连接顶点。subgraph 操作可以用在一些情景，限制感兴趣的图顶点和边，删除损坏连接。例如，在下面代码中，我们可以移除损坏连接：

```
[java]
01. // Create an RDD for the vertices
02. val users: RDD[(VertexId, (String, String))] =
03.   sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
04.                       (5L, ("franklin", "prof")), (2L, ("istoica", "prof")),
05.                       (4L, ("peter", "student"))))
06. // Create an RDD for edges
07. val relationships: RDD[Edge[String]] =
08.   sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
09.                       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._
  ).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._
  ).collect.foreach(println(_))
```

注意： 在上面的实例中仅仅顶点判断被提到。subgraph 操作默认是true ，如果顶点和边判断没有被提到时。

mask操作构建了一个subgraph 通过返回图，其包含顶点和边也被发现在输入图中。这可以联合subgraph操作使用来限制一个图在其他相关图属性的基础上。例如，我们可以使用丢失顶点的图运行连接组件，然后限制有效子图的返回。

```
[java]

// 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)
```

groupEdges操作合并了多重图的并行边（例如，顶点之间的重复边）。在一些数字应用程序中，并行边能被增加（权重融合）到一个边，因此减少了图的大小。

6.4.4 join操作

在很多情况下，需要将外部数据集合（RDDs）添加到图中。例如，我们可能有额外的用户属性，我们想把它融合到一个存在图中或者我们可能想拉数据属性从一个图到另一个图。这些任务可以使用join操作来实现。下面我们列出了关键的join操作：

```
[java]

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操作连接vertices 和输入RDD，返回一个新图，其顶点属性通过应用用户定义map函数到joined vertices结果上获得的。在RDD顶点没有一个匹配值保留其原始值。

注意：如果RDD对一个给定顶点包含超过一个值，仅仅有一个将会使用。因此，建议输入RDD保持唯一性，这可以使用下面方法，预索引结果值，加快join执行速度。

```
[java]

val nonUniqueCosts: RDD[(VertexID, Double)]
val uniqueCosts: VertexRDD[Double] =
```

```
  graph.vertices.aggregateUsingIndex(nonUnique, (a,b) => a + b)
val joinedGraph = graph.joinVertices(uniqueCosts)(
  (id, oldCost, extraCost) => oldCost + extraCost)
```

更加一般的 outerJoinVertices 行为和joinVertices相似除了用户定义的map函数被应用到所有顶点和可以改变顶点类型。因为不是所有的顶点有一个匹配值在输入RDD，map函数使用了一个Option类型。例如，我们可以设置一个图对PageRank通过初始化顶点属性使用出度：

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

你可能已经觉察到了柯里函数模式的多参数列表（例如f(a)(b)）被使用在上面的实例中。当我们能有等同写f(a)(b)为 f(a,b)，这将意味着类型接口b将不会依赖于a。因此用户需要提供类型注释对用户自定义函数：

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

6.4.5 相邻聚合（Neighborhood Aggregation）

在图分析任务中一个关键步骤就是聚集每一个顶点的邻居信息。例如，我们想知道每一个用户的追随者数量或者追随者的平均年龄。一些迭代的图算法（像PageRank,最短路径和联通组件）反复的聚集相邻顶点的属性（像当前pagerank值，源的最短路径，最小可到达的顶点id）。

为了改善原始聚集操作的性能，将graph.mapReduceTriplets 改为新的graph.AggregateMessages。当然API的改变很小，下面提供了过度向导。

6.4.5.1 信息聚集（Aggregate Messages (aggregateMessages)）

在GraphX中核心的聚集操作是aggregateMessages。这个操作应用了一个用户定义的sendMsg函数到图中的每一个边 triplet，然后用mergeMsg函数在目的节点聚集这些信息。

```
[java]
class Graph[VD, ED] {
  def aggregateMessages[Msg: ClassTag](
      sendMsg: EdgeContext[VD, ED, Msg] => Unit,
      mergeMsg: (Msg, Msg) => Msg,
      tripletFields: TripletFields = TripletFields.All)
    : VertexRDD[Msg]
}
```

用户定义一个 sendMsg 函数使用 EdgeContext，其暴露了源和目的属性，及它们相关的边属性，函数（sendToSrc, and sendToDst）发送信息到源和目的属性。考虑 sendMsg 作为map-reduce中的map函数。用户定义的mergeMsg函数使用到相同顶点的两个信息，将它们计算产出一条信息。考虑mergeMsg 作为map-reduce的reduce函数。aggregateMessages函数返回一个VertexRDD[Msg]，其包含了到达每一个顶点的融合信息（Msg类型）。没有接收一个信息的顶点

不被包含在返回的VertexRDD中。

除此之外，aggregateMessages使用了一个选项tripletsFields，其表明在EdgeContext中什么数据可以被访问（例如，有源顶点属性没有目的顶点属性）。tripletsFields 可能的选项被定义在TripletsFields中，默认值为 TripletFields.All，其表明用户定义的sendMsg 函数可以访问EdgeContext的任何属性。tripletFields 参数通知GraphX仅仅需要EdgeContext的一部分，允许GraphX 选择一个优化的连接策略。例如，如果我们计算每一个用户追随者的平均年龄，我们仅仅要求源属性即可，所以我们使用 TripletFields.Src 来表明我们仅仅使用源属性。

在之前的GraphX版本中，我们使用字节码检测来推断 TripletFields ，然而我们已经发现字节码检测是稍微不可靠，所以代替先前方式使用更加明确的用户控制。

在下面的实例中，我们使用 aggregateMessages操作来计算每一个用户更年长追随者的平均年龄。

```java
// 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 simpl
val graph: Graph[Double, Int] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapVertices( (id, _) => id.toDoubl
// 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) => totalAg
// Display the results
avgAgeOfOlderFollowers.collect.foreach(println(_))
```

当messages（以及消息总数）是常量大小（例如， float和addition代替lists和连接（concatenation）），aggregateMessages 操作效果最好。

6.4.5.2 Map Reduce Triplets Transition Guide (Legacy)

在早的GraphX版本中我们计算邻居聚合使用mapReduceTriplets操作：

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

mapReduceTriplets 操作应用用户定义的map函数到每一个triplet ，使用用户定义的reduce函数聚合产生 messages。。然而，我们发现用户返回迭代器是昂贵的，它抑制了我们应用额外优化(例如，本地顶点的重新编号)的能力。在 aggregateMessages 中我们引进了EdgeContext，其暴露triplet属性，也明确了函数发送信息的源和目的顶点。除此之外，我们移除了字节码检测，取而代之的是要求用户指明哪个triplet属性被需要。

下面的代码块使用 mapReduceTriplets:

```
[java]
val graph: Graph[Int, Float] = ...
def msgFun(triplet: Triplet[Int, Float]): Iterator[(Int, String)] = {
  Iterator((triplet.dstId, "Hi"))
}
def reduceFun(a: Int, b: Int): Int = a + b
val result = graph.mapReduceTriplets[String](msgFun, reduceFun)
```

使用aggregateMessages重写为：

```
[java]
val graph: Graph[Int, Float] = ...
def msgFun(triplet: EdgeContext[Int, Float, String]) {
  triplet.sendToDst("Hi")
}
def reduceFun(a: Int, b: Int): Int = a + b
val result = graph.aggregateMessages[String](msgFun, reduceFun)
```

6.4.5.3 计算度（Degree）信息

一个普通的聚合任务是计算每一个顶点的度：每一个顶点边的数量。在有向图的情况下，它经常知道入度，出度和每个顶点的总度。 GraphOps 类包含了每一个顶点的一系列的度的计算。例如：在下面将计算最大入度，出度和总度：

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

6.4.5.4 邻居收集

在一些情形下，通过收集每一个顶点的邻居顶点和它的属性来表达计算是更加容易的。这容易完成通过使用 collectNeighborIds 和 collectNeighbors 操作。

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

这些操作代价比较高，由于复制信息和要求大量的通信。尽可能直接使用aggregateMessages 操作完成相同的计算。

6.4.6 缓存和取消缓存

在spark中，RDDs默认没有持久化在内存中。当多次使用它们时，为了避免重复计算，它们必须被明确缓存。GraphX 中的图也是相同的方式。当使用一个图多次时，首先确认调用Graph.cache()。在迭代计算中，为了最好的性能，uncaching 也可能是需要的。默认，缓存的RDDs和图将会保留在内存中直到内存不足，迫使它们以LRU顺序被驱除。对于迭代计算，从过去相关迭代产生的中间结

果将被缓存，即使最终被驱除，不需要的数据存储在内存中将会减缓垃圾回收。取消不需要的中间结果的缓存将会更加高效。这涉及每次迭代物化（缓存和强迫）一个图和RDD，取消所有其他数据集缓存，仅仅使用物化数据集在将来迭代中。然而，因为图由多个RDDs组成，正确解除他们的持久化是比较难的。对迭代计算我们推荐使用 Pregel API，其能正确的解除中间结果的持久化。

6.5 Pregel API

Graphs 本质上就是递归的数据结构，顶点的属性依赖于他们邻居的属性，邻居属性依次依赖于他们邻居的属性。因此，一些重要的图算法迭代的重复计算每一个顶点的属性直到固定条件得到满足。一些列的图并行抽象已经被提出来满足这些迭代算法。GraphX 提供了一个变种的Pregel API。

在GraphX中，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操作执行一系列高级步骤，顶点从过去的超级步骤接收他们流入信息总和，对顶点属性计算一个新值，发送信息到邻居节点在下一个高级步骤。不像Pregel，信息作为边triplet函数被平行计算，信息计算访问源和目的顶点属性。没有接收信息的顶点在一个高级步骤中被跳过。当没有保留信息时，pregel终止迭代并返回最终图。

注意： 不像更加标准的Pregel实现，GraphX 的顶点仅仅发送消息到邻居顶点，使用用户定义的消息函数并行构建消息。这些限制允许GraphX额外优化。

下面是Pregel操作的类型签名和它的实现概述（注意，graph.cache调用被移除）

```java
class GraphOps[VD, ED] {
  def pregel[A]
      (initialMsg: A,
       maxIter: Int = Int.MaxValue,
       activeDir: EdgeDirection = EdgeDirection.Out)
      (vprog: (VertexId, VD, A) => VD,
       sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
       mergeMsg: (A, A) => A)
    : Graph[VD, ED] = {
    // Receive the initial message at each vertex
    var g = mapVertices( (vid, vdata) => vprog(vid, vdata, initialMsg) ).cache()
    // compute the messages
    var messages = g.mapReduceTriplets(sendMsg, mergeMsg)
    var activeMessages = messages.count()
    // Loop until no messages remain or maxIterations is achieved
    var i = 0
    while (activeMessages > 0 && i < maxIterations) {
      // Receive the messages: -----------------------------------------------------------
      // Run the vertex program on all vertices that receive messages
      val newVerts = g.vertices.innerJoin(messages)(vprog).cache()
      // Merge the new vertex values back into the graph
      g = g.outerJoinVertices(newVerts) { (vid, old, newOpt) => newOpt.getOrElse(old) }.ca
      // Send Messages: ------------------------------------------------------------------
      // Vertices that didn't receive a message above don't appear in newVerts and therefo
      // get to send messages.  More precisely the map phase of mapReduceTriplets is only
      // on edges in the activeDir of vertices in newVerts
      messages = g.mapReduceTriplets(sendMsg, mergeMsg, Some((newVerts, activeDir))).cache
      activeMessages = messages.count()
      i += 1
    }
    g
  }
}
```

注意： Pregel使用两个参数列表（像graph.pregel(list1)(list2)）。第一个参数列表包含配置参数包括初始化信息，最大迭代次数和发送信息边方向（默认沿着out边）。第二个参数列表包含用户自定义函数，对应接收信息（顶点程序Vprog），计算信息（sendMsg）和组合信息（mergeMsg）。

我们可以使用Pregel操作表达计算，像下面的单元最短路径实例。

```
[java]
import org.apache.spark.graphx._
// Import random graph generation library
import org.apache.spark.graphx.util.GraphGenerators
// A graph with edge attributes containing distances
val graph: Graph[Int, Double] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapEdges(e => e.attr.toDouble)
val sourceId: VertexId = 42 // The ultimate source
// Initialize the graph such that all vertices except the root have distance infinity.
val initialGraph = graph.mapVertices((id, _) => if (id == sourceId) 0.0 else Double.Positi
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"))
```

6.6 图构建（Graph Builders）

GraphX 提供了一些方法来构建一个图，从一个RDD的顶点和边或者硬盘上。默认情况下，没有图构建者重新将图的边分区；取而代之，边留住他们默认的分区（像hdfs原始块）。Graph.groupEdges 要求图重新分区，因为它假定相同的边在同一个分区，所有你必须在调用groupEdges之前调用Graph.partitionBy 。

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

GraphLoader.edgeListFile提供了一种方式加载硬盘上边的列表。它解析下面的邻接对（起始顶点id和目的顶点id）列表,跳过#开始的行注释：

```
[java]
# This is a comment
2 1
4 1
1 2
```

它从指定的边创建一个图，自动创建边涉及的顶点。所有的顶点和边属性默认为1。canonicalOrientation参数允许重定向边在正方向(srcid

6.7 顶点和边RDDs

GraphX 公开了存储在图中顶点和边的RDD视图。然而，因为GraphX 使用优化的数据结构存储顶点和边，这些数据结构提供了额外的功能，顶点和边被返回为VertexRDD 和 EdgeRDD。这一节我们温习这些类型的额外有用的功能。

6.7.1 VertexRDDs

VertexRDD[A]继承 RDD[(VertexID, A)] ，并且增加了限制：每一个VertexID 仅出现一次。除此之外，VertexRDD[A] 代表每一个顶点的属性为A。在内部，这被实现通过存储顶点属性在一个可重复使用的hash-map数据结构。因此，如果两个 VertexRDDs 从相同的基 VertexRDD 获得（例，通过filter或者mapValues），他们可以在一个常数时间进行join，没有hash评估。为了评估这些索引数据结构，VertexRDD 公开了下面额外的功能：

```
[java]
01. class VertexRDD[VD] extends RDD[(VertexID, VD)] {
02.   // Filter the vertex set but preserves the internal index
03.   def filter(pred: Tuple2[VertexId, VD] => Boolean): VertexRDD[VD]
04.   // Transform the values without changing the ids (preserves the internal index)
05.   def mapValues[VD2](map: VD => VD2): VertexRDD[VD2]
06.   def mapValues[VD2](map: (VertexId, VD) => VD2): VertexRDD[VD2]
07.   // Remove vertices from this set that appear in the other set
08.   def diff(other: VertexRDD[VD]): VertexRDD[VD]
09.   // Join operators that take advantage of the internal indexing to accelerate joins (subs
10.   def leftJoin[VD2, VD3](other: RDD[(VertexId, VD2)])
(f: (VertexId, VD, Option[VD2]) => VD3): VertexRDD[VD3]
11.   def innerJoin[U, VD2](other: RDD[(VertexId, U)])
(f: (VertexId, VD, U) => VD2): VertexRDD[VD2]
12.   // Use the index on this RDD to accelerate a `reduceByKey` operation on the input RDD.
13.   def aggregateUsingIndex[VD2]
(other: RDD[(VertexId, VD2)], reduceFunc: (VD2, VD2) => VD2): VertexRDD[VD2]
14. }
```

注意：例如，filter 操作怎样返回一个 VertexRDD。Filter 实际上的实现使用了一个 BitSet ，因此可以重复使用索引和保留了快速join其他VertexRDDs的能力。同样地，mapValues操作不允许 map 函数改变VertexID ，因此相同HashMap数据结构被重复使用。当join两个来自相同HashMap的 VertexRDDs，leftJoin和innerJoin 都能使用，join使用线性扫描而不是代价很高的点查找。aggregateUsingIndex 操作是有用的对从RDD[(VertexID, A)]演变的VertexRDD高效**架构**。概念上，如果已经构建了一系列顶点的VertexRDD[B]，其是一些RDD[(VertexID, A)]的超顶点集，然后我们可以在聚合和随后的索引RDD[(VertexID, A)]中重复使用索引。

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

6.7.2 EdgeRDDs

EdgeRDD[ED]继承RDD[Edge[ED]]，使用不同的分区策略（定义在PartitionStrategy）组织到块中。在每一个分区中，边属性和邻接结构被分别存储，确保属性值变化时可以最大化的重复使用。在EdgeRDD 中有三个额外的函数：

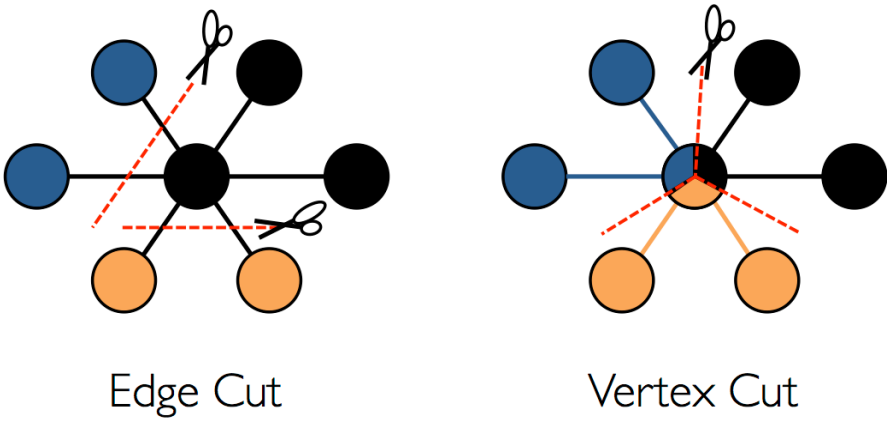
```
[java]
01. // Transform the edge attributes while preserving the structure
02. def mapValues[ED2](f: Edge[ED] => ED2): EdgeRDD[ED2]
03. // Revere the edges reusing both attributes and structure
04. def reverse: EdgeRDD[ED]
05. // Join two `EdgeRDD`s partitioned using the same partitioning strategy.
```

```
06. def innerJoin[ED2, ED3](other: EdgeRDD[ED2])
    (f: (VertexId, VertexId, ED, ED2) => ED3): EdgeRDD[ED3]
```

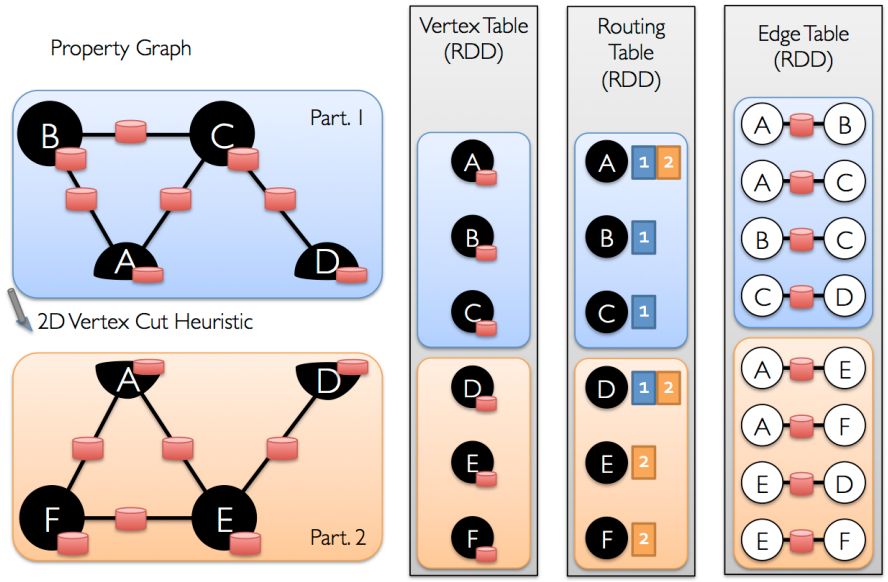
在最多应用程序中，我们已经发现EdgeRDD 操作通过图操作来实现或者操作定义在基类RDD中。

6.8 优化表示

分布式图的GraphX表示的详细优化描述超出本向导的范畴，一些高水平的理解可能帮助对扩展算法的设计和API的最佳使用。GraphX 对分布式图分区采用了vertex-cut的方法：



代替沿着边拆分图，GraphX 沿着vertices 分片，这种方式可以减少通信和存储开销。逻辑上，这指分配边到机器上用时允许顶点跨多台机器。这种分配边的确切方法依赖于 PartitionStrategy，在不同启发式方式中有不同的权衡。用户可以在重新分区图的不同策略和 Graph.partitionBy操作之间选择。默认的分区策略是使用图构造的边初始分区。然而，用户可以容易的切换到2D-partitioning或者其他启发式的分区策略。



一旦边被分区，对高效的并行图计算的挑战就是高效的连接顶点属性到边。因为典型的现实图有更多的边比顶点，我们移动顶点属性到边。因为不是所有的分区包含边邻接的所有顶点，我们内在的维持一个路由表，当对triplets 和aggregateMessages实现连接请求时，路由表标识广播顶点位置。

6.9 图算法

GraphX 包含一些列的图算法来简化分析任务。算法被包含在org.apache.spark.graphx.lib包里面，能被Graph通过GraphOps直接访问。这部分描述算法和算法如何使用。

6.9.1 PageRank

PageRank 测量在图中每一个顶点的重要性，假设一条u到v的代表u对v重要性的一个支持。例如，

如果一个Twitter用户被其他用户浏览，这个用户排名将会升高。

GraphX 自带了静态和动态的PageRank 实现，作为PageRank对象的方法。静态的PageRank 运行固定的迭代次数，然而动态的PageRank 运行知道排名收敛（例如，超过设定容忍值停止迭代）。GraphOps允许直接调用这些算法作为graph的方法。

GraphX 也包含了一个社会网络数据集实例，我们可以在上面运行PageRank 。一个 [illegible] graphx/data/users.txt中给出，用户之间的关系在 graphx/data/followers.txt中给 [illegible] 每一个用户的PageRank 如下：

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

6.9.2 Connected Components

连通图算法使用最小编号的顶点标记图的连通体。例如，在一个社会网络，连通图近似聚类。

GraphX 在 ConnectedComponents 对象中包含一个算法实现，我们计算连通图实例，数据集和PageRank部分一样：

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

6.9.3 Triangle Counting

当顶点有两个邻接顶点并且它们之间有边相连，它就是三角形的一部分。GraphX 在TriangleCount对象中实现了一个三角形计数算法，其确定通过每一个顶点的三角形数量，提供了一个集群的测量。我们计算社交网络三角形的数量，数据集同样使用PageRank部分数据集。注意：三角形数量要求边是标准方向（srcId < dstId），图使用Graph.partitionBy进行分区。

```
[java]
// Load the edges in canonical order and partition the graph for triangle count
val graph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt", true).partitionBy(Pa
// Find the triangle count for each vertex
val triCounts = graph.triangleCount().vertices
// Join the triangle counts with the usernames
```

```
val users = sc.textFile("graphx/data/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val triCountByUsername = users.join(triCounts).map { case (id, (username, tc)) =>
  (username, tc)
}
// Print the result
println(triCountByUsername.collect().mkString("\n"))
```

6.10 Examples

假设我们想从一些文本文件构建一个图，约束图为重要的人际关系和用户，在子图运行page-rank，然后返回顶点用户相关的属性。我们使用GraphX做这些事情仅仅需要几行代码：

```
[java]

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

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

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

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

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

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

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

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


- 上一篇 Cloudera Manager5配置管理之配置Namenode 的HA
- 下一篇 马士兵-多线程学习第01课 线程的创建和启动

- 王家林+Spark+GraphX大规模图计算和图
- MySQL在微信支付下的高可用运营--莫晓东
- Spark GraphX in Action 无水印pdf
- 容器技术在58同城的实践--姚远
- Spark_GraphX大规模图计算和图挖掘V3.0
- SDCC 2017之容器技术实战线上峰会
- Spark_GraphX大规模图计算和图挖掘
- SDCC 2017之数据库技术实战线上峰会
- Spark Graphx in Action
- 腾讯云容器服务架构实现介绍--董晓杰
- Spark.GraphX.in.Action.2016.6.pdf
- 微博热点事件背后的数据库运维心得--张冬洪
- Spark GraphX 对图进行可视化
- spark graphx 图计算demo,结果展现
- Spark GraphX原理介绍
- 图并行计算实践（二）（spark streaming+graph...

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场