Spark GraphX的概要与应用

作者：费英林

目录

[1. 系统概要 2](#_Toc430176002)

[2. 属性图 2](#_Toc430176003)

[2.1. 属性图示例 3](#_Toc430176004)

[3. 图操作符 4](#_Toc430176005)

[3.1. 操作符列表 5](#_Toc430176006)

[4. Pregel API 6](#_Toc430176007)

[5. Graph构建器 7](#_Toc430176008)

[6. 顶点和边的RDD 8](#_Toc430176009)

[6.1. VertexRDD 8](#_Toc430176010)

[6.2. EdgeRDD 9](#_Toc430176011)

[7. 优化表示 9](#_Toc430176012)

[8. 图算法 10](#_Toc430176013)

[8.1. PageRank 10](#_Toc430176014)

[8.2. 连通图 11](#_Toc430176015)

[8.3. 三角计算 11](#_Toc430176016)

[9. 实例 12](#_Toc430176017)

[9.1. 下载测试数据 12](#_Toc430176018)

[9.2. 运行 12](#_Toc430176019)

[9.3. 结果确认 12](#_Toc430176020)

# 系统概要

GraphX是Spark中关于图和图并行计算的一个组件。GraphX扩展了Spark RDD，引入一个新的Graph抽象概念，Graph是一个有向多重图，每个点和边都有自己的属性。GraphX提供了一组基本的操作符来支持图计算，如subgraph、joinVertices和aggregateMessages等，GraphX还包含了优化过的Pregel API和一些图算法来简化图的分析工作。

# 属性图

属性图是一种有向多重图，每个顶点和边上都附有用户定义的对象。有向多重图中可能包含一些平行边，这些平行边共享源顶点和目标顶点，这简化了对于在相同顶点间包含多重关系的应用的建模过程。例如同事和朋友的关系，两个人之间可能既是同事又是朋友的关系。每个顶点由一个唯一的64位标志符来标记，这些标志符之间并不存在着顺序关系。每条边都有对应的源和目标顶点标志符。

在属性图里，与每个顶点和边所关联对象的类型是参数化的，即关联对象的类型可以通过参数传入。GraphX优化了原生数据类型（如int,double等）关联对象的表示方式，将它们存储在专用数组里，减少了内存使用。有时候我们希望同一个图里的顶点可以有不同的属性类型，这可以通过继承来实现。例如，我们对用户和产品建立一个二部图模型，代码如下：

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

与RDD一样，属性图是不可变的、分布的和可容错的。改变图的值或结构会产生一个相应的新图。新图与原图之间会共享很多数据，包括未改变的结构、属性和索引等。我们是使用一组顶点分区启发法来对图在Spark的执行器（可能有多个执行器）间进行分区的。就像RDD，如果某个主机出现异常，这个主机上的图分区是可以在另一台健康的主机上重新创建的。

从逻辑上看，属性图对应于两个类型化集合（RDD），分别是顶点和边的属性集。相应的，图类（class）包含用于访问顶点和边的成员变量：

class Graph[VD, ED] {

 val vertices: VertexRDD[VD]

 val edges: EdgeRDD[ED]

}

类VertexRDD[VD]和EdgeRDD[ED]分别继承和优化了RDD[(VertexID, VD)]和RDD[Edge[ED]]。VertexRDD[VD]和EdgeRDD[ED]都提供了额外的功能用于图计算，更好的利用了内部优化。

## 属性图示例

假设我们要对GraphX项目中的协作者建立一个属性图模型，顶点的属性包含用户名和职业，边的属性表示协作者间的关系：



这个图的类型标识如下：

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

下面的代码从一个RDD的集合里构建一个图：

// 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)

在上例中，我们使用了Edge类，Edge类有一个srcId和一个dstId，分别对应于源顶点和目标顶点。Edge类还有一个attr变量用于存储边的属性。

通过调用 graph.vertices和graph.edges，我们可以将图拆解为顶点和边的视图，如：

 val graph: Graph[(String, String), String] // Constructed from above

// Count all users which are postdocs

graph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count

// Count all the edges where src > dst

graph.edges.filter(e => e.srcId > e.dstId).count

需要注意的是，graph.vertices返回的是RDD[(VertexID, (String, String))]的子类VertexRDD[(String, String)]，我们需要使用Scala的case表达式来解析这个元组。graph.edges返回的是一个包含Edge[String]对象的EdgeRDD。我们也可以使用case的class类型构造器对graph.edges的返回值进行构造，如下：

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

除了顶点和边的视图，GraphX还提供了一个triplet视图。triplet视图关联了顶点和边，形成一个包含类EdgeTriplet的实例的RDD[EdgeTriplet[VD, ED]]。这种关联可以用下面的SQL表示：



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

类EdgeTriplet扩展了类Edge，增加了成员变量srcAttr和dstAttr，分别用于保存源顶点和目标顶点的属性。我们可以利用triplet视图来获取用户之间的关系：

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(\_))

# 图操作符

就像RDD有map、filter和reduceByKey这样的基本操作，属性图也有一组基本操作符，这些操作符使用用户定义的函数，通过属性转换和结构转换来产生新图。优化过的核心操作符都定义在Graph里，GraphOps里定义了一些核心操作符的组合形式操作符，便于使用。但在Scala里，GraphOps里的操作符都可以作为Graph的成员来使用的。例如，我们可以通过以下代码计算每个顶点的入度（in-degree，在GraphOps里定义）：

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

// Use the implicit GraphOps.inDegrees operator

val inDegrees: VertexRDD[Int] = graph.inDegrees

区分核心Graph操作符和GraphOps操作符是为了将来可以支持不同的图表示方法。每种图表示方法必须实现核心Graph操作符，重用很多在GraphOps里定义的操作符。

## 操作符列表

下面列举了Graph和GraphOps中定义的一些操作符，为了便于理解，函数表达式作了简化：

/\*\* 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]

}

# Pregel API

图本身是递归的数据结构，顶点的属性依赖于邻居的属性，而邻居的属性又依赖于他们的邻居的属性。所以很多重要的图算法都是对每个顶点的属性进行迭代运算，直到某个固定条件出现。一系列图并行的抽象概念被提了出来，用于表达这些迭代算法。GraphX提供了另一种形式的Pregel API。

总体来看，GraphX中的Pregel操作符是一种应用在拓扑图中的整体的、同步的、并行的消息计算模型。Pregel操作符是在一系列的超步里执行的，在超步里，顶点接收来自上一个超步里的消息汇总，计算出一个新的顶点属性值，然后发送消息到下一个超步中的邻居顶点。与Pregel不同，消息的并行计算已经被封装成了Edge Triplet的一个函数，计算过程中可以访问源顶点和目标顶点的属性。如果一个顶点在超步里不接收消息，这个顶点将不参与计算。在所有的消息都处理完成后，Pregel操作符会结束迭代并返回一个新图。

与标准的Pregel实现相比，GraphX中的顶点只能给邻居顶点发送消息，消息是使用用户定义的消息函数来构建的。有了这些限制，GraphX就可以对Pregel API做进一步的优化。下面是Pregel操作符的表达式说明：

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 and update the vertices.

 g = g.joinVertices(messages)(vprog).cache()

 val oldMessages = messages

 // Send new messages, skipping edges where neither side received a message. We must cache

 // messages so it can be materialized on the next line, allowing us to uncache the previous

 // iteration.

 messages = g.mapReduceTriplets(

 sendMsg, mergeMsg, Some((oldMessages, activeDirection))).cache()

 activeMessages = messages.count()

 i += 1

 }

 g

 }

}

# Graph构建器

GraphX提供了几种图的构建方法，可以使用RDD或磁盘中的顶点和边的集合进行构建。缺省情况下，每个构建器都不会对图的边进行再分区，边是保存在它们的默认分区里的，如HDFS中的数据块。Graph.groupEdges方法是假定相同的边是保存在相同的分区里的，所以在调用这个方法前需要调用Graph.partitionBy进行再分区。

object GraphLoader {

 def edgeListFile(

 sc: SparkContext,

 path: String,

 canonicalOrientation: Boolean = false,

 minEdgePartitions: Int = 1)

 : Graph[Int, Int]

}

通过解析下例中的源顶点和目标顶点的数据对，GraphLoader.edgeListFile可以从磁盘上的边列表里加载一个图，以#开头的注释行忽略不计。相应的顶点是自动创建的，所有的顶点和边的属性都是1。canonicalOrientation参数可以对边做重定向（定向为正向，srcId < dstId），这个功能是连通图算法需要的。minEdgePartitions参数指定了边的最小分区数，实际上可以有多于这个参数的分区，如HDFS文件有多个块。

# This is a comment

2 1

4 1

1 2

object Graph {

 def apply[VD, ED](

 vertices: RDD[(VertexId, VD)],

 edges: RDD[Edge[ED]],

 defaultVertexAttr: VD = null)

 : Graph[VD, ED]

 def fromEdges[VD, ED](

 edges: RDD[Edge[ED]],

 defaultValue: VD): Graph[VD, ED]

 def fromEdgeTuples[VD](

 rawEdges: RDD[(VertexId, VertexId)],

 defaultValue: VD,

 uniqueEdges: Option[PartitionStrategy] = None): Graph[VD, Int]

}

Graph.apply是从顶点和边的RDD中构建图的。如果顶点RDD中有重复的顶点，则会选择其中的任意一个来使用。如果一个顶点在边的RDD里并且不在顶点的RDD里，这个顶点会被赋一个默认值。

Graph.fromEdges可以只从边的RDD中构建图，相应的顶点会被自动创建并赋一个默认值。

Graph.fromEdgeTuples可以从边元组的RDD中构建图，设定边的值为1，自动创建相应的顶点并赋一个默认值。这个方法支持边的去重，可传入一个PartitionStrategy给uniqueEdges（如uniqueEdges = Some(PartitionStrategy.RandomVertexCut)）。对于去重来说，分区策略可以保证相同的边保存在相同的分区里，这样才能实现去重。

# 顶点和边的RDD

GraphX提供了图的顶点和边的视图。GraphX把顶点和边保存在优化的数据结构里，这些数据结构提供了额外的功能，返回的RDD分别是VertexRDD和EdgeRDD。

## VertexRDD

VertexRDD[A]扩展了RDD[(VertexID, A)]，规定每个VertexID只能出现一次。VertexRDD[A]表示一个类型为A的顶点集合。在内部实现上，顶点属性是存储在一个可重用的Hash Map里的。这样的话，如果两个VertexRDD来自于同一个VertexRDD（通过filter或mapValues），这两个RDD可以在常数时间内完成Join操作，不需要重新计算哈希值。

class VertexRDD[VD] extends RDD[(VertexID, VD)] {

 // Filter the vertex set but preserves the internal index

 def filter(pred: Tuple2[VertexId, VD] => Boolean): VertexRDD[VD]

 // Transform the values without changing the ids (preserves the internal index)

 def mapValues[VD2](map: VD => VD2): VertexRDD[VD2]

 def mapValues[VD2](map: (VertexId, VD) => VD2): VertexRDD[VD2]

 // Show only vertices unique to this set based on their VertexId's

 def minus(other: RDD[(VertexId, VD)])

 // Remove vertices from this set that appear in the other set

 def diff(other: VertexRDD[VD]): VertexRDD[VD]

 // Join operators that take advantage of the internal indexing to accelerate joins (substantially)

 def leftJoin[VD2, VD3](other: RDD[(VertexId, VD2)])(f: (VertexId, VD, Option[VD2]) => VD3): VertexRDD[VD3]

 def innerJoin[U, VD2](other: RDD[(VertexId, U)])(f: (VertexId, VD, U) => VD2): VertexRDD[VD2]

 // Use the index on this RDD to accelerate a `reduceByKey` operation on the input RDD.

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

}

## EdgeRDD

EdgeRDD[ED]扩展了RDD[Edge[ED]]，将边存储在依据PartitionStrategy定义的策略进行分区的数据块里。在每个分区里，边的属性和关联结构是分开存储的，保证了属性变更时的最大重用性。

// Transform the edge attributes while preserving the structure

def mapValues[ED2](f: Edge[ED] => ED2): EdgeRDD[ED2]

// Revere the edges reusing both attributes and structure

def reverse: EdgeRDD[ED]

// Join two `EdgeRDD`s partitioned using the same partitioning strategy.

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

# 优化表示

在边切割和点切割两种方案里，GraphX采用了顶点切割的分布式分区方案：



相对于沿着边对图进行切割，GraphX沿着顶点对图进行切割分区，这降低了通讯和存储的开销。逻辑上看，是将边分配到主机，点要横跨多个主机。实际分配过程依赖于PartitionStrategy（分区策略），要根据实际情况在不同的策略间进行权衡，通过Graph.partitionBy来设定。默认的分区策略是使用图构建过程中采用的边分区策略。用户可以改变默认策略。做完分区之后的最大挑战是如何高效的关联边和顶点属性。在真实场景里，相对于顶点，图通常有更多的边，所以我们通常移动顶点属性到边所有的主机。因为不是所有的分区都包含关联到某些点的边，我们维护了一个路由表来标识与顶点有关联的存储边的主机。



# 图算法

GraphX包含了一组算法来简化分析过程，这些算法包含在org.apache.spark.graphx.lib中，可通过Graph类直接访问。

## PageRank

假设从u到v的边代表u对v的一种加权，则PageRank算法可用来度量图中每个顶点的重要性或权重。例如，一个有很多粉丝的Twitter用户将有很高的排位。

PageRank对象包含动态和静态的算法实现。静态算法运行固定的迭代次数，而动态算法则会一直运算，直到排位汇集为止。

graphx/data/users.txt

---------------------------------

1,BarackObama,Barack Obama

2,ladygaga,Goddess of Love

3,jeresig,John Resig

4,justinbieber,Justin Bieber

6,matei\_zaharia,Matei Zaharia

7,odersky,Martin Odersky

8,anonsys

graphx/data/followers.txt

---------------------------------

2 1

4 1

1 2

6 3

7 3

7 6

6 7

3 7

// 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"))

## 连通图

连通图算法为图中每一个连通图关联一个ID，这个ID是连通图中最小的顶点ID。例如一个社交网络中的连通图可以看作是一个群。ConnectedComponents对象包含这个算法，代码如下：

// 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"))

## 三角计算

当一个顶点的两个相邻顶点之间有一条关联边时，我们说这个顶点是三角形的一部分。TriangleCount对象包含三角形计算的算法，可以计算出通过每个顶点的三角形个数，提供一种群的度量方式。TriangleCount要求边是正向的(srcId < dstId)，而且图要使用Graph.partitionBy进行分割。

// Load the edges in canonical order and partition the graph for triangle count

val graph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt", true).partitionBy(PartitionStrategy.RandomVertexCut)

// Find the triangle count for each vertex

val triCounts = graph.triangleCount().vertices

// Join the triangle counts with the usernames

val users = sc.textFile("graphx/data/users.txt").map { line =>

 val fields = line.split(",")

 (fields(0).toLong, fields(1))

}

val triCountByUsername = users.join(triCounts).map { case (id, (username, tc)) =>

 (username, tc)

}

// Print the result

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

# 实例

附件包含的实例实现上面的三个算法，具体代码参考附件中的源代码。

## 下载测试数据

<http://snap.stanford.edu/data/soc-LiveJournal1.txt.gz>是一个社交网站的真实数据，里面包含边的数据，即源顶点和目标顶点数据。

## 运行

对于PageRank:

spark-submit --class org.apache.spark.examples.graphx.LiveJournalPageRank --master yarn-cluster --deploy-mode cluster --num-executors 3 --driver-memory 500m --executor-memory 500m --executor-cores 1 /tmp/Spark-Graphx-1.0-SNAPSHOT.jar /tmp/soc-LiveJournal2.txt --numEPart=3

对于连通图:

spark-submit --class org.apache.spark.examples.graphx.Analytics --master yarn-cluster --deploy-mode cluster --num-executors 3 --driver-memory 500m --executor-memory 500m --executor-cores 1 /tmp/Spark-Graphx-1.0-SNAPSHOT.jar cc /tmp/soc-LiveJournal2.txt --numEPart=3

对于三角形计算:

spark-submit --class org.apache.spark.examples.graphx.Analytics --master yarn-cluster --deploy-mode cluster --num-executors 3 --driver-memory 500m --executor-memory 500m --executor-cores 1 /tmp/Spark-Graphx-1.0-SNAPSHOT.jar triangles /tmp/soc-LiveJournal2.txt --numEPart=3

## 结果确认

打开yarn的History服务器，查看作业日志。