Creating Custom Graph Aggregation Operators

# 创建自定义图聚合操作符

黄字是不确定，红字是备注。

Node通常翻译成顶点，有时候翻译成节点我觉得更好听，要统一一下。

还有几个类似情况的词。

In the previous chapter, we have seen various operations for transforming the elements of a graph and for modifying its structure.

在前一章中，我们已经学习了许多不同的操作，以从一个图中转化元素并修改它的结构。

Here, we will learn to use a generic and powerful operator named **aggregateMessages** that is useful for aggregating the neighborhood information of all nodes in the graph.

本章我们将学习强有力的通用操作符aggregateMessages，它在聚合图中所有节点的邻居信息时十分有用。

In fact, many graph-processing algorithms rely on iteratively accessing the properties of neighboring nodes and adjacent edges. One such example is the PageRank algorithm.

实际上，许多图处理算法依赖于迭代访问相邻节点和相邻边的属性。PageRank算法就是一个典型的例子。

By applying **aggregateMessages** to the NCAA College Basketball datasets, you will be able to:

通过将**aggregateMessages应用于**NCAA大学篮球数据集，你将能够：

* Understand the basic mechanisms and patterns of **aggregateMessages**

理解aggregateMessages的基本机制和模式

* Apply it to create custom graph aggregation operations

应用它创建自定义图聚合操作

* Optimize the performance and efficiency of **aggregateMessages**

优化aggregateMessages的性能和效率

**NCAA College Basketball datasets**

We will again learn by doing in this chapter. This time, we will take the NCAA College Basketball as an illustrative example.

我们将在本章中继续学习具体操作。这一次，我们使用NCAA大学数据集作为讲解示例。

Specifically, we use two CSV datasets. The first one **teams.csv** contains the list of all college teams that played in the NCAA Division I competition. Each team is associated with a four-digit ID number.

具体来说，我们使用了两个CSV数据集。第一个数据集teams.csv包含了一张列表，列表中是所有参与NCAA第一级别联赛的大学篮球队。每支球队用一个四位数的ID号表示。

The second dataset **stats.csv** contains the score and statistics of every game during the 2014-2015 regular season.

第二个数据集stats.csv包含了2014-2015年常规赛季每场比赛的比分和统计数据。

Using the techniques learned in *Chapter 2*, *Building and Exploring Graphs*, let's parse and load these datasets and load them into RDDs:

使用在第二章“构建和探索图”中学到的技术，让我们分析并加载这些数据集到RDDs中：让我们加载这些数据集到RDD中进行分析：

1. We create a class **GameStats** that records the statistics of one team during a specific basketball game:

我们创建一个GameStats类，记录特定比赛中一支球队的统计数据：

**case class GameStats(**

**val score: Int,**

**val fieldGoalMade: Int,**

**val fieldGoalAttempt: Int,**

**val threePointerMade: Int,**

**val threePointerAttempt: Int,**

**val threeThrowsMade: Int,**

**val threeThrowsAttempt: Int,**

**val offensiveRebound: Int,**

**val defensiveRebound: Int,**

**val assist: Int,**

**val turnOver: Int,**

**val steal: Int,**

**val block: Int,**

**val personalFoul: Int**

**)**

1. We also add the following methods to **GameStats** in order to know how efficient a team's offense was during a game:

我们添加以下方法到GameStats类中，以了解球队在比赛中的进攻效率：

**// 投篮命中率Field Goal percentage**

**def fgPercent: Double = 100.0 \* fieldGoalMade / fieldGoalAttempt**

**// 三分球命中率Three Point percentage**

**def tpPercent: Double = 100.0 \* threePointerMade / threePointerAttempt**

**// 罚球命中率Free throws percentage**

**def ftPercent: Double = 100.0 \* freeThrowsMade / freeThrowsAttempt**

**override def toString: String = "Score: " + score**

1. We now create a couple of classes for the games' result:

我们现在为比赛的结果创建两个类：

**Abstract classGame Result(**

**val season: Int,**

**val day: Int,**

**val loc: String**

**)**

**case class FullResult(**

**override val season: Int,**

**override val day: Int,**

**override val loc: String,**

**val winnerStats: ameStats,**

**val loserStats: GameStats**

**) extends GameResult(season, day, loc)**

**FullResult** has the year and day of the season, the location where the game was played, and the game statistics of both the winning and losing teams.

FullResult类包含本赛季的年份和日期，比赛的场地以及胜负双方球队的比赛统计数据。

1. We will then create a statistics graph of the regular seasons. In this graph, the nodes are the teams, whereas each edge corresponds to a specific game. To create the graph, let's parse the CSV file **teams.csv** into the RDD teams:

**然后我们创建一个常规赛季的统计图。在该图中，节点是球队，每条边对应一场特定的比赛。为了创建图，让我们在RDD组中分析CSV文件teams.csv：**

**val teams: RDD[(VertexId, String)] =**

**sc.textFile("./data/teams.csv").**

**filter(! \_.startsWith("#")).**

**map {line =>**

**val row = line split ','**

**(row(0).toInt, row(1))**

**}**

1. We can check the first few teams in this new RDD:

我们可以在这个新的RDD中查看前几个球队：

**scala> teams.take(3).foreach{println}**

**(1101,Abilene Chr)**

**(1102,Air Force)**

**(1103,Akron)**

1. We do the same thing to obtain an RDD of the game results, which will have a type **RDD[Edge[FullResult]]**. We just parse **stats.csv** and record the fields that we need— the ID of the winning team, the ID of the losing team, and the game statistics of both teams:

我们以同样的方法来获得比赛结果的RDD，其中RDD的类型是[Edge [FullResult]]。 我们仅仅分析stats.csv并记录我们需要的字段——胜者球队的ID，败者球队的ID以及两队的比赛统计数据：

**val detailedStats: RDD[Edge[FullResult]] = sc.textFile("./data/stats.csv").**

**filter(! \_.startsWith("#")).**

**map {line =>**

**val row = line split ','**

**Edge(row(2).toInt, row(4).toInt,**

**FullResult(**

**row(0).toInt, row(1).toInt,**

**row(6),**

**GameStats(**

**score = row(3).toInt, fieldGoalMade = row(8).toInt,**

**fieldGoalAttempt = row(9).toInt,**

**threePointerMade = row(10).toInt,**

**threePointerAttempt = row(11).toInt,**

**threeThrowsMade = row(12).toInt,**

**threeThrowsAttempt = row(13).toInt,**

**offensiveRebound = row(14).toInt,**

**defensiveRebound = row(15).toInt,**

**assist = row(16).toInt,**

**turnOver = row(17).toInt,**

**steal = row(18).toInt,**

**block = row(19).toInt,**

**personalFoul = row(20).toInt**

**),**

**GameStats(**

**score = row(5).toInt,**

**fieldGoalMade = row(21).toInt,**

**fieldGoalAttempt = row(22).toInt,**

**threePointerMade = row(23).toInt,**

**threePointerAttempt = row(24).toInt,**

**threeThrowsMade = row(25).toInt,**

**threeThrowsAttempt = row(26).toInt,**

**offensiveRebound = row(27).toInt,**

**defensiveRebound = row(28).toInt,**

**assist = row(20).toInt,**

**turnOver = row(30).toInt,**

**steal = row(31).toInt,**

**block = row(32).toInt,**

**personalFoul = row(33).toInt**

**)**

**)**

**)**

**}**

**Let's check what we have got:**

**让我们查看下我们获得了什么：**

**scala> detailedStats.take(3).foreach(println) Edge(1165,1384,FullResult(2006,8,N,Score: 75-54))**

**Edge(1393,1126,FullResult(2006,8,H,Score: 68-37))**

**Edge(1107,1324,FullResult(2006,9,N,Score: 90-73))**

1. We then create our graph of stats:

然后我们创建自己的统计图**scoreGraph**：

**scala> val scoreGraph = Graph(teams,detailedStats)**

For curiosity, let's see which team has won against the 2015 NCAA champions Duke in the regular season. To do that, we filter the graph triplets whose destination attribute is **Duke**. This is because when we created our stats graph, each edge is directed from the winner node to the loser node. So, Duke has lost only four games in the regular season:

出于好奇，让我们看看哪支球队在常规赛季中击败过2015年NCAA的冠军Duke队。为此，我们过滤下图中目的顶点是Duke的那些triplets。这是因为在我们创建统计图时，每条边都是从胜者顶点导向败者顶点。所以，我们得知Duke在常规赛季仅仅输了四场。

**scala> scoreGraph.triplets.filter(\_.dstAttr == "Duke").foreach(println) ((1274,Miami FL),(1181,Duke),FullResult(2015,71,A,Score: 90-74))**

**((1301,NC State),(1181,Duke),FullResult(2015,69,H,Score: 87-75))**

**((1323,Notre Dame),(1181,Duke),FullResult(2015,86,H,Score: 77-73))**

**((1323,Notre Dame),(1181,Duke),FullResult(2015,130,N,Score: 74-64))**

**The aggregateMessages operator**

# aggregateMessages操作符

Once we have our graph ready, let's start our mission, which is aggregating the stats data in **scoreGraph**. In GraphX, **aggregateMessages** is the operator for that kind of job.

当我们的图创建好了，就可以在图中开始聚合统计数据的任务了。在GraphX中，此类工作是由**aggregateMessages操作符完成的。**

For example, let's find out the average field goals made per game by the winning teams. In other words, the games that the teams lost will not be counted. To get the average for each team, we first need to have the number of games won by the team and the total field goals that the team made in those games:

举个例子，让我们看看获胜球队每场比赛的平均投篮命中数。换句话说，球队输掉的比赛将不会被计算在内。为了获得每个团队的平均命中数，首先，我们需要获得球队获胜的比赛数量，以及球队在这些比赛中所获得的总投篮命中数:

**// Aggregate the total field goals made by winning teams type**

**汇总获胜球队总的投篮命中数**

**FGMsg = (Int, Int)**

**val winningFieldGoalMade: VertexRDD[FGMsg] = scoreGraph**

**aggregateMessages(**

**// sendMsg**

**triplet => triplet.sendToSrc(1, triplet.attr.winnerStats.fieldGoalMade),**

**// mergeMsg**

**(x, y) => (x.\_1 + y.\_1, x.\_2+ y.\_2)**

**)**

**// Aggregate the total field goals made by winning teams type**

**汇总获胜球队总的投篮命中数**

**Msg = (Int, Int)**

**type Context = EdgeContext[String, FullResult, Msg]**

**val winningFieldGoalMade: VertexRDD[Msg] = scoreGraph**

**aggregateMessages(**

**// sendMsg**

**(ec: Context) => ec.sendToSrc(1, ec.attr.winnerStats.fieldGoalMade),**

**// mergeMsg**

**(x: Msg, y: Msg) => (x.\_1 + y.\_1, x.\_2+ y.\_2)**

**)**

（这代码一句太长了，它分了几行，在其中加了几处注释，代码我没看懂，注释我觉得可能有点问题）

#### EdgeContext

There is a lot going on in the previous call to **aggregateMessages**. So, let's see it working in slow motion. When we called **aggregateMessages** on the **scoreGraph** method, we had to pass two functions as arguments.

在调用**aggregateMessages之前有很多事情要做。所以，让我们通过慢镜头再看一遍它的工作流程。**当我们在scoreGraph方法上调用aggregateMessages时，我们必须传递两个函数作为参数。

The first function has a signature **EdgeContext[VD, ED, Msg] => Unit**. It takes an **EdgeContext** parameter as input. It does not return anything but it can produce side effects, such as sending a message to a node.

第一个函数具有签名EdgeContext [VD，ED，Msg] => Unit。它需要一个EdgeContext参数作为输入。该函数不会返回任何内容，但会产生副作用，例如向节点发送消息。

Ok, but what is that **EdgeContext** type? Similar to **EdgeTriplet**, **EdgeContext** represents an edge along with its neighboring nodes. It can access both the edge attribute, and the source and destination nodes' attributes. In addition, **EdgeContext** has two methods to send messages along the edge to its source node or to its destination node. These methods are **sendToSrc** and **sendToDst** respectively. Then, the type of message that we want each triplet in the graph to send is defined by **Msg**. Similar to **VD** and **ED**, we can define the concrete type that **Msg** takes.

但是，**EdgeContext的类型是什么呢？与EdgeTriplet相似，EdgeContext表示一条边以及它相邻的顶点。它可以访问边的属性，以及源顶点与目的顶点的属性。此外，EdgeContext有两种方法可以沿着边传递消息到它的源顶点或者目的顶点。这两种方法分别是sendToSrc**和**sendToDst。然后，我们希望图中每一个triplet传递的消息的类型由Msg定义。与VD和ED类似，我们可以定义Msg的具体类型。**

In our example, we need to aggregate the number of games played and the number of field goals made. Therefore, we define **Msg** as a pair of **Int**.

Furthermore, each edge context sends a message to only its source node, that is the winning team, because we are interested in the total field goals made by the teams for only the games that they won.

**在我们的例子中，我们需要汇总球赛的数量和总的投篮命中数。因此，我们定义Msg的类型为一对Int。此外，因为我们仅仅对球队在其获胜球赛中的总投篮命中数感兴趣，所以每一个edgecontext仅向它的源顶点（即获胜球队）发送一条信息。**

The actual message sent to each winner node is a pair of integers **(1, ec.attr.winnerStats.fieldGoalMade)**. The first integer serves as a counter for the games won by the source node, whereas the second one corresponds to the number of field goals made by the winner. This latter integer is then extracted from the edge attribute.

发送给每个获胜节点的具体消息是一对整数(1, ec.attr.winnerStats.fieldGoalMade)。第一个整数是源节点所赢球赛数量的计数器，而第二个整数是相应的获胜比赛中球队的投篮命中数。后者是从边属性中抽取出来的一个整数。

In addition to **sendMsg**, the second function that we need to pass to **aggregateMessages** is a **mergeMsg** function with the signature **(Msg, Msg) => Msg**. As its name implies, **mergeMsg** is used to merge two messages received at each node into a new one. Its output type must be the same, for example **Msg**. Using these two functions, **aggregateMessages** returns the aggregated messages inside **VertexRDD[Msg]**.

除了sendMsg函数之外，我们需要传递给aggregateMessages的第二个函数是带有签名(Msg, Msg) => Msg的mergeMsg函数。顾名思义，mergeMsg函数用于将每个节点接收到的两个消息合并为一个新消息。它的输出类型必须是与之相同的，例如Msg。使用这两个函数，aggregatemessage返回VertexRDD[Msg]中的聚合消息。

Returning to our example, we set out to compute the average field goals per winning game for all teams. To get this final result, we simply apply **mapValues** to the output of **aggregateMessages**, as follows:

回到我们的例子，我们开始计算所有球队在每场获胜比赛中的平均投篮得分。为了得到最终结果，我们只需将mapValues应用于aggregatemessage的输出，如下所示:

**// Average field goals made per Game by winning teams**

**获胜球队每场比赛的平均投篮命中数**

**val avgWinningFieldGoalMade: VertexRDD[Double] =**

**winningFieldGoalMade mapValues (**

**(id: VertexId, x: Msg) => x match {**

**case (count: Int, total: Int) => total.toDouble/count**

**})**

Let's check the output:

我们查看下输出：

**scala> avgWinningFieldGoalMade.take(5).foreach(println) (1260,24.71641791044776)**

**(1410,23.56578947368421)**

**(1426,26.239436619718308)**

**(1166,26.137614678899084)**

**(1434,25.34285714285714)**

The definitions of **aggregateMessages** and **EdgeContext**, as we explained previously, are shown as follows:

正如我们前面所解释的，aggregateMessages和EdgeContext的定义如下所示：

**class Graph[VD, ED] {**

**def aggregateMessages[Msg:ClassTag](**

**sendMsg: EdgeContext[VD, ED, Msg] => Unit,**

**mergeMsg: (Msg, Msg) => Msg,**

**tripletFields: TripletFields = TripletFields.All)**

**: VertexRDD[Msg]**

**}**

**abstract class EdgeContext[VD, ED, A] {**

**// Attribute associated with the edge:**

边关联的属性：

**abstract def attr: ED**

**// Vertex attribute of the edge's source vertex.**

**边的源顶点的顶点属性**

**abstract def srcAttr: VD**

**// Vertex attribute of the edge's destination vertex.**

**边的目的顶点的顶点属性**

**abstract def dstAttr: VD**

**// Vertex id of the edge's source vertex.**

**边的源顶点的顶点ID**

**abstract def srcId: VertexId**

**// Vertex id of the edge's destination vertex.**

**边的目的顶点的顶点ID**

**abstract def dstId: VertexId**

**// Sends a message to the destination vertex.**

**向目的顶点发送一条信息**

**abstract def sendToDst(msg: A): Unit**

**// Sends a message to the source vertex.**

**向源顶点发送一条信息**

**abstract def sendToSrc(msg: A): Unit**

**}**

#### Abstracting out the aggregation

That was kinda cool! We can do the same to average the points per game scored by winning teams:

这太酷了！我们也可以同样去计算获胜球队每场比赛的平均得分：

**// Aggregate the points scored by winning teams**

**汇总获胜球队总的投篮得分**

**val winnerTotalPoints: VertexRDD[(Int, Int)] = scoreGraph.aggregateMessages(**

**// sendMsg**

**triplet => triplet.sendToSrc(1, triplet.attr.winnerStats.score),**

**// mergeMsg**

**(x, y) => (x.\_1 + y.\_1, x.\_2+ y.\_2)**

**)**

**// Average** **field goals made per Game by winning teams**

**此处我认为是：**

**Average** **points scored per Game by winning teams**

**不过意思一样，field goals，field goals made，points scored分不太清楚**

**获胜球队每场比赛的平均投篮得分数**

**var winnersPPG: VertexRDD[Double] =**

**winnerTotalPoints mapValues (**

**(id: VertexId, x: (Int, Int)) => x match{**

**case (count: Int, total: Int) =>**

**total.toDouble/count**

**})**

Let's check the output:

我们查看下输出：

**scala> winnersPPG.take(5).foreach(println) (1260,71.19402985074628)**

**(1410,71.11842105263158)**

**(1426,76.30281690140845)**

**(1166,76.89449541284404)**

**(1434,74.28571428571429)**

Now, the coach wants us to list the top five teams with the highest average three-pointer made per winning game. By the way, he also wants to know which teams are the most efficient in three-pointers.

现在，教练希望我们列出在每场获胜球赛中，平均三分球得分前五的球队。同时，他还想知道哪些支球队的三分球进攻最有效率的。

#### Keeping things DRY

We can copy and modify the previous code but that would be repetitive. Instead, let's abstract out the average aggregation operator so that it can work on any statistics that the coach needs. Luckily, Scala's higher-order functions are there to help in this task.

我们可以复制并修改以前的代码，但这是重复的。相反的话，如果我们抽象出平均聚合运算符，那么它可以处理教练需要的任何统计数据。幸运的是，Scala的高阶功能可以帮助完成这项任务。

For each statistic that our coach wants, let's define a function that takes a team's **GameStats** as input and returns the statistic that we are interested in. For now, we will need the number of three-pointers made and the average three-pointer percentage:

我们为每一个教练需要的统计数据定义一个函数，它将每个球队的GameStats类作为输入，并返回我们感兴趣的统计数据。现在，我们需要三分球命中数和平均三分球命中率。

**// Getting individual stats**

**获得各自的统计数据**

**def threePointMade(stats: GameStats) =**

**stats.threePointerMade**

**def threePointPercent(stats: GameStats) = stats.tpPercent**

Then, we create a generic function that takes as inputs a stats graph and one of the functions defined previously, which has a signature **GameStats => Double**:

然后，我们创建一个通用函数，它将一个统计图和一个先前定义的函数作为输入，该函数具有签名GameStats => Double：

**// Generic function for stats averaging**

**用于统计平均值的通用函数**

**def averageWinnerStat(graph: Graph[String, FullResult])(getStat: GameStats => Double): VertexRDD[Double] = {**

**type Msg = (Int, Double)**

**val winningScore: VertexRDD[Msg] =**

**graph.aggregateMessages[Msg](**

**// sendMsg**

**triplet => triplet.sendToSrc(1, getStat(triplet.attr.winnerStats)),**

**// mergeMsg**

**(x, y) => (x.\_1 + y.\_1, x.\_2+ y.\_2)**

**)**

**winningScore mapValues (**

**(id: VertexId, x: Msg) => x match {**

**case (count: Int, total: Double) => total/count**

**})**

**}**

Then, we can use the average stats by passing the functions **threePointMade** and **threePointPercent** to **averageWinnerStat**:

接着，我们可以通过将threePointMade 和 threePointPercent两个函数传递给averageWinnerStat来使用average stats（统计数据的平均值？）：

**val winnersThreePointMade =**

**averageWinnerStat(scoreGraph)(threePointMade)**

**val winnersThreePointPercent = averageWinnerStat(scoreGraph)(threePointPercent)**

With little effort, we can tell the coach which five winning teams scored the highest number of threes per game:

几乎毫不费力地，我们就可以告诉教练哪五支获胜球队的平均胜场三分球数最高：

**scala> winnersThreePointMade.sortBy(\_.\_2,false).take(5).foreach(println) (1440,11.274336283185841)**

**(1125,9.521929824561404)**

**(1407,9.008849557522124)**

**(1172,8.967441860465117)**

**(1248,8.915384615384616)**

While we are at it, let's find out the five most efficient teams in three-pointers:

与此同时，让我们找出三分球进攻最高效的五支球队：

**scala> winnersThreePointPercent.sortBy(\_.\_2,false).take(5). foreach(println)**

**(1101,46.90555728464225)**

**(1147,44.224282479431224)**

**(1294,43.754532434101534)**

**(1339,43.52308905887638)**

**(1176,43.080814169045105)**

Interestingly, the teams that made the most three-pointers per winning game are not always the ones who are the most efficient at it. But, they still won those games, which is more important.

有趣的是，每场比赛三分球最多的球队并不总是那些三分球效率最高的球队。但是，他们仍然赢得了这些比赛，这是更重要的。

#### Coach wants more numbers

教练想要更多的信息

The coach seems unsatisfied with that argument and wants us to get the same statistics but wants us to average them over all the games that each team has played.

教练似乎对这个结果不满意，并希望我们给出相同的统计数据，有所不同的是，希望我们对每个球队所参加的所有比赛进行平均。

Thus, we have to aggregate the information from all the nodes of our graph, and not only at the destination nodes. To make our previous abstraction more flexible, let's create the following types:

因此，我们必须从图的所有节点聚合信息，而不仅仅从目标节点处。为了使我们以前的抽象更灵活，我们创建以下类型:

**trait Teams**

**case class Winners extends Teams**

**case class Losers extends Teams**

**case class AllTeams extends Teams**

We modify the previous higher-order function to have an extra argument **Teams**, which will help us specify at which nodes we want to collect and aggregate the required game stats. The new function becomes:

我们修改了之前的高阶函数来增加一个额外的参数组，该参数组将帮助我们指定我们想要收集哪些节点，并辅助聚合所需的赛事统计数据。新的函数修改如后下：

**def averageStat(graph: Graph[String, FullResult])(getStat:**

**GameStats => Double, tms: Teams): VertexRDD[Double] = {**

**type Msg = (Int, Double)**

**val aggrStats: VertexRDD[Msg] = graph.aggregateMessages[Msg](**

**//sendMsg**

**tms match {**

**case \_ : Winners => t => t.sendToSrc((1,**

**getStat(t.attr.winnerStats)))**

**case \_ : Losers => t => t.sendToDst((1,**

**getStat(t.attr.loserStats)))**

**case \_ => t => {**

**t.sendToSrc((1, getStat(t.attr.winnerStats)))**

**t.sendToDst((1, getStat(t.attr.loserStats)))**

**} }**

**,**

**// mergeMsg**

**(x, y) => (x.\_1 + y.\_1, x.\_2+ y.\_2)**

**)**

**aggrStats mapValues (**

**(id: VertexId, x: Msg) => x match{**

**case(count:Int,total:Double) => total/count**

**})**

**}**

这段是无脑翻译的，还没有联系上下文修改。

Compared to **averageWinnerStat**, **aggregateStat** allows us to choose whether we want to aggregate the stats for winners only, for losers only, or for all teams.

与**averageWinnerStat相比，aggregateStat允许我们选择单独聚合获胜者或者失败者的统计数据，或者选择聚合所有球队的统计数据。**

Since the coach wants the overall stats averaged over all games played, we aggregate the stats by passing the **AllTeams()** flag in **aggregateStat**.

由于教练想要的是球队参加的所有赛事的统计数据均值，我们通过在aggregateStat中传递Allteam()标志来聚合统计数据。

In this case, we simply define the **sendMsg** argument in **aggregateMessages** so that the required stats are sent to both the source (the winner) and to the destination (the loser) using the **EdgeContext** class's **sendToSrc** and **sendToDst** functions respectively.

在这种情况下，我们只需要在aggregatemessage中定义sendMsg参数，这样就可以分别使用EdgeContext类的sendToSrc和sendToDst函数将所需的统计信息发送给源节点(获胜者)和目的节点(失败者)。

This mechanism is pretty straightforward.

这个机制是非常清晰明了的。

We just need to make sure we send the right information to the right node. In this case, we send **winnerStats** to the winner and **loserStats** to the loser.

我们只需要确保我们发送了正确的信息给正确的节点。在这种情况下，我们把**winnerStats发送给获胜者，并把loserStats发给给失败者。**

Ok, you've got the idea now. So, let's apply it to please our coach.

好了，你现在已经学会了。所以，让我们用它来取悦我们的教练。

Here are the teams with the overall highest three-pointers per page:

这里没看懂这个per page，不知道是Scala的东西还是什么，下面有个per game，这两句话Here开头的话都没翻译好，three-pointers这个也不清楚翻译是否和three point一样

以下是每page所有赛事中平均三分球得分最高的球队:

**// Average Three Point Made Per Game for All Teams**

所有球队每场比赛的平均三分球得分

**val allThreePointMade = averageStat(scoreGraph)(threePointMade, AllTeams())**

Let's see the output:

让我们查看下输出：

**scala> allThreePointMade.sortBy(\_.\_2, false).take(5).foreach(println) (1440,10.180811808118081)**

**(1125,9.098412698412698)**

**(1172,8.575657894736842)**

**(1184,8.428571428571429)**

**(1407,8.411149825783973)**

Here are the five most efficient teams overall in three-pointers per game:

以下是每场比赛的三分球进攻最高效的五支球队:

**// Average Three Point Percent for All Teams**

**所有球队的平均三分球命中率**

**val allThreePointPercent = averageStat(scoreGraph)(threePointPercent, AllTeams())**

The output is:

它的输出为：

**scala> allThreePointPercent.sortBy(\_.\_2,false).take(5).foreach(println) (1429,38.8351815824302)**

**(1323,38.522819895594)**

**(1181,38.43052051444854)**

**(1294,38.41227053353959)**

**(1101,38.097896464168954)**

Actually, there is only a 2 percent difference between the most efficient team and the one in the fiftieth position. Most NCAA teams are therefore pretty efficient behind the line. I bet the coach knew that already!

实际上，进攻最高效的球队和排第五十的球队之间只有2％的差距。因此，大多数NCAA球队在这方面都非常有效率。我敢打赌教练已经知道了！

#### Calculating average points per game

计算每场比赛的平均得分

We can also reuse the **averageStat** function to get the average points per game for the winners. In particular, let's take a look at the two teams that won games with the highest and lowest scores:

我们还可以重用averageStat函数来获得获胜球队每场比赛的平均得分。具体来说，我们来看看以最高分和最低得分赢得比赛的两支球队:

**// Winning teams**

**获胜球队**

**val winnerAvgPPG = averageStat(scoreGraph)(score, Winners())**

Let's check the output:

让我们查看下输出：

**scala> winnerAvgPPG.max()(Ordering.by(\_.\_2))**

**res36: (org.apache.spark.graphx.VertexId, Double) = (1322,90.73333333333333)**

**scala> winnerAvgPPG.min()(Ordering.by(\_.\_2))**

**res39: (org.apache.spark.graphx.VertexId, Double) = (1197,60.5)**

Apparently, the most defensive team can win games by scoring only 60 points, whereas the most offensive team can score an average of 90 points.

显然，最擅长防守的球队只需要60分就能赢得比赛，而最擅长进攻的队平均得分可以达到90分。

Next, let's average the points per game for all games played and look at the two teams with the best and worst offense during the 2015 season:

接下来，让我们平均所有赛事的场均得分，看看在2015赛季的表现最好和最差的两支球队的情况:

**// Average Points Per Game of All Teams**

**所有球队的每场比赛的平均得分**

**val allAvgPPG = averageStat(scoreGraph)(score, AllTeams())**

The output is:

它的输出为：

**scala> allAvgPPG.max()(Ordering.by(\_.\_2))**

**res42: (org.apache.spark.graphx.VertexId, Double) = (1322,83.81481481481481)**

**scala> allAvgPPG.min()(Ordering.by(\_.\_2))**

**res43: (org.apache.spark.graphx.VertexId, Double) =(1212,51.111111111111114)**

To no surprise, the best offensive team is the same as the one who scored most in winning games. To win a game, 50 points is not enough of an average for a team.

毫无意外，最擅长进攻的球队就是那支在比赛中得分最多的球队。要赢得一场比赛，场均50分对于一支球队来说是不够的。

#### Defense stats – D matters as in direction

Previously, we obtained some statistics such as field goals or the three-point percentages that a team achieves.

在此之前，我们获得了一些球队的统计数据，诸如其投篮得分或者三分球命中率。

What if instead we want to aggregate the average points or rebounds that each team concedes to their opponents?

相反，如果我们想要将每支球队的平均得分或篮板数汇总给对手呢？

篮球比赛球队的得分是己方进球的记录，如果想知道己方失球的情况，就要看对手的得分，所以这里的意思是把球队和它的失球情况关联起来。

To compute that, we define a new higher-order function **averageConcededStat**. Compared to **averageStat**, this function needs to send **loserStats** to the winning team and **winnerStats** to the losing team.

为了计算这一点，我们定义了一个新的高阶函数averageConcededStat。相比于averageStat，这个函数需要将loserStats发送给获胜的球队，并将winnerStats发送给失败的球队。

To make things more interesting, we are going to make the team name part of the message **Msg**:

为了让事情更有趣，我们让球队的名字成为消息Msg的一部分:

**def averageConcededStat(graph: Graph[String, FullResult])(getStat: GameStats => Double, rxs: Teams): VertexRDD[(String, Double)] ={**

**type Msg = (Int, Double, String)**

**val aggrStats: VertexRDD[Msg] = graph.aggregateMessages[Msg](**

**// sendMsg rxs match {**

**case \_ : Winners => t => t.sendToSrc((1,**

**getStat(t.attr.loserStats), t.srcAttr))**

**case \_ : Losers => t => t.sendToDst((1, getStat(t.attr.winnerStats), t.dstAttr))**

**case \_ => t => {**

**t.sendToSrc((1,**

**getStat(t.attr.loserStats),t.srcAttr))**

**t.sendToDst((1,**

**getStat(t.attr.winnerStats),t.dstAttr))**

**}**

**}**

**,**

**// mergeMsg**

**(x, y) => (x.\_1 + y.\_1, x.\_2+ y.\_2, x.\_3)**

**)**

**aggrStats mapValues (**

**(id: VertexId, x: Msg) => x match {**

**case (count: Int, total: Double, name: String) =>**

**(name, total/count)**

**})**

**}**

With that, we can calculate the average points conceded by the winning and losing teams as follows:

在此基础上，我们可以计算出输赢球队的平均得分如下:

**val winnersAvgConcededPoints =**

**averageConcededStat(scoreGraph)(score, Winners())**

**val losersAvgConcededPoints =**

**averageConcededStat(scoreGraph)(score, Losers())**

Let's check the output:

让我们查看下输出：

**scala> losersAvgConcededPoints.min()(Ordering.by(\_.\_2))**

**res: (VertexId, (String, Double)) = (1101,(Abilene Chr,74.04761904761905))**

**scala> winnersAvgConcededPoints.min()(Ordering.by(\_.\_2))**

**res: (org.apache.spark.graphx.VertexId, (String, Double)) = (1101,(Abilene Chr,74.04761904761905))**

**scala> losersAvgConcededPoints.max()(Ordering.by(\_.\_2))**

**res: (VertexId, (String, Double)) = (1464,(Youngstown St,78.85714285714286))**

**scala> winnersAvgConcededPoints.max()(Ordering.by(\_.\_2))**

**res: (VertexId, (String, Double)) = (1464,(Youngstown St,71.125))**

The previous code tells us that Abilene Christian University is the most defensive team. They concede the least points whether they win a game or not. On the other hand, Youngstown has the worst defense.

我们从先前的代码中得知，艾柏林基督大学队是最擅长防守的球队。无论他们是否赢得一场比赛，他们的失球最少。与之相反的是，扬斯敦州立大学队的防守能力最弱。

这节Concede指在比赛中被对手进球，我不玩球不知道在中文中应该如何精确简短表达

**Joining average stats into a graph**

将平均统计数据关联到一张图中

The previous example shows us how flexible the **aggregateMessages** operator is. We can define the type **Msg** of the messages to be aggregated to fit our needs. Moreover, we can select which nodes receive the messages. Finally, we can also define how we want to merge the messages.

As a final example, let's aggregate many statistics about each team and join this information into the nodes of the graph:

前面的例子向我们展示了aggregatemessage操作符的灵活性。我们可以定义要聚合的消息的类型Msg以满足我们的需要。此外，我们可以选择接收消息的节点。最后，我们还可以定义合并消息的方式。

作为最后的例子，让我们汇总每个球队的各种统计信息，并将这些信息关联到图的节点中:

1. To start, we create its own class for the team stats:

我们从创建球队统计数据的类开始：

**// Average Stats of All Teams**

**所有球队的平均统计数据**

**case class TeamStat(**

**wins: Int = 0 // Number of wins**

**,losses: Int = 0 // Number of losses**

**,ppg: Int = 0 // Points per game**

**,pcg: Int = 0 // Points conceded pergame**

**,fgp: Double = 0 // Field goal percentage**

**,tpp: Double = 0 // Three point percentage**

**,ftp: Double = 0 // Free Throw percentage**

**){**

**override def toString = wins + "-" + losses**

**}**

2. We collect the average stats for all teams using **aggregateMessages**. For that, we define the type of the message to be an 8-element tuple that holds the counter for games played, won, lost, and other statistics that will be stored in **TeamStat**, as listed previously:

我们使用aggregatemessage收集所有球队的平均统计数据。为此，我们将消息的类型定义为一个8元素的元组，该元组保存一些计数器，用于存储前述的已参加的，已赢球的，已输球的比赛等其他即将存储在TeamStat中的统计信息：

**type Msg = (Int, Int, Int, Int, Int, Double, Double, Double)**

**val aggrStats: VertexRDD[Msg] = scoreGraph.aggregateMessages(**

**// sendMsg**

**t => {**

**t.sendToSrc(( 1,**

**1, 0,**

**t.attr.winnerStats.score,**

**t.attr.loserStats.score,**

**t.attr.winnerStats.fgPercent,**

**t.attr.winnerStats.tpPercent,**

**t.attr.winnerStats.ftPercent**

**))**

**t.sendToDst(( 1,**

**0, 1,**

**t.attr.loserStats.score,**

**t.attr.winnerStats.score,**

**t.attr.loserStats.fgPercent,**

**t.attr.loserStats.tpPercent,**

**t.attr.loserStats.ftPercent**

**))**

**}**

**,**

**// mergeMsg**

**(x, y) => ( x.\_1 + y.\_1, x.\_2 + y.\_2,**

**x.\_3 + y.\_3, x.\_4 + y.\_4,**

**x.\_5 + y.\_5, x.\_6 + y.\_6,**

**x.\_7 + y.\_7, x.\_8 + y.\_8**

**)**

**)**

3. Given the aggregate message **aggrStats**, we map them into a collection of **TeamStats**:

给定聚合消息集合**aggrStats，**我们将它们映射到集合**TeamStats**去**：**

不确定这个aggrStats是集合还是什么，你确认下

**val teamStats: VertexRDD[TeamStat] = aggrStats mapValues {**

**(id: VertexId, m: Msg) => m match{**

**case ( count: Int,**

**wins: Int,**

**losses: Int,**

**totPts: Int,**

**totConcPts: Int,**

**totFG: Double,**

**totTP: Double,**

**totFT: Double) => TeamStat( wins, losses,**

**totPts/count,**

**totConcPts/count,**

**totFG/count,**

**totTP/count,**

**totFT/count)**

**}**

**}**

1. Let's join **teamStats** into the graph. For that, we first create a class **Team** as a new type for the vertex attribute. **Team** will have the name and the **TeamStat** option:

让我们将teamStats关联到图中。为此，我们首先创建一个类Team作为顶点属性的新类型。Team具有两个参数，类型为string的name和类型为可选TeamStat的stats。

Option表示TeamStat的值是可选的，返回值或者Null，翻译的时候我不好把握，校正的时候注意下。

**case class Team(name: String, stats: Option[TeamStat]) {**

**override def toString = name + ": " + stats**

**}**

1. We use the **joinVertices** operator, which we have seen in the previous chapter:

我们使用前面章节中已经见过的**joinVertices**操作符**：**

**// Joining the average stats to vertex attributes**

**将平均统计数据关联到顶点属性：**

**def addTeamStat(id: VertexId, t: Team, stats: TeamStat) = Team(t.name, Some(stats))**

**val statsGraph: Graph[Team, FullResult] =**

**scoreGraph.mapVertices((\_, name) => Team(name, None)).**

**joinVertices(teamStats)(addTeamStat)**

1. We can see that the join has worked well by printing the first three vertices in the new graph **statsGraph**:

打印新的图**statsGraph**的前三个顶点，我们可以发现联接运行良好：

**scala> statsGraph.vertices.take(3).foreach(println)**

**(1260,Loyola-Chicago: Some(17-13))**

**(1410,TX Pan American: Some(7-21))**

**(1426,UT Arlington: Some(15-15))**

1. To conclude this task, let's find out the top 10 teams in the regular seasons. To do so, we define an **Ordering** option for **Option[TeamStat]** as follows:

为了完成这项任务，让我们来看看在常规赛季中排名前10的球队。为此，我们为**Option[TeamStat]**定义了一个**Ordering**选项，如下所示：

Option的翻译比较犹豫，不确定，保留没有翻译的文字原文字体都有所不同

**import scala.math.Ordering**

**object winsOrdering extends Ordering[Option[TeamStat]] {**

**def compare(x: Option[TeamStat], y: Option[TeamStat]) =**

**(x, y) match {**

**case (None, None) => 0**

**case (Some(a), None) => 1**

**case (None, Some(b)) => -1**

**case (Some(a), Some(b)) => if (a.wins == b.wins)**

**a.losses compare b.losses**

**else a.wins compare b.wins**

**}**

**}**

8. Finally:

最后：

**import scala.reflect.classTag**

**import scala.reflect.ClassTag**

**scala> statsGraph.vertices.sortBy(v =>**

**v.\_2.stats,false)(winsOrdering, classTag[Option[TeamStat]]).**

**| take(10).foreach(println)**

**(1246,Kentucky: Some(34-0))**

**(1437,Villanova: Some(32-2))**

**(1112,Arizona: Some(31-3))**

**(1458,Wisconsin: Some(31-3))**

**(1211,Gonzaga: Some(31-2))**

**(1320,Northern Iowa: Some(30-3))**

**(1323,Notre Dame: Some(29-5))**

**(1181,Duke: Some(29-4))**

**(1438,Virginia: Some(29-3))**

**(1268,Maryland: Some(27-6))**

Note that the **ClassTag** parameter is required in **sortBy** to make use of Scala's reflection. That is why we had the previous imports.

注意：sortBy中需要ClassTag参数才能使用Scala的反射。这就是我们需要先前的导入操作的原因。

**Performance optimization**

性能优化：

In addition to the **sendMsg** and **mergeMsg** methods, **aggregateMessages** can also take an optional argument **TripletFields**, which indicates what data is accessed in **EdgeContext**. The main reason for explicitly specifying such information is to help optimize the performance of the **aggregateMessages** operation.

除了sendMsg和mergeMsg方法之外，aggregatemessage还可以使用一个可选参数TripletFields，它指示在EdgeContext中访问哪些数据。显式指定这些信息的主要原因是为了优化aggregatemessage操作的性能。

In fact, **TripletFields** represents a subset of the fields of **\_EdgeTriplet\_** and it enables GraphX to populate only those fields that are necessary.

事实上，TripletFields表示\_ Edgetriple \_字段的一个子集，它使得GraphX能够仅填充那些必需的字段。

The default value is **TripletFields.All**, which means that the **sendMsg** function may access any of the fields in the **EdgeContext** class. Otherwise, the **TripletFields** argument is used to tell GraphX that only part of **EdgeContext** will be required so that an efficient join strategy can be used. All possible options for the **TripletFields** are listed as follows:

TripletFields参数的默认值是TripletFields.All，这意味着sendMsg函数可以访问EdgeContext类中的任何字段。否则，TripletFields参数用于告诉GraphX，只需要访问EdgeContext类的部分字段，这样可以使得连接策略更为高效。TripletFields的所有可能选项如下:

* **TripletFields.All**: This option exposes all the fields (source, edge, and destination)

**TripletFields.All**：这个选项表示可以访问所有字段（源顶点，边和目的顶点）

* **TripletFields.Dst**: This one exposes the destination and edge fields but not the source field

**TripletFields.Dst**：这个选项表示可以访问除了源顶点字段以外的边和目的顶点的字段

* **TripletFields.EdgeOnly**: This option exposes only the edge field but not the source or destination field

**TripletFields.EdgeOnly**：这个选项表示仅可以访问边属性的字段

* **TripletFields.None**: With this option none of the triplet fields are exposed

**TripletFields.None**：这个选项表示不可以访问triple的任一字段

* **TripletFields.Src**: This one exposes the source and edge fields but not the destination field

**TripletFields.Src**：这个选项表示仅可以访问目的顶点以外的源顶点和边属性的字段

此处的expose没有把握翻译，应该是表示可以访问代表[源顶点/目的顶点/边]的属性的那些字段

Using our previous example, if we are interested in computing the total number of wins and losses for each team, we will not need to access any fields of the **EdgeContext** class. In this case, we should use **TripletFields.None** to indicate so:

使用我们前面的例子，如果我们有兴趣计算每个球队的总胜负数，我们将不需要访问EdgeContext类的任何字段。在这种情况下，我们应该使用TripletFields.None来指示：

**// Number of wins of the teams**

球队的赢球比赛数量：

**val numWins: VertexRDD[Int] = scoreGraph.aggregateMessages(**

**triplet => {**

**triplet.sendToSrc(1) // No attribute is passed but an**

**integer仅仅传递一个整数**

**},**

**(x, y) => x + y,**

**TripletFields.None**

**)**

**// Number of losses of the teams**

球队的输球比赛数量：

**val numLosses: VertexRDD[Int] = scoreGraph.aggregateMessages(**

**triplet => {**

**triplet.sendToDst(1) // No attribute is passed but an**

**integer仅仅传递一个整数**

**},**

**(x, y) => x + y,**

**TripletFields.None**

**)**

To see that this works, let's print the top five and bottom five teams:

为了验证它是否正常工作，我们打印前五名和最后五名的球队：

**scala> numWins.sortBy(\_.\_2,false).take(5).foreach(println)**

**(1246,34)**

**(1437,32)**

**(1112,31)**

**(1458,31)**

**(1211,31)**

**scala> numLosses.sortBy(\_.\_2, false).take(5).foreach(println) (1363,28)**

**(1146,27)**

**(1212,27)**

**(1197,27)**

**(1263,27)**

Should you want the name of the top five teams, you need to access the **srcAttr** attribute. In this case, we need to set **tripletFields** to **TripletFields.Src**.

如果你想知道前五名球队的名字，那么你需要访问srcAttr属性。在这种情况下，我们需要将tripletFields设置为TripletFields.Src。

Kentucky as the undefeated team in the regular season:

肯塔基队在常规赛中是一支常胜球队:

**val numWinsOfTeams: VertexRDD[(String, Int)] = scoreGraph.**

**aggregateMessages(**

**t => {**

**t.sendToSrc(t.srcAttr, 1) // Pass source attribute**

**only**

**},**

**(x, y) => (x.\_1, x.\_2 + y.\_2),**

**TripletFields.Src**

**)**

Et voila!:

就是这样！：

法语，在线翻译相互矛盾，应该是个语气词

**scala> numWinsOfTeams.sortBy(\_.\_2.\_2, false).take(5).foreach(println) (1246,(Kentucky,34))**

**(1437,(Villanova,32))**

**(1112,(Arizona,31))**

**(1458,(Wisconsin,31))**

**(1211,(Gonzaga,31))**

**scala> numWinsOfTeams.sortBy(\_.\_2.\_2).take(5).foreach(println)**

**(1146,(Cent Arkansas,2))**

**(1197,(Florida A&M,2))**

**(1398,(Tennessee St,3))**

**(1263,(Maine,3))**

**(1420,(UMBC,4))**

Kentucky has not lost any of its 34 games during the regular season. Too bad that they could not make it into the championship final.

在常规赛季中，肯塔基队的34场比赛都没有输过。可惜的是他们没能进入决赛。

因为他们季后赛输了，这种联赛分为常规赛和季后赛，常规赛决定季后赛排名，再通过季后赛的淘汰赛最后产生决赛队伍。

**The MapReduceTriplets operator**

Prior to Spark 1.2, there was no **aggregateMessages** method in Graph. Instead, the now deprecated **mapReduceTriplets** was the primary aggregation operator. The API for **mapReduceTriplets** is:

在Spark 1.2之前，Graph中并没有aggregateMessages方法。相反，现在不建议使用的mapReduceTriplets是主要的聚合运算符。mapReduceTriplets的API是：

**class Graph[VD, ED] {**

**def mapReduceTriplets[Msg](**

**map: EdgeTriplet[VD, ED] => Iterator[(VertexId, Msg)],**

**reduce: (Msg, Msg) => Msg)**

**: VertexRDD[Msg]**

**}**

Compared to **mapReduceTriplets**, the new operator **aggregateMessages** is more expressive as it employs the message passing mechanism instead of returning an iterator of messages as **mapReduceTriplets** does. In addition, **aggregateMessages** explicitly requires the user to specify the **TripletFields** object for performance improvement as we explained previously. In addition to API improvements, **aggregateMessages** is optimized for performance.

与mapReduceTriplets相比，新的运算符aggregateMessages更具表现力，因为它使用消息传递机制，而不是像mapReduceTriplets那样返回一个消息迭代器。  
另外，正如我们前面所解释的，aggregateMessages明确要求用户指定TripletFields对象以提高性能。  
除了API改进之外，aggregateMessages还针对性能进行了优化。

Since **mapReduceTriplets** is now deprecated, we will not discuss it further. If you have to use it with earlier versions of Spark, you can refer to the Spark programming guide.

由于mapReduceTriplets现已被弃用，我们不会再讨论它。 如果你必须在早期版本的Spark中使用它，可以参考Spark编程指南。

Deprecated翻译成弃用还是不建议使用不确定，校正时根据mapReduceTriplets情况选择，草稿的翻译是为了语句流畅。

**Summary总结**

**AggregateMessages** provides a functional abstraction for aggregating neighborhood information in Spark graphs. This operator applies a user-defined **sendMsg** function to each edge in the graph using **EdgeContext**.

AggregateMessages提供了一个函数抽象，用于聚合Spark图中的邻居信息。该运算符使用EdgeContext将用户定义的sendMsg函数应用于图中的每条边。

Each **EdgeContext** class accesses the required information about the edge and passes that information to its source node and/or destination node using the **sendToSrc** and/or **sendToDst** methods respectively. After all messages have been received by the nodes, the **mergeMsg** function is used to aggregate those messages at each node.

每个EdgeContext类访问有关边的所需信息，并分别使用sendToSrc和/或sendToDst方法将该信息传递给其源节点和/或目标节点。在节点收到所有消息后，使用mergeMsg函数在每个节点上聚合这些消息。

In the next chapter, we will introduce another operator called **Pregel**, which will be useful for creating custom iterative graph-processing algorithms.

在下一章中，我们将介绍另一个名为Pregel的运算符，它将用于创建自定义迭代图处理算法。