同濟大學
TONGJI UNIVERSITY

大数据管理系统调研报告

Spark GraphX 图数据分布式存储原理

学　　院：电子信息工程学院
专　　业：数据科学与大数据技术
学生姓名：苗成林
学　　号：1851804
指导教师：李文根

2021 年 12 月 29 日

摘要

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 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（如 Graph 和 GraphLab）的发展。本文从 GraphX 图存储系统的原理介绍，点分割存储算法和 GraphX 构建图的过程三个维度介绍 GraphX 系统。

关键词：Apache Spark,GraphX, 点分割, 分布式计算, 可伸缩的分布式数据集 (RDD)

目录

1 GraphX 图存储系统介绍

1.1 GraphX 的优势

传统的 spark 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。

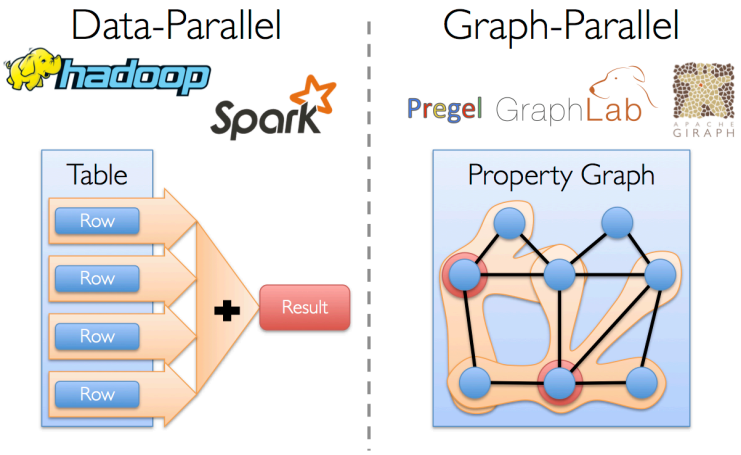

图 1: *DataParallel* 与 *GraphParallel* 示意图

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 的集合视图，而分布式图计算采用了一种 vertex-centric 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 以及 label diffusion，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。

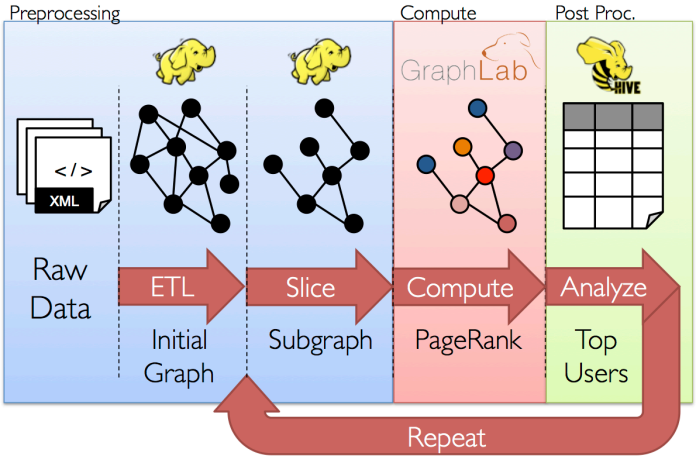

图 2: 图计算基本流程

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：

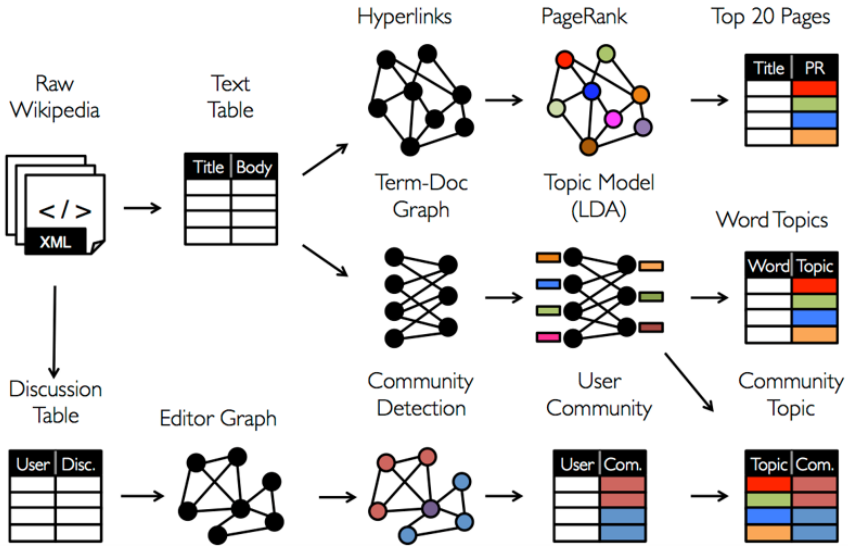

图 3: 图与表之间的数据互动

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 GraphX 整体架构

GraphX 的整体架构可以分为三个部分：存储层和原语层：Graph 类是图计算的核心类，内部含有 VertexRDD、EdgeRDD 和 RDD[EdgeTriplet] 引用。GraphImpl 是 Graph 类的子类，实现了图操作。接口层：在底层 RDD 的基础之上实现 Pragel 模型，BSP 模式的计算接口。算法层：基于 Pregel 接口实现了常用的图算法。包含：PageRank、SVDPlusPlus、TriangleCount、ConnectedComponents、StronglyConnectedConponents 等算法。

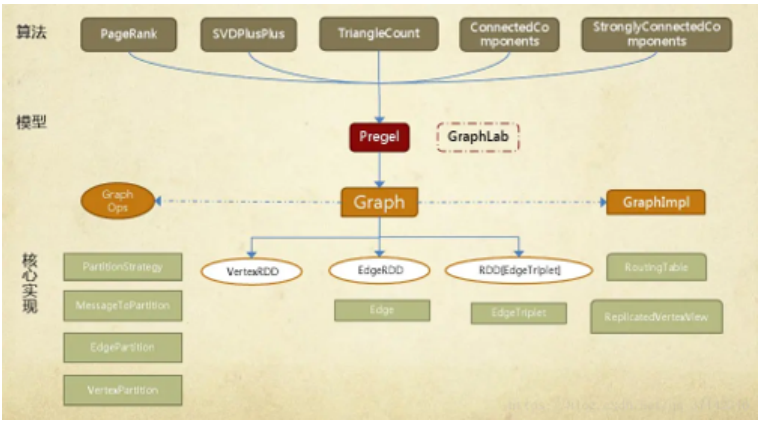

图 4: *GraphX* 整体架构

1.3 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

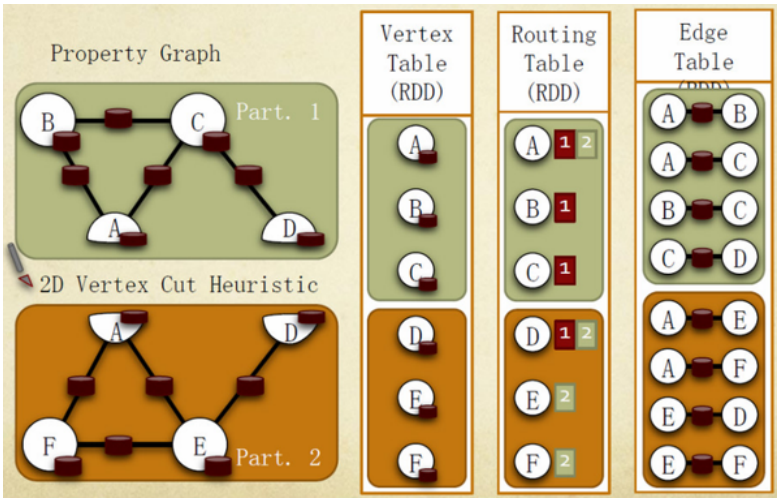

图 5: *GraphX 存储模式*

1.4 GraphX 数据结构

vertices 对应着名为 VertexRDD 的 RDD。这个 RDD 由顶点 id 和顶点属性两个成员变量。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。VertexId 实际上是一个 Long 类型的数据。

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。Edge 代表边，由源顶点 id、目标顶点 id、以及边的属性构成。

triplets 表示边点三元组，如下图所示（其中圆柱形分别代表顶点属性与边属性）：

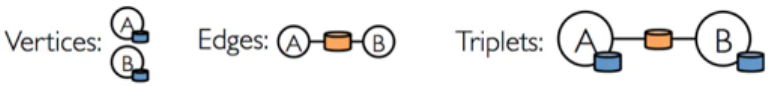

图 6: *GraphX 数据结构*

通过 triplets 成员，用户可以直接获取到起点顶点、起点顶点属性、终点顶点、终点顶点属性、边与边属性信息。triplets 的生成可以由边表与顶点表通过 ScrId 与 DstId 连接而成。

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]]。

2 GraphX 点分割存储算法

GraphX 借鉴 powerGraph，使用的是点分割方式存储图。这种存储方式特点是任何一条边只会出现在一台机器上，每个点有可能分布到不同的机器上。当点被分割到不同机器上时，是相同的镜像，但是有一个点作为主点, 其他的点作为虚点，当点的数据发生变化时, 先更新主点的数据，然后将所有更新好的数据发送到虚点所在的所有机器，更新虚点。这样做的好处是在边的存储上是没有冗余的，而且对于某个点与它的邻居的交互操作，只要满足交换律和结合律，就可以在不同的机器上面执行，网络开销较小。但是这种分割方式会存储多份点数据，更新点时，会发生网络传输，并且有可能出现同步问题。

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 RandomVertexCut

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。

```
1 case object RandomVertexCut extends PartitionStrategy {
2         override def getPartition(src: VertexId, dst: VertexId, numParts:
              PartitionID): PartitionID = {
3             math.abs((src, dst).hashCode()) % numParts
4         }
5 }
```

2.2 CanonicalRandomVertexCut

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。

```
1 case object CanonicalRandomVertexCut extends PartitionStrategy {
2         override def getPartition(src: VertexId, dst: VertexId,numParts:
              PartitionID):
3         PartitionID = {
4             if (src < dst) {
```

```
5                       math.abs((src, dst).hashCode()) % numParts
6                   } else {
7                       math.abs((dst, src).hashCode()) % numParts
8                   }
9           }
10  }
```

2.3 EdgePartition1D

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。

```
1   case object EdgePartition1D extends PartitionStrategy {
2           override def getPartition(src: VertexId, dst: VertexId, numParts:
                PartitionID): PartitionID = {
3               val mixingPrime: VertexId = 1125899906842597L
4               (math.abs(src * mixingPrime) % numParts).toInt
5           }
6   }
```

2.4 EdgePartition2D

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。

```
1   case object EdgePartition2D extends PartitionStrategy {
2           override def getPartition(src: VertexId, dst: VertexId, numParts:
                PartitionID): PartitionID = {
3               val ceilSqrtNumParts: PartitionID = math.ceil(math.sqrt(
                    numParts)).toInt
4               val mixingPrime: VertexId = 1125899906842597L
5               if (numParts == ceilSqrtNumParts * ceilSqrtNumParts) {
6                       // Use old method for perfect squared to ensure we
                            get same results
7                       val col: PartitionID = (math.abs(src * mixingPrime)
                            % ceilSqrtNumParts).toInt
8                       val row: PartitionID = (math.abs(dst * mixingPrime)
                            % ceilSqrtNumParts).toInt
```

```
9                       (col * ceilSqrtNumParts + row) % numParts
10                  } else {
11                      // Otherwise use new method
12                      val cols = ceilSqrtNumParts
13                      val rows = (numParts + cols - 1) / cols
14                      val lastColRows = numParts - rows * (cols - 1)
15                      val col = (math.abs(src * mixingPrime) % numParts /
                            rows).toInt
16                      val row = (math.abs(dst * mixingPrime) % (if (col <
                            cols - 1) rows else lastColRows)).toInt
17                      col * rows + row
18                  }
19          }
20  }
```

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:

```
1  ------------------------------------
2  v0  | P0 *     | P1       | P2    *  |
3  v1  |  ****    |  *       |          |
4  v2  |  ******* |      **  |  ****    |
5  v3  |  *****   |  *  *    |       *  |
6  ------------------------------------
7  v4  | P3 *     | P4 ***   | P5 **  * |
8  v5  |  *  *    |  *       |          |
9  v6  |       *  |      **  |  ****    |
10 v7  |  * * *   |  *  *    |       *  |
11 ------------------------------------
12 v8  | P6   *   | P7    *  | P8 *   *|
13 v9  |     *    |  *    *  |          |
14 v10 |       *  |      **  |  *  *    |
15 v11 | * <-E    |  ***     |       ** |
16 ------------------------------------
```

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。

3 GraphX 构建图的过程

3.1 构建边 EdgeRDD

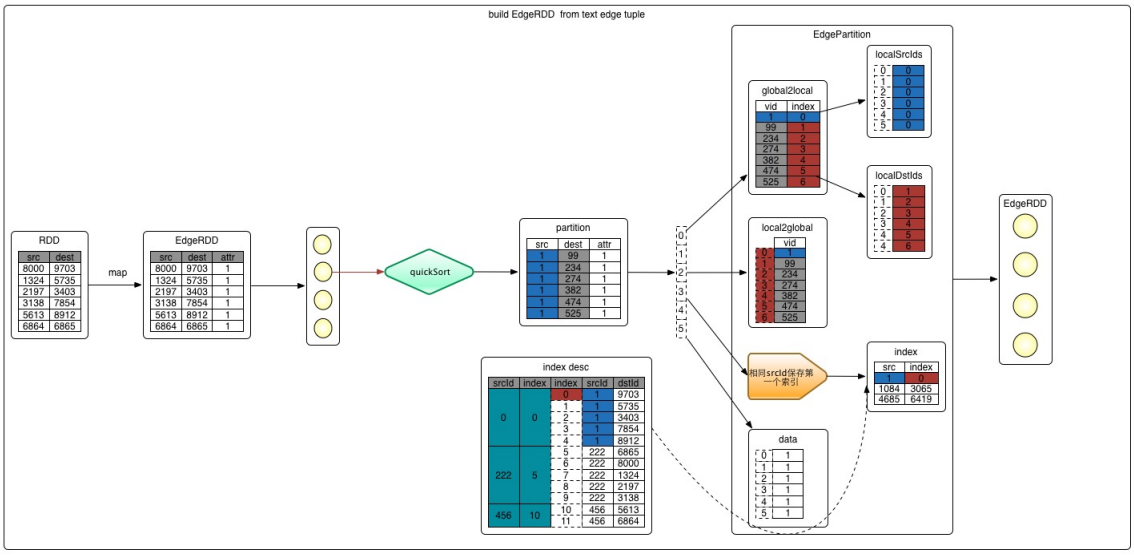

图 7: *EdgeRDD 构建流程*

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

```
1  val rawEdgesRdd: RDD[(Long, Long)] =
2  sc.textFile(input).filter(s => s != "0,0").repartition(partitionNum).map {
3          case line =>
4          val ss = line.split(",")
5          val src = ss(0).toLong
6          val dst = ss(1).toLong
7          if (src < dst)
8          (src, dst)
9          else
10         (dst, src)
11 }.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

```
1 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。

```
1 val graph = GraphImpl(edges, defaultValue , edgeStorageLevel ,
      vertexStorageLevel )
2 def apply[VD: ClassTag, ED: ClassTag](
3 edges: RDD[Edge[ED]],
4 defaultVertexAttr : VD,
5 edgeStorageLevel : StorageLevel ,
6 vertexStorageLevel : StorageLevel ): GraphImpl[VD, ED] = {
7           fromEdgeRDD(EdgeRDD.fromEdges(edges), defaultVertexAttr,
              edgeStorageLevel, vertexStorageLevel )
8 }
```

3.2 构建顶点 VertexRDD

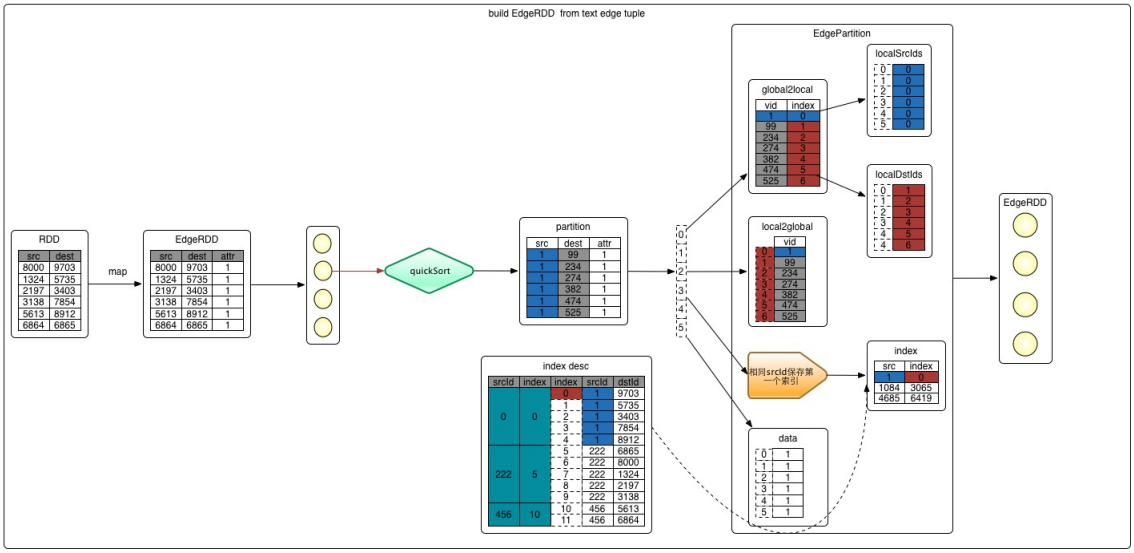

图 8: *VertexRDD 构建流程*

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：

1 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。

```
1 private [graphx] def createRoutingTables (
```

```
2 edges: EdgeRDD[_], vertexPartitioner: Partitioner): RDD[RoutingTablePartition
    ] = {
3       // 将edge partition 中的数据转换成RoutingTableMessage类型,
4       val vid2pid = edges.partitionsRDD.mapPartitions(_.flatMap(
5       Function.tupled(RoutingTablePartition.edgePartitionToMsgs)))
6 }
```

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。

2 根据路由表生成分区对象

```
1 private [graphx] def createRoutingTables(
2 edges: EdgeRDD[_], vertexPartitioner: Partitioner): RDD[RoutingTablePartition
    ] = {
3       // 将edge partition 中的数据转换成RoutingTableMessage类型,
4       val numEdgePartitions = edges.partitions.size
5       vid2pid.partitionBy(vertexPartitioner).mapPartitions(
6       iter => Iterator(RoutingTablePartition.fromMsgs(numEdgePartitions,
            iter)),
7       preservesPartitioning = true)
8 }
```

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。

3.3 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

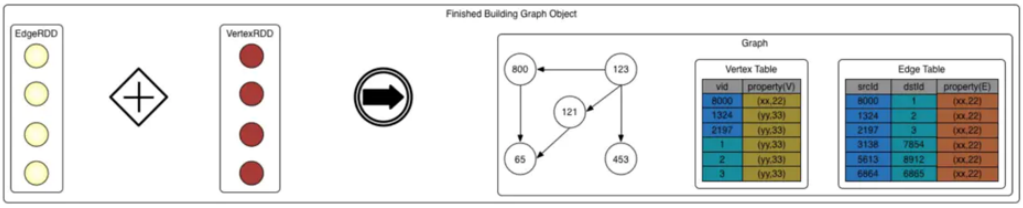

图 9: 生成 *Graph* 对象

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。

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

参考文献

[1] Gonzalez, Joseph E. et al. “GraphX: Graph Processing in a Distributed Dataflow Framework.” OSDI (2014).

[2] Gonzalez, Joseph E. et al. “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.” OSDI (2012).

[3] Xin, Reynold et al. “GraphX: Unifying Data-Parallel and Graph-Parallel Analytics.” ArXiv abs/1402.2394 (2014): n. pag.

[4] Gonzalez, Joseph E.. “From graphs to tables the design of scalable systems for graph analytics.” Proceedings of the 23rd International Conference on World Wide Web (2014): n. pag.

[5] Xin, Reynold. “Go with the Flow: Graphs, Streaming and Relational Computations over Distributed Dataflow.” (2018).

[6] Crankshaw, Daniel et al. “The GraphX Graph Processing System.” (2013).

[7] Kyrola, Aapo et al. “GraphChi: Large-Scale Graph Computation on Just a PC.” OSDI (2012).

[8] Salihoglu, Semih and Jennifer Widom. “GPS: a graph processing system.” SSDBM (2013).

[9] https: //endymecy.gitbooks.io/spark-graphx-source-analysis/content/build-graph.html

[10] https: //blog.csdn.net/$weixin_4$7134119/$article/details$/117756930

[11] https: //www.cnblogs.com/$zhanghuicheng/p$/11484200.$html$