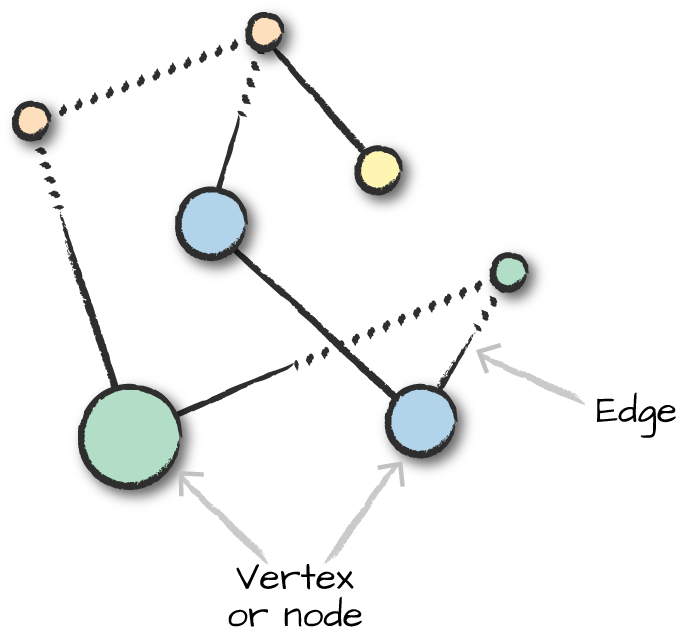
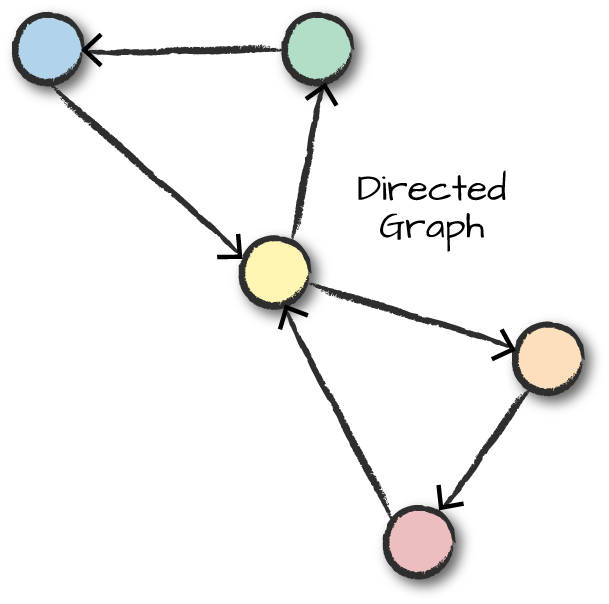
**Chapter 30. Graph Analytics**

The previous chapter covered some conventional unsupervised techniques. This chapter is going to dive into a more specialized toolset: graph processing. Graphs are data structures composed of *nodes*, or *vertices*, which are arbitrary objects, and *edges* that define relationships between these nodes. *Graph analytics* is the process of analyzing these relationships. An example graph might be your friend group. In the context of graph analytics, each vertex or node would represent a person, and each edge would represent a relationship. Figure 30-1 shows a sample graph.



*Figure 30-1. A sample graph with seven nodes and seven edges*

This particular graph is *undirected*, in that the edges do not have a specified “start” and “end” vertex. There are also *directed* graphs that specify a start and end. Figure 30-2 shows a *directed* graph where the edges are directional.



*Figure 30-2. A directed graph*

Edges and vertices in graphs can also have data associated with them. In our friend example, the weight of the edge might represent the intimacy between different friends; acquaintances would have low-weight edges between them, while married individuals would have edges with large weights. We could set this value by looking at communication frequency between nodes and weighting the edges accordingly. Each vertex (person) might also have data such as a name.

Graphs are a natural way of describing relationships and many different problem sets, and Spark provides several ways of working in this analytics paradigm. Some business use cases could be detecting credit card fraud, motif finding, determining importance of papers in bibliographic networks (i.e., which papers are most referenced), and ranking web pages, as Google famously used the PageRank algorithm to do.

Spark has long contained an RDD-based library for performing graph processing: GraphX. This provided a very low-level interface that was extremely powerful, but just like RDDs, wasn’t easy to use or optimize. GraphX remains a core part of Spark. Companies continue to build production applications on top of it, and it still sees some minor feature development. The GraphX API is well documented simply because it hasn’t changed much since its creation. However, some of the developers of Spark (including some of the original authors of GraphX) have recently created a next-generation graph analytics library on Spark: GraphFrames. GraphFrames extends GraphX to provide a DataFrame API and support for Spark’s different language bindings so that users of Python can take advantage of the scalability of the tool. In this book, we will focus on GraphFrames.

[GraphFrames](http://graphframes.github.io/index.html) is currently [available as a Spark package](http://spark-packages.org/package/graphframes/graphframes), an external package that you need to load when you start up your Spark application, but may be merged into the core of Spark in the future. For the most part, there should be little difference in performance between the two (except for a huge user experience improvement in GraphFrames). There is some small overhead when using GraphFrames, but for the most part it tries to call down to GraphX where appropriate; and for most, the user experience gains greatly outweigh this minor overhead.

**HOW DOES GRAPHFRAMES COMPARE TO GRAPH DATABASES?**

Spark is not a database. Spark is a distributed computation engine, but it does not store data long-term or perform transactions. You can build a graph computation on top of Spark, but that’s fundamentally different from a database. GraphFrames can scale to much larger workloads than many graph databases and performs well for analytics but does not support transactional processing and serving.

The goal of this chapter is to show you how to use GraphFrames to perform graph analytics on Spark. We are going to be doing this with publicly available bike data from [the Bay Area Bike Share portal](http://www.bayareabikeshare.com/open-data).

**TIP**

During the course of writing this book, this map and data have changed dramatically (even the naming!). We include a copy of the dataset inside the *[data](https://github.com/databricks/Spark-The-Definitive-Guide/tree/master/data)* [folder](https://github.com/databricks/Spark-The-Definitive-Guide/tree/master/data) of this book’s repository. Be sure to use that dataset to replicate the following results; and when you’re feeling adventurous, expand to the whole dataset!

To get set up, you’re going to need to point to the proper package. To do this from the command line, you’ll run:

./bin/spark-shell --packages graphframes:graphframes:0.5.0-spark2.2-s\_2.11

*// in Scala*

**val** bikeStations **=** spark.read.option("header","true")

.csv("/data/bike-data/201508\_station\_data.csv")

**val** tripData **=** spark.read.option("header","true")

.csv("/data/bike-data/201508\_trip\_data.csv")

*# in Python*

bikeStations = spark.read.option("header","true")\

.csv("/data/bike-data/201508\_station\_data.csv")

tripData = spark.read.option("header","true")\

.csv("/data/bike-data/201508\_trip\_data.csv")

**Building a Graph**

The first step is to build the graph. To do this we need to define the vertices and edges, which are DataFrames with some specifically named columns. In our case, we’re creating a *directed graph*. This graph will point from the source to the location. In the context of this bike trip data, this will point from a trip’s starting location to a trip’s ending location. To define the graph, we use the naming conventions for columns presented in the GraphFrames library. In the vertices table we define our identifier as id (in our case this is of string type), and in the edges table we label each edge’s source vertex ID as src and the destination ID as dst:

*// in Scala*

**val** stationVertices **=** bikeStations.withColumnRenamed("name", "id").distinct()

**val** tripEdges **=** tripData

.withColumnRenamed("Start Station", "src")

.withColumnRenamed("End Station", "dst")

*# in Python*

stationVertices = bikeStations.withColumnRenamed("name", "id").distinct()

tripEdges = tripData\

.withColumnRenamed("Start Station", "src")\

.withColumnRenamed("End Station", "dst")

We can now build a GraphFrame object, which represents our graph, from the vertex and edge DataFrames we have so far. We will also leverage caching because we’ll be accessing this data frequently in later queries:

*// in Scala*

**import** **org.graphframes.GraphFrame**

**val** stationGraph **=** **GraphFrame**(stationVertices, tripEdges)

stationGraph.cache()

*# in Python*

**from** **graphframes** **import** GraphFrame

stationGraph = GraphFrame(stationVertices, tripEdges)

stationGraph.cache()

Now we can see the basic statistics about graph (and query our original DataFrame to ensure that we see the expected results):

*// in Scala*

println(s"Total Number of Stations: ${stationGraph.vertices.count()}")

println(s"Total Number of Trips in Graph: ${stationGraph.edges.count()}")

println(s"Total Number of Trips in Original Data: ${tripData.count()}")

*# in Python*

**print** "Total Number of Stations: " + str(stationGraph.vertices.count())

**print** "Total Number of Trips in Graph: " + str(stationGraph.edges.count())

**print** "Total Number of Trips in Original Data: " + str(tripData.count())

This returns the following results:

Total Number of Stations: 70

Total Number of Trips in Graph: 354152

Total Number of Trips in Original Data: 354152

**Querying the Graph**

The most basic way of interacting with the graph is simply querying it, performing things like counting trips and filtering by given destinations. GraphFrames provides simple access to both vertices and edges as DataFrames. Note that our graph retained all the additional columns in the data in addition to IDs, sources, and destinations, so we can also query those if needed:

*// in Scala*

**import** **org.apache.spark.sql.functions.desc**

stationGraph.edges.groupBy("src", "dst").count().orderBy(desc("count")).show(10)

*# in Python*

**from** **pyspark.sql.functions** **import** desc

stationGraph.edges.groupBy("src", "dst").count().orderBy(desc("count")).show(10)

+--------------------+--------------------+-----+

| src| dst|count|

+--------------------+--------------------+-----+

|San Francisco Cal...| Townsend at 7th| 3748|

|Harry Bridges Pla...|Embarcadero at Sa...| 3145|

...

| Townsend at 7th|San Francisco Cal...| 2192|

|Temporary Transba...|San Francisco Cal...| 2184|

+--------------------+--------------------+-----+

We can also filter by any valid DataFrame expression. In this instance, I want to look at one specific station and the count of trips in and out of that station:

*// in Scala*

stationGraph.edges

.where("src = 'Townsend at 7th' OR dst = 'Townsend at 7th'")

.groupBy("src", "dst").count()

.orderBy(desc("count"))

.show(10)

*# in Python*

stationGraph.edges\

.where("src = 'Townsend at 7th' OR dst = 'Townsend at 7th'")\

.groupBy("src", "dst").count()\

.orderBy(desc("count"))\

.show(10)

+--------------------+--------------------+-----+

| src| dst|count|

+--------------------+--------------------+-----+

|San Francisco Cal...| Townsend at 7th| 3748|

| Townsend at 7th|San Francisco Cal...| 2734|

...

| Steuart at Market| Townsend at 7th| 746|

| Townsend at 7th|Temporary Transba...| 740|

+--------------------+--------------------+-----+

**Subgraphs**

*Subgraphs* are just smaller graphs within the larger one. We saw in the last section how we can query a given set of edges and vertices. We can use this query ability to create subgraphs:

*// in Scala*

**val** townAnd7thEdges **=** stationGraph.edges

.where("src = 'Townsend at 7th' OR dst = 'Townsend at 7th'")

**val** subgraph **=** **GraphFrame**(stationGraph.vertices, townAnd7thEdges)

*# in Python*

townAnd7thEdges = stationGraph.edges\

.where("src = 'Townsend at 7th' OR dst = 'Townsend at 7th'")

subgraph = GraphFrame(stationGraph.vertices, townAnd7thEdges)

We can then apply the following algorithms to either the original graph or the subgraph.

**Motif Finding**

*Motifs* are a way of expresssing structural patterns in a graph. When we specify a motif, we are querying for patterns in the data instead of actual data. In GraphFrames, we specify our query in a domain-specific language similar to Neo4J’s Cypher language. This language lets us specify combinations of vertices and edges and assign then names. For example, if we want to specify that a given vertex a connects to another vertex b through an edge ab, we would specify (a)-[ab]->(b). The names inside parentheses or brackets do not signify values but instead what the columns for matching vertices and edges should be named in the resulting DataFrame. We can omit the names (e.g., (a)-[]->()) if we do not intend to query the resulting values.

Let’s perform a query on our bike data. In plain English, let’s find all the rides that form a “triangle” pattern between three stations. We express this with the following motif, using the find method to query our GraphFrame for that pattern. (a) signifies the starting station, and [ab] represents an edge from (a) to our next station (b). We repeat this for stations (b) to (c) and then from (c) to (a):

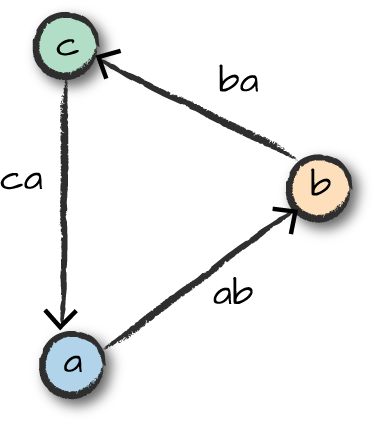
*// in Scala*

**val** motifs **=** stationGraph.find("(a)-[ab]->(b); (b)-[bc]->(c); (c)-[ca]->(a)")

*# in Python*

motifs = stationGraph.find("(a)-[ab]->(b); (b)-[bc]->(c); (c)-[ca]->(a)")

Figure 30-3 presents a visual representation of this query.



*Figure 30-3. Triangle motif in our triangle query*

The DataFrame we get from running this query contains nested fields for vertices a, b, and c, as well as the respective edges. We can now query this as we would a DataFrame. For example, given a certain bike, what is the shortest trip the bike has taken from station a, to station b, to station c, and back to station a? The following logic will parse our timestamps, into Spark timestamps and then we’ll do comparisons to make sure that it’s the same bike, traveling from station to station, and that the start times for each trip are correct:

*// in Scala*

**import** **org.apache.spark.sql.functions.expr**

motifs.selectExpr("\*",

"to\_timestamp(ab.`Start Date`, 'MM/dd/yyyy HH:mm') as abStart",

"to\_timestamp(bc.`Start Date`, 'MM/dd/yyyy HH:mm') as bcStart",

"to\_timestamp(ca.`Start Date`, 'MM/dd/yyyy HH:mm') as caStart")

.where("ca.`Bike #` = bc.`Bike #`").where("ab.`Bike #` = bc.`Bike #`")

.where("a.id != b.id").where("b.id != c.id")

.where("abStart < bcStart").where("bcStart < caStart")

.orderBy(expr("cast(caStart as long) - cast(abStart as long)"))

.selectExpr("a.id", "b.id", "c.id", "ab.`Start Date`", "ca.`End Date`")

.limit(1).show(**false**)

*# in Python*

**from** **pyspark.sql.functions** **import** expr

motifs.selectExpr("\*",

"to\_timestamp(ab.`Start Date`, 'MM/dd/yyyy HH:mm') as abStart",

"to\_timestamp(bc.`Start Date`, 'MM/dd/yyyy HH:mm') as bcStart",

"to\_timestamp(ca.`Start Date`, 'MM/dd/yyyy HH:mm') as caStart")\

.where("ca.`Bike #` = bc.`Bike #`").where("ab.`Bike #` = bc.`Bike #`")\

.where("a.id != b.id").where("b.id != c.id")\

.where("abStart < bcStart").where("bcStart < caStart")\

.orderBy(expr("cast(caStart as long) - cast(abStart as long)"))\

.selectExpr("a.id", "b.id", "c.id", "ab.`Start Date`", "ca.`End Date`")

.limit(1).show(1, False)

We see the fastest trip is approximately 20 minutes. Pretty fast for three different people (we assume) using the same bike!

Note also that we had to filter the triangles returned by our motif query in this example. In general, different vertex IDs used in the query will not be forced to match distinct vertices, so you should perform this type of filtering if you want distinct vertices. One of the most powerful features of GraphFrames is that you can combine motif finding with DataFarme queries over the resulting tables to further narrow down, sort, or aggregate the patterns found.

**Graph Algorithms**

A graph is just a logical representation of data. Graph theory provides numerous algorithms for analyzing data in this format, and GraphFrames allows us to leverage many algorithms out of the box. Development continues as new algorithms are added to GraphFrames, so this list will most likely continue to grow.

**PageRank**

One of the most prolific graph algorithms is [PageRank](https://en.wikipedia.org/wiki/PageRank). Larry Page, cofounder of Google, created PageRank as a research project for how to rank web pages. Unfortunately, a complete explanation of how PageRank works is outside the scope of this book. However, to quote Wikipedia, the high-level explanation is as follows:

*PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.*

PageRank generalizes quite well outside of the web domain. We can apply this right to our own data and get a sense for important bike stations (specifically, those that receive a lot of bike traffic). In this example, important bike stations will be assigned large PageRank values:

*// in Scala*

**import** **org.apache.spark.sql.functions.desc**

**val** ranks **=** stationGraph.pageRank.resetProbability(0.15).maxIter(10).run()

ranks.vertices.orderBy(desc("pagerank")).select("id", "pagerank").show(10)

*# in Python*

**from** **pyspark.sql.functions** **import** desc

ranks = stationGraph.pageRank(resetProbability=0.15, maxIter=10)

ranks.vertices.orderBy(desc("pagerank")).select("id", "pagerank").show(10)

+--------------------+------------------+

| id| pagerank|

+--------------------+------------------+

|San Jose Diridon ...| 4.051504835989922|

|San Francisco Cal...|3.3511832964279518|

...

| Townsend at 7th| 1.568456580534273|

|Embarcadero at Sa...|1.5414242087749768|

+--------------------+------------------+

**GRAPH ALGORITHM APIS: PARAMETERS AND RETURN VALUES**

Most algorithms in GraphFrames are accessed as methods which take parameters (e.g., resetProbability in this PageRank example). Most algorithms return either a new GraphFrame or a single DataFrame. The results of the algorithm are stored as one or more columns in the GraphFrame’s vertices and/or edges or the DataFrame. For PageRank, the algorithm returns a GraphFrame, and we can extract the estimated PageRank values for each vertex from the new pagerank column.

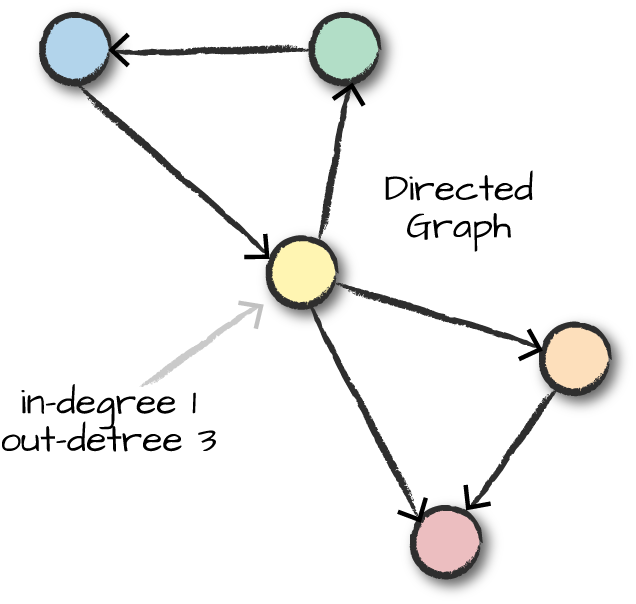
**WARNING**

Depending on the resources available on your machine, this may take some time. You can always try a smaller set of data before running this to see the results. On Databricks Community Edition, this takes about 20 seconds to run, although some reviewers found it to take much longer on their machines.

Interestingly, we see that Caltrain stations rank quite highly. This makes sense because these are natural connection points where a lot of bike trips might end up. Either as commuters move from home to the Caltrain station for their commute or from the Caltrain station to home.

**In-Degree and Out-Degree Metrics**

Our graph is a directed graph. This is due to the bike trips being directional, starting in one location and ending in another. One common task is to count the number of trips into or out of a given station. To measure trips in and out of stations, we will use a metric called in-degree and out-degree, respectively, as seen in Figure 30-4.



*Figure 30-4. In-degree and out-degree*

This is particularly applicable in the context of social networking because certain users may have many more inbound connections (i.e., followers) than outbound connections (i.e., people they follow). Using the following query, you can find interesting people in the social network who might have more influence than others. GraphFrames provides a simple way to query our graph for this information:

*// in Scala*

**val** inDeg **=** stationGraph.inDegrees

inDeg.orderBy(desc("inDegree")).show(5, **false**)

*# in Python*

inDeg = stationGraph.inDegrees

inDeg.orderBy(desc("inDegree")).show(5, False)

The result of querying for the stations sorted by the highest in-degree:

+----------------------------------------+--------+

|id |inDegree|

+----------------------------------------+--------+

|San Francisco Caltrain (Townsend at 4th)|34810 |

|San Francisco Caltrain 2 (330 Townsend) |22523 |

|Harry Bridges Plaza (Ferry Building) |17810 |

|2nd at Townsend |15463 |

|Townsend at 7th |15422 |

+----------------------------------------+--------+

We can query the out degrees in the same fashion:

*// in Scala*

**val** outDeg **=** stationGraph.outDegrees

outDeg.orderBy(desc("outDegree")).show(5, **false**)

*# in Python*

outDeg = stationGraph.outDegrees

outDeg.orderBy(desc("outDegree")).show(5, False)

+---------------------------------------------+---------+

|id |outDegree|

+---------------------------------------------+---------+

|San Francisco Caltrain (Townsend at 4th) |26304 |

|San Francisco Caltrain 2 (330 Townsend) |21758 |

|Harry Bridges Plaza (Ferry Building) |17255 |

|Temporary Transbay Terminal (Howard at Beale)|14436 |

|Embarcadero at Sansome |14158 |

+---------------------------------------------+---------+

The ratio of these two values is an interesting metric to look at. A higher ratio value will tell us where a large number of trips end (but rarely begin), while a lower value tells us where trips often begin (but infrequently end):

*// in Scala*

**val** degreeRatio **=** inDeg.join(outDeg, **Seq**("id"))

.selectExpr("id", "double(inDegree)/double(outDegree) as degreeRatio")

degreeRatio.orderBy(desc("degreeRatio")).show(10, **false**)

degreeRatio.orderBy("degreeRatio").show(10, **false**)

*# in Python*

degreeRatio = inDeg.join(outDeg, "id")\

.selectExpr("id", "double(inDegree)/double(outDegree) as degreeRatio")

degreeRatio.orderBy(desc("degreeRatio")).show(10, False)

degreeRatio.orderBy("degreeRatio").show(10, False)

Those queries result in the following data:

+----------------------------------------+------------------+

|id |degreeRatio |

+----------------------------------------+------------------+

|Redwood City Medical Center |1.5333333333333334|

|San Mateo County Center |1.4724409448818898|

...

|Embarcadero at Vallejo |1.2201707365495336|

|Market at Sansome |1.2173913043478262|

+----------------------------------------+------------------+

+-------------------------------+------------------+

|id |degreeRatio |

+-------------------------------+------------------+

|Grant Avenue at Columbus Avenue|0.5180520570948782|

|2nd at Folsom |0.5909488686085761|

...

|San Francisco City Hall |0.7928849902534113|

|Palo Alto Caltrain Station |0.8064516129032258|

+-------------------------------+------------------+

**Breadth-First Search**

*Breadth-first search* will search our graph for how to connect two sets of nodes, based on the edges in the graph. In our context, we might want to do this to find the shortest paths to different stations, but the algorithm also works for *sets* of nodes specified through a SQL expression. We can specify the maximum of edges to follow with the maxPathLength, and we can also specify an edgeFilter to filter out edges that do not meet a requirement, like trips during nonbusiness hours.

We’ll choose two fairly close stations so that this does not run too long. However, you can do interesting graph traversals when you have sparse graphs that have distant connections. Feel free to play around with the stations (especially those in other cities) to see if you can get distant stations to connect:

*// in Scala*

stationGraph.bfs.fromExpr("id = 'Townsend at 7th'")

.toExpr("id = 'Spear at Folsom'").maxPathLength(2).run().show(10)

*# in Python*

stationGraph.bfs(fromExpr="id = 'Townsend at 7th'",

toExpr="id = 'Spear at Folsom'", maxPathLength=2).show(10)

+--------------------+--------------------+--------------------+

| from| e0| to|

+--------------------+--------------------+--------------------+

|[65,Townsend at 7...|[913371,663,8/31/...|[49,Spear at Fols...|

|[65,Townsend at 7...|[913265,658,8/31/...|[49,Spear at Fols...|

...

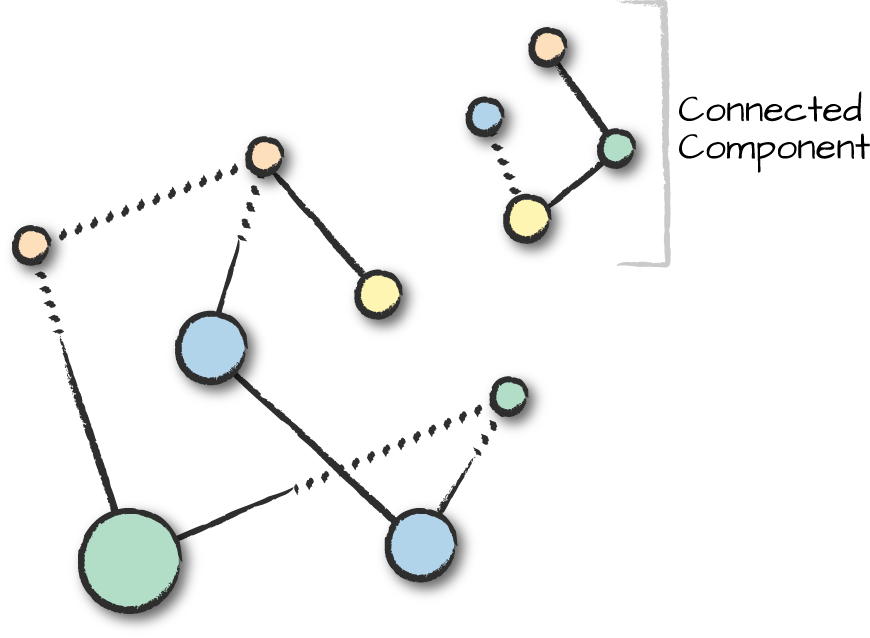
|[65,Townsend at 7...|[903375,850,8/24/...|[49,Spear at Fols...|

|[65,Townsend at 7...|[899944,910,8/21/...|[49,Spear at Fols...|

+--------------------+--------------------+--------------------+

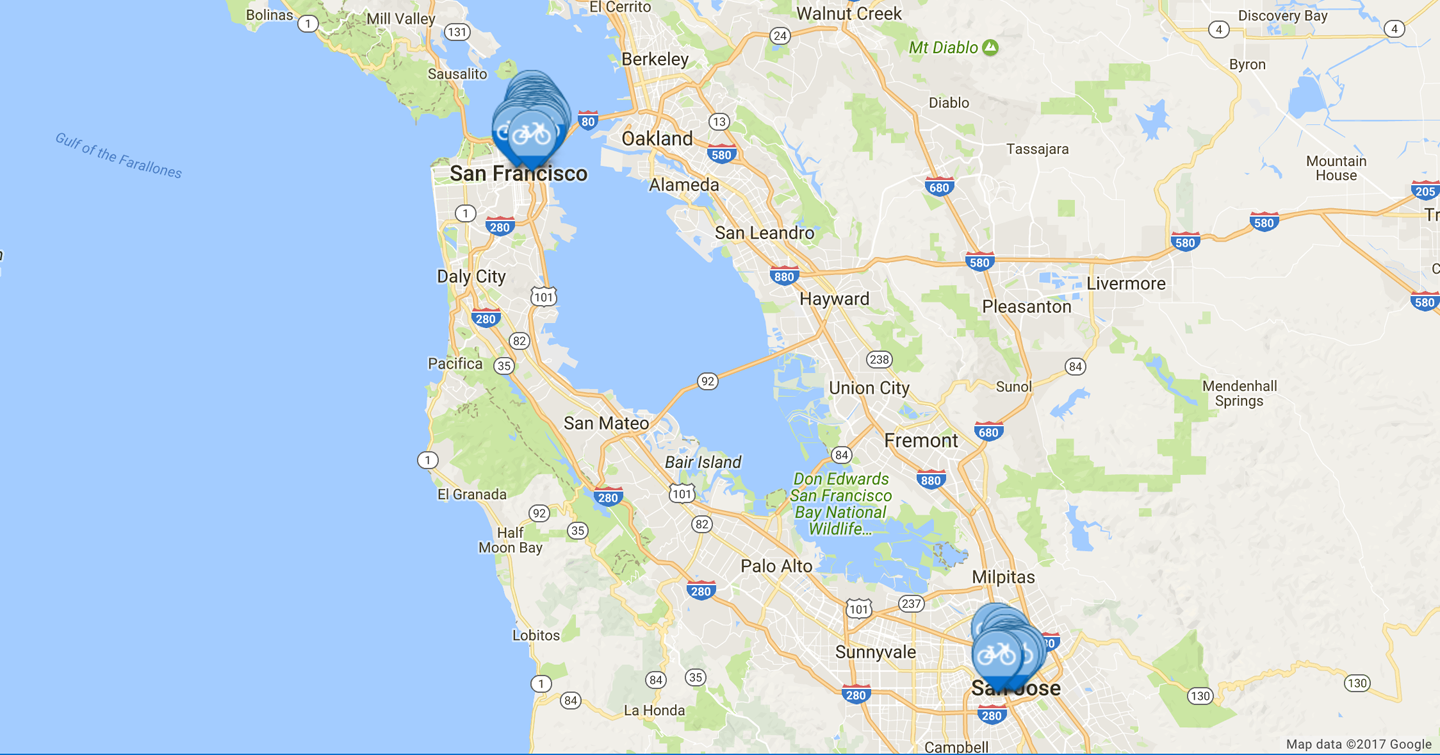
**Connected Components**

A *connected component* defines an (undirected) subgraph that has connections to itself but does not connect to the greater graph, as illustrated in Figure 30-5.



*Figure 30-5. A connected component*

The connected components algorithm does not directly relate to our current problem because they assume an undirected graph. However, we can still run the algorithm, which just assumes that there are is no directionality associated with our edges. In fact, if we look at the bike share map, we assume that we would get two distinct connected components (Figure 30-6).



*Figure 30-6. A map of Bay Area bike share locations*

**WARNING**

To run this algorithm, you will need to set a checkpoint directory which will store the state of the job at every iteration. This allows you to continue where you left off if the job crashes. This is probably one of the most expensive algorithms currently in GraphFrames, so expect delays.

One thing you will likely have to do to run this algorithm on your local machine is take a sample of the data, just as we do in the following code example (taking a sample can help you get to a result without crashing the Spark application with garbage collection issues):

*// in Scala*

spark.sparkContext.setCheckpointDir("/tmp/checkpoints")

*# in Python*

spark.sparkContext.setCheckpointDir("/tmp/checkpoints")

*// in Scala*

**val** minGraph **=** **GraphFrame**(stationVertices, tripEdges.sample(**false**, 0.1))

**val** cc **=** minGraph.connectedComponents.run()

*# in Python*

minGraph = GraphFrame(stationVertices, tripEdges.sample(False, 0.1))

cc = minGraph.connectedComponents()

From this query we get two connected components but not necessarily the ones we might expect. Our sample may not have all of the correct data or information so we’d probably need more compute resources to investigate further:

*// in Scala*

cc.where("component != 0").show()

*# in Python*

cc.where("component != 0").show()

+----------+------------------+---------+-----------+---------+------------+-----

|station\_id| id| lat| long|dockcount| landmark|in...

+----------+------------------+---------+-----------+---------+------------+-----

| 47| Post at Kearney|37.788975|-122.403452| 19|San Franc...| ...

| 46|Washington at K...|37.795425|-122.404767| 15|San Franc...| ...

+----------+------------------+---------+-----------+---------+------------+-----

**Strongly Connected Components**

GraphFrames includes another related algorithm that relates to directed graphs: *strongly connected components*, which takes directionality into account. A strongly connected component is a subgraph that has paths between all pairs of vertices inside it.

*// in Scala*

**val** scc **=** minGraph.stronglyConnectedComponents.maxIter(3).run()

*# in Python*

scc = minGraph.stronglyConnectedComponents(maxIter=3)

scc.groupBy("component").count().show()

**Advanced Tasks**

This is just a short selection of some of the features of GraphFrames. The GraphFrames library also includes features such as writing your own algorithms via a message-passing interface, triangle counting, and converting to and from GraphX. You can find more information in the GraphFrames documentation.

**Conclusion**

In this chapter, we took a tour of GraphFrames, a library for performing graph analysis on Apache Spark. We took a more tutorial-based approach, since this processing technique is not necessarily the first tool that people use when performing advanced analytics. It is nonetheless a powerful tool for analyzing relationships between different objects, and critical in many domains. The next chapter will talk about more cutting-edge functionality—specifically, deep learning.