

Experiment report: Pregel

Done by student: **Bylkova Kristina (伯汀娜), 1820249049**

Course: **Big Data Analysis Technology**

Date: **October 2024**

Questions

1. What happened in Graph level, partition level and node level when using Pregel do Graph Computation?

Answer: Doing graph computation leads to the entire graph division. The graph is divided into smaller, manageable pieces, which are called partitions. Each vertex and its edges are assigned to specific workers based on the partitioning strategy (graph level). Each worker computes the state of its local vertices independently during each superstep (partition level). During a superstep, each vertex processes incoming messages from other vertices. Based on these messages, the vertex may update its state or send messages to other vertices (node level).

2. How can we divide the graph into partition?

Answer: Graph is partitioned by hash (vertex ID) by default, but user partitioning function can be designed individually. At first, it needed to select a static or dynamic partitioning method. Then write code to distribute vertices and edges across partitions based on the chosen method. Next step will be assigning each partition to a worker node (determining which vertices belong to which worker based on the partitioning strategy). As a result, during each superstep, workers process their local vertices and exchange messages with vertices in partitions as needed.

3. Explain the Fault Tolerance mechanism in Pregel.

Answer: During execution, Pregel periodically saves the state of the graph, including the state of each vertex and the messages that have been sent. This snapshot acts as a recovery point. If a worker node fails, the system can restart the computation from the last successful checkpoint. It means that the master reassigns graph partitions to the currently available workers and all workers reload their partition state from most recent available checkpoint. It is important to say that confined recovery is used here: recovery is only confined to the lost partitions. So the system recomputes using logged messages from healthy partitions and recalculated ones from recovering partitions.

MindMap

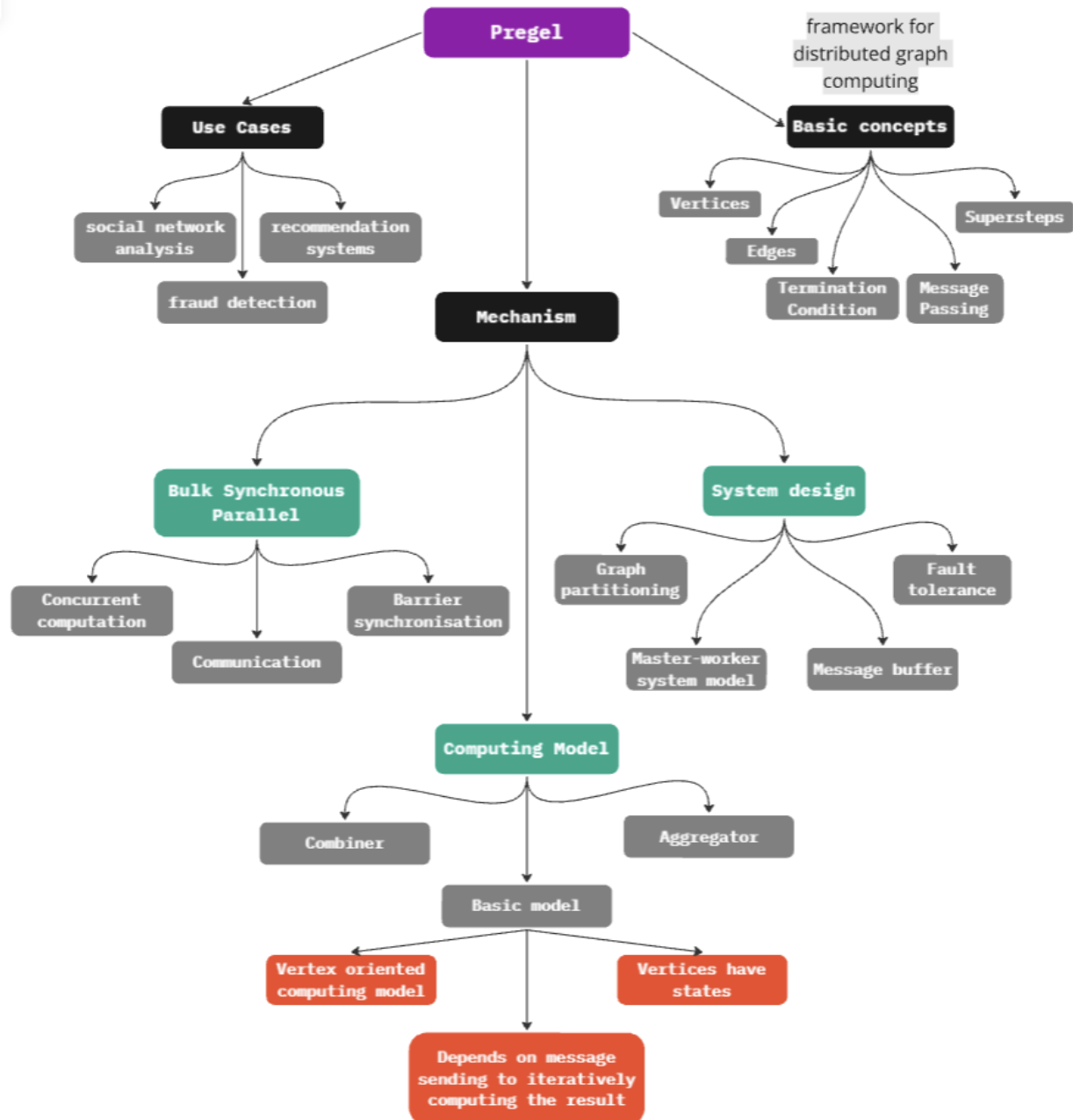
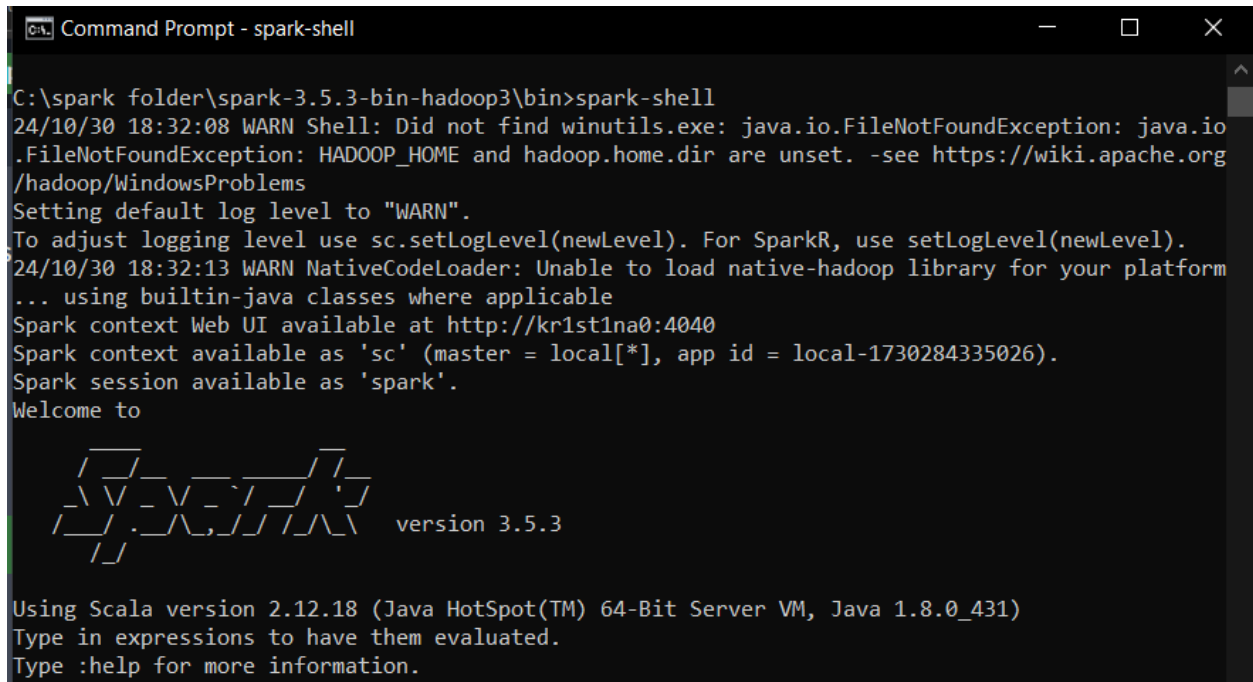


Figure 1: Pregel MindMap

Launching Spark



```
Command Prompt - spark-shell

C:\spark folder\spark-3.5.3-bin-hadoop3\bin>spark-shell
24/10/30 18:32:08 WARN Shell: Did not find winutils.exe: java.io.FileNotFoundException: java.io
.FileNotFoundException: HADOOP_HOME and hadoop.home.dir are unset. -see https://wiki.apache.org
/hadoop/WindowsProblems
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
24/10/30 18:32:13 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform
... using builtin-java classes where applicable
Spark context Web UI available at http://kr1st1na0:4040
Spark context available as 'sc' (master = local[*], app id = local-1730284335026).
Spark session available as 'spark'.
Welcome to

  ____      _
 / ___|    / \
| |  | |  / _ \
| |  | | / ___ \
| |  | || |___) |
| |  | || |___) |
| |  | || |___) |
|_|  |_| \____/

version 3.5.3

Using Scala version 2.12.18 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_431)
Type in expressions to have them evaluated.
Type :help for more information.
```

Figure 2: Spark-Shell

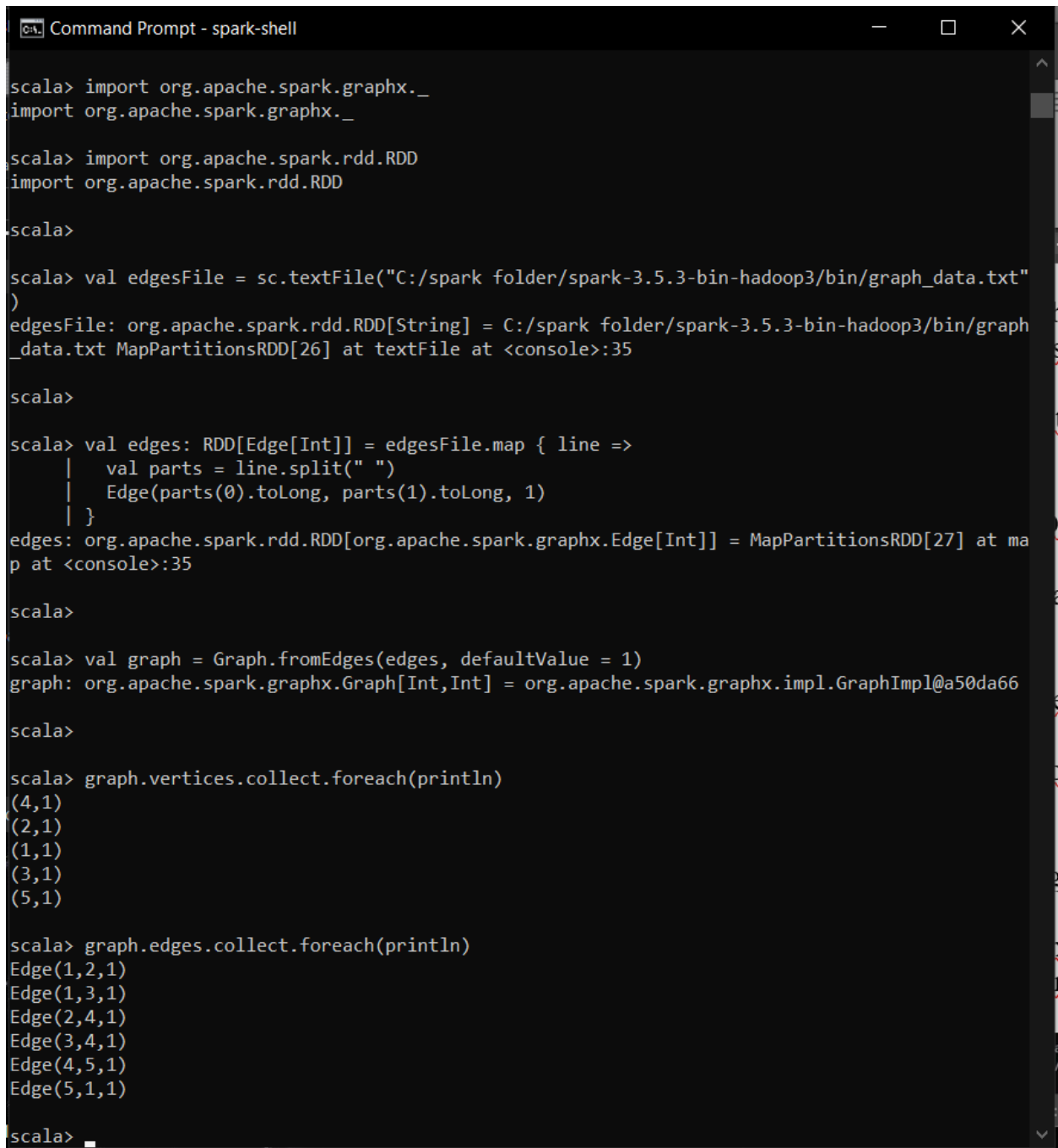
Experiment 1: Working with Graph Data Files

Task: Learn how to read edge list data from an external file and convert it into a graph data structure.

Code:

```
1 import org.apache.spark.graphx._
2 import org.apache.spark.rdd.RDD
3
4 val edgesFile = sc.textFile("C:/spark folder/spark-3.5.3-bin-hadoop3/bin/graph_data.txt")
5
6 val edges: RDD[Edge[Int]] = edgesFile.map { line =>
7   val parts = line.split(" ")
8   Edge(parts(0).toLong, parts(1).toLong, 1)
9 }
10
11 val graph = Graph.fromEdges(edges, defaultValue = 1)
12
13 graph.vertices.collect.foreach(println)
14 graph.edges.collect.foreach(println)
```

Result: I read the graph from the text file. The edges of the graph are stored in graph variable. Then I printed vertices and edges of the graph. Thus, I learned how to read edge list data from a file and convert it into a graph data structure.



```
scala> import org.apache.spark.graphx._
import org.apache.spark.graphx._

scala> import org.apache.spark.rdd.RDD
import org.apache.spark.rdd.RDD

scala>

scala> val edgesFile = sc.textFile("C:/spark folder/spark-3.5.3-bin-hadoop3/bin/graph_data.txt")
edgesFile: org.apache.spark.rdd.RDD[String] = C:/spark folder/spark-3.5.3-bin-hadoop3/bin/graph_data.txt MapPartitionsRDD[26] at textFile at <console>:35

scala>

scala> val edges: RDD[Edge[Int]] = edgesFile.map { line =>
  |   val parts = line.split(" ")
  |   Edge(parts(0).toLong, parts(1).toLong, 1)
  | }
edges: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Int]] = MapPartitionsRDD[27] at map at <console>:35

scala>

scala> val graph = Graph.fromEdges(edges, defaultValue = 1)
graph: org.apache.spark.graphx.Graph[Int,Int] = org.apache.spark.graphx.impl.GraphImpl@a50da66

scala>

scala> graph.vertices.collect.foreach(println)
(4,1)
(2,1)
(1,1)
(3,1)
(5,1)

scala> graph.edges.collect.foreach(println)
Edge(1,2,1)
Edge(1,3,1)
Edge(2,4,1)
Edge(3,4,1)
Edge(4,5,1)
Edge(5,1,1)

scala>
```

Figure 3: Execution 1

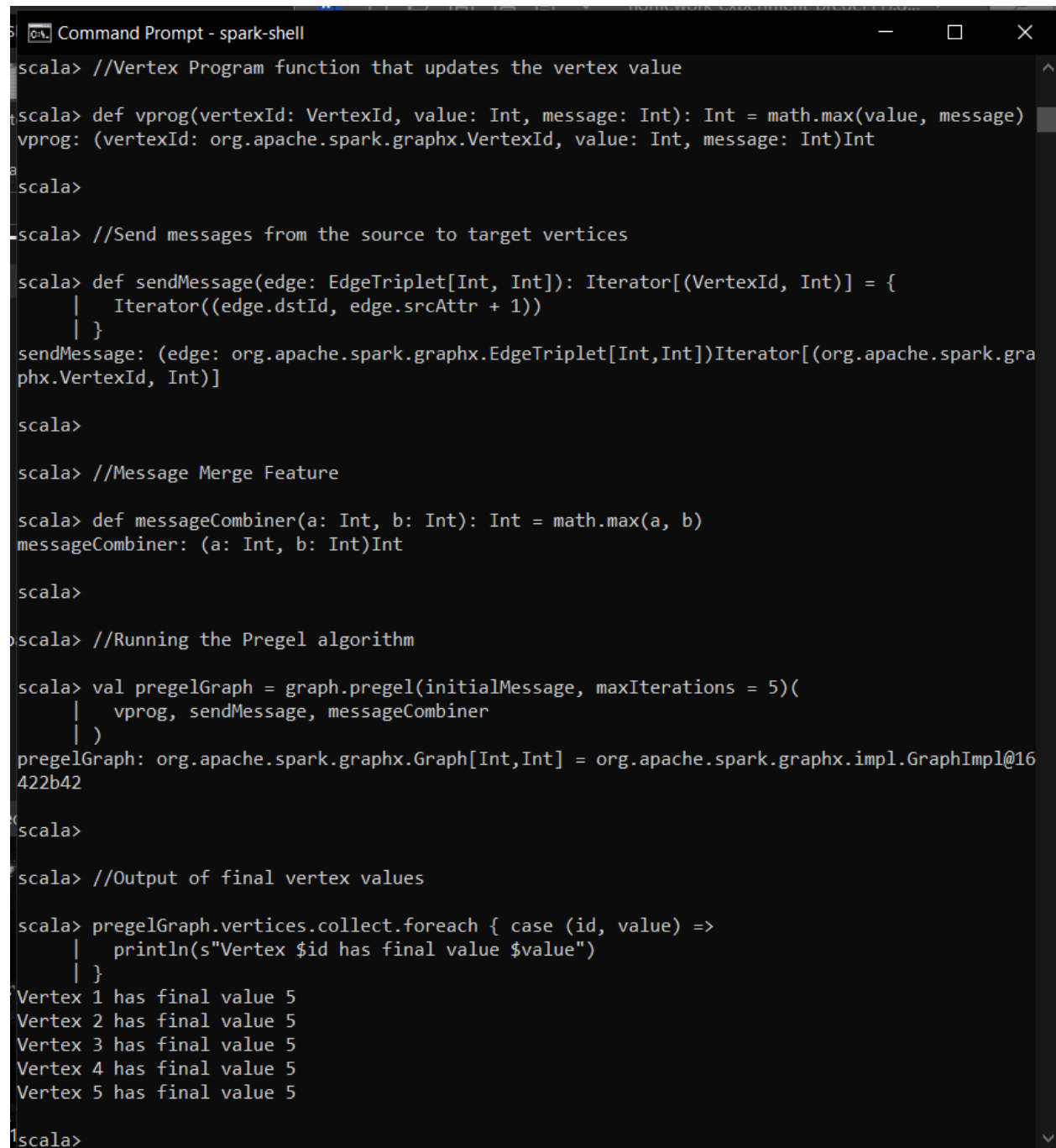
Experiment 2: Propagation of Pregel Values

Task: Create a graph with vertices and edges, and then apply the Pregel algorithm to propagate values between the vertices.

Code:

```
1 import org.apache.spark.graphx._
2 import org.apache.spark.rdd.RDD
3
4 //Vertex Definition (Vertex ID and Seed Values)
5 val vertices: RDD[(VertexId, Int)] = sc.parallelize(Seq(
6   (1L, 0), (2L, 0), (3L, 0), (4L, 0), (5L, 0)
7 ))
8
9 //Rib Definition (Source, Purpose, Weight)
10 val edges: RDD[Edge[Int]] = sc.parallelize(Seq(
11   Edge(1L, 2L, 1), Edge(2L, 3L, 1), Edge(3L, 4L, 1),
12   Edge(4L, 5L, 1), Edge(5L, 1L, 1)
13 ))
14
15 //Graph Creation
16 val graph = Graph(vertices, edges)
17
18 //A seed value for each vertex (for example, a message)
19 val initialMessage = 0
20
21 //Vertex Program function that updates the vertex value
22 def vprog(vertexId: VertexId, value: Int, message: Int): Int = math.max(value, message)
23
24 //Send messages from the source to target vertices
25 def sendMessage(edge: EdgeTriplet[Int, Int]): Iterator[(VertexId, Int)] = {
26   Iterator((edge.dstId, edge.srcAttr + 1))
27 }
28
29 //Message Merge Feature
30 def messageCombiner(a: Int, b: Int): Int = math.max(a, b)
31
32 //Running the Pregel algorithm
33 val pregelGraph = graph.pregel(initialMessage, maxIterations = 5)(
34   vprog, sendMessage, messageCombiner
35 )
36
37 //Output of final vertex values
38 pregelGraph.vertices.collect.foreach { case (id, value) =>
39   println(s"Vertex $id has final value $value")
40 }
```

Result: I created a graph with vertices and edges and then applied the Pregel algorithm to propagate values between the vertices. As a result I can see final values of each vertex of the graph.



```
scala> //Vertex Program function that updates the vertex value
scala> def vprog(vertexId: VertexId, value: Int, message: Int): Int = math.max(value, message)
vprog: (vertexId: org.apache.spark.graphx.VertexId, value: Int, message: Int)Int
scala>
scala> //Send messages from the source to target vertices
scala> def sendMessage(edge: EdgeTriplet[Int, Int]): Iterator[(VertexId, Int)] = {
    |   Iterator((edge.dstId, edge.srcAttr + 1))
    | }
sendMessage: (edge: org.apache.spark.graphx.EdgeTriplet[Int,Int])Iterator[(org.apache.spark.graphx.VertexId, Int)]
scala>
scala> //Message Merge Feature
scala> def messageCombiner(a: Int, b: Int): Int = math.max(a, b)
messageCombiner: (a: Int, b: Int)Int
scala>
scala> //Running the Pregel algorithm
scala> val pregelGraph = graph.pregel(initialMessage, maxIterations = 5)(
    |   vprog, sendMessage, messageCombiner
    | )
pregelGraph: org.apache.spark.graphx.Graph[Int,Int] = org.apache.spark.graphx.impl.GraphImpl@16422b42
scala>
scala> //Output of final vertex values
scala> pregelGraph.vertices.collect.foreach { case (id, value) =>
    |   println(s"Vertex $id has final value $value")
    | }
Vertex 1 has final value 5
Vertex 2 has final value 5
Vertex 3 has final value 5
Vertex 4 has final value 5
Vertex 5 has final value 5
scala>
```

Figure 4: Execution 2

Experiment 3: Finding the minimum distance to a given vertex

Task: Create a graph with vertices and edges, and then apply the Pregel algorithm to determine the shortest paths from a given vertex to all others.

Code:

```
1 import org.apache.spark.graphx._
2 import org.apache.spark.rdd.RDD
3
4 //Vertex Definition (Vertex ID and Seed Values)
5 val vertices: RDD[(VertexId, Double)] = sc.parallelize(Seq(
6   (1L, Double.PositiveInfinity), (2L, Double.PositiveInfinity),
7   (3L, Double.PositiveInfinity), (4L, Double.PositiveInfinity),
8   (5L, Double.PositiveInfinity)
9 ))
10
11 //Rib Definition (Source, Purpose, Weight)
12 val edges: RDD[Edge[Double]] = sc.parallelize(Seq(
13   Edge(1L, 2L, 1.0), Edge(2L, 3L, 1.0), Edge(3L, 4L, 1.0),
14   Edge(4L, 5L, 1.0), Edge(5L, 1L, 1.0)
15 ))
16
17 //Graph Creation
18 val graph = Graph(vertices, edges)
19
20 //The initial message is infinity
21 val initialMessage = Double.PositiveInfinity
22
23 //The starting vertex from which the search begins (for example, vertex 1)
24 val sourceId: VertexId = 1L
25
26 //Update the vertex value if the new message is less than the current value
27 def vprog(id: VertexId, dist: Double, newDist: Double): Double = {
28   math.min(dist, newDist)
29 }
30
31 //Send a message only if a shorter path is found
32 def sendMessage(edge: EdgeTriplet[Double, Double]): Iterator[(VertexId, Double)] = {
33   if (edge.srcAttr + edge.attr < edge.dstAttr) {
34     Iterator((edge.dstId, edge.srcAttr + edge.attr))
35   } else {
36     Iterator.empty
37   }
38 }
39
40 //Function for merging messages (looking for minimum distances)
41 def messageCombiner(a: Double, b: Double): Double = math.min(a, b)
42
43 Set the initial distance for the original vertex
44 val initialGraph = graph.mapVertices((id, _) =>
45   if (id == sourceId) 0.0 else Double.PositiveInfinity
46 )
```

```
47
48 //Running the Pregel algorithm to find the shortest paths
49 val shortestPaths = initialGraph.pregel(initialMessage)(
50   vprog, sendMessage, messageCombiner
51 )
52
53 //Output of the shortest distances from the original vertex
54 shortestPaths.vertices.collect.foreach { case (id, dist) =>
55   println(s"Vertex $id has distance $dist from source $sourceId")
56 }
```


Result: In this experiment I created a graph with vertices and edges. Then using the Pregel algorithm, I printed shortest paths from a given vertex to all others.

```
Command Prompt - spark-shell
rk.graphx.VertexId, Double)]

scala>

scala> //Function for merging messages (looking for minimum distances)

scala> def messageCombiner(a: Double, b: Double): Double = math.min(a, b)
messageCombiner: (a: Double, b: Double)Double

scala>

scala> Set the initial distance for the original vertex
<console>:1: error: illegal start of simple expression
    Set the initial distance for the original vertex
        ^

scala> val initialGraph = graph.mapVertices((id, _) =>
    |   if (id == sourceId) 0.0 else Double.PositiveInfinity
    | )
initialGraph: org.apache.spark.graphx.Graph[Double,Double] = org.apache.spark.graphx.impl.Graph
Impl@3047ff30

scala>

scala> //Running the Pregel algorithm to find the shortest paths

scala> val shortestPaths = initialGraph.pregel(initialMessage)(
    |   vprog, sendMessage, messageCombiner
    | )
shortestPaths: org.apache.spark.graphx.Graph[Double,Double] = org.apache.spark.graphx.impl.Grap
hImpl@7f46062e

scala>

scala> //Output of the shortest distances from the original vertex

scala> shortestPaths.vertices.collect.foreach { case (id, dist) =>
    |   println(s"Vertex $id has distance $dist from source $sourceId")
    | }
Vertex 1 has distance 0.0 from source 1
Vertex 2 has distance 1.0 from source 1
Vertex 3 has distance 2.0 from source 1
Vertex 4 has distance 3.0 from source 1
Vertex 5 has distance 4.0 from source 1

scala>
```

Figure 5: Execution 3

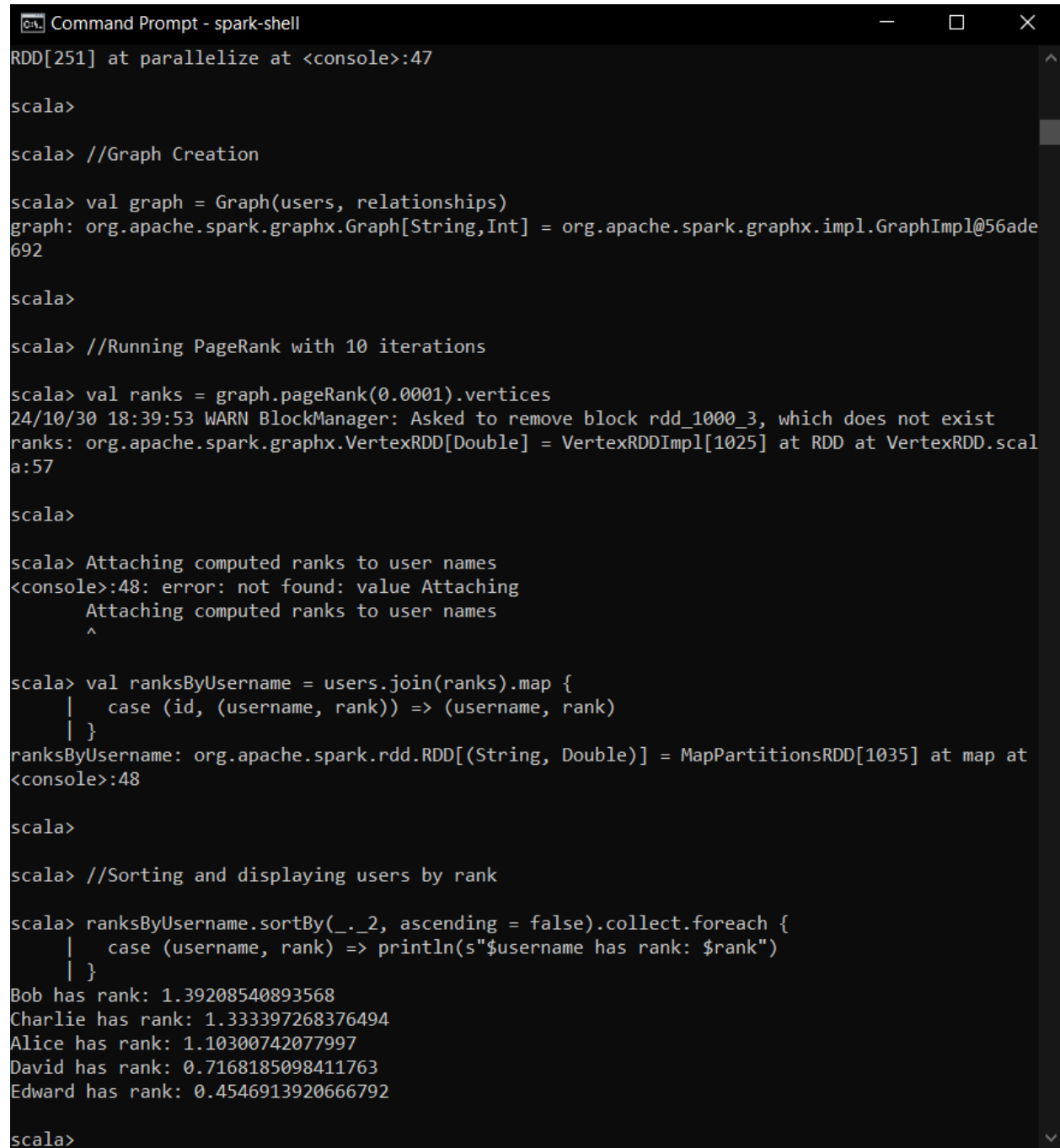
Experiment 4: User Relationship Analysis

Task: Apply the PageRank algorithm to evaluate the importance of each user based on their relationships within the network.

Code:

```
1 import org.apache.spark.graphx._
2 import org.apache.spark.rdd.RDD
3
4 //Define the vertices (VertexId, user name)
5 val users: RDD[(VertexId, String)] = sc.parallelize(Seq(
6   (1L, "Alice"), (2L, "Bob"), (3L, "Charlie"),
7   (4L, "David"), (5L, "Edward")
8 ))
9
10 //Define the edges (source node, target node, link weight)
11 val relationships: RDD[Edge[Int]] = sc.parallelize(Seq(
12   Edge(1L, 2L, 1), Edge(2L, 3L, 1), Edge(3L, 4L, 1),
13   Edge(4L, 5L, 1), Edge(5L, 1L, 1), Edge(3L, 1L, 1),
14   Edge(4L, 2L, 1)
15 ))
16
17 //Graph Creation
18 val graph = Graph(users, relationships)
19
20 //Running PageRank with 10 iterations
21 val ranks = graph.pageRank(0.0001).vertices
22
23 Attaching computed ranks to user names
24 val ranksByUsername = users.join(ranks).map {
25   case (id, (username, rank)) => (username, rank)
26 }
27
28 //Sorting and displaying users by rank
29 ranksByUsername.sortBy(_._2, ascending = false).collect.foreach {
30   case (username, rank) => println(s"$username has rank: $rank")
31 }
```

Result: Here I created a graph where vertices represent users and edges represent their connections. Using the PageRank algorithm, I evaluated the importance of each user based on their relationships. For example, Bob has the highest rank among other users.



```
Command Prompt - spark-shell
RDD[251] at parallelize at <console>:47

scala>

scala> //Graph Creation

scala> val graph = Graph(users, relationships)
graph: org.apache.spark.graphx.Graph[String,Int] = org.apache.spark.graphx.impl.GraphImpl@56ade692

scala>

scala> //Running PageRank with 10 iterations

scala> val ranks = graph.pageRank(0.0001).vertices
24/10/30 18:39:53 WARN BlockManager: Asked to remove block rdd_1000_3, which does not exist
ranks: org.apache.spark.graphx.VertexRDD[Double] = VertexRDDImpl[1025] at RDD at VertexRDD.scala:57

scala>

scala> Attaching computed ranks to user names
<console>:48: error: not found: value Attaching
    Attaching computed ranks to user names
    ^

scala> val ranksByUsername = users.join(ranks).map {
  |   case (id, (username, rank)) => (username, rank)
  | }
ranksByUsername: org.apache.spark.rdd.RDD[(String, Double)] = MapPartitionsRDD[1035] at map at <console>:48

scala>

scala> //Sorting and displaying users by rank

scala> ranksByUsername.sortBy(_._2, ascending = false).collect.foreach {
  |   case (username, rank) => println(s"$username has rank: $rank")
  | }
Bob has rank: 1.39208540893568
Charlie has rank: 1.333397268376494
Alice has rank: 1.10300742077997
David has rank: 0.7168185098411763
Edward has rank: 0.4546913920666792

scala>
```

Figure 6: Execution 4