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 Faut 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

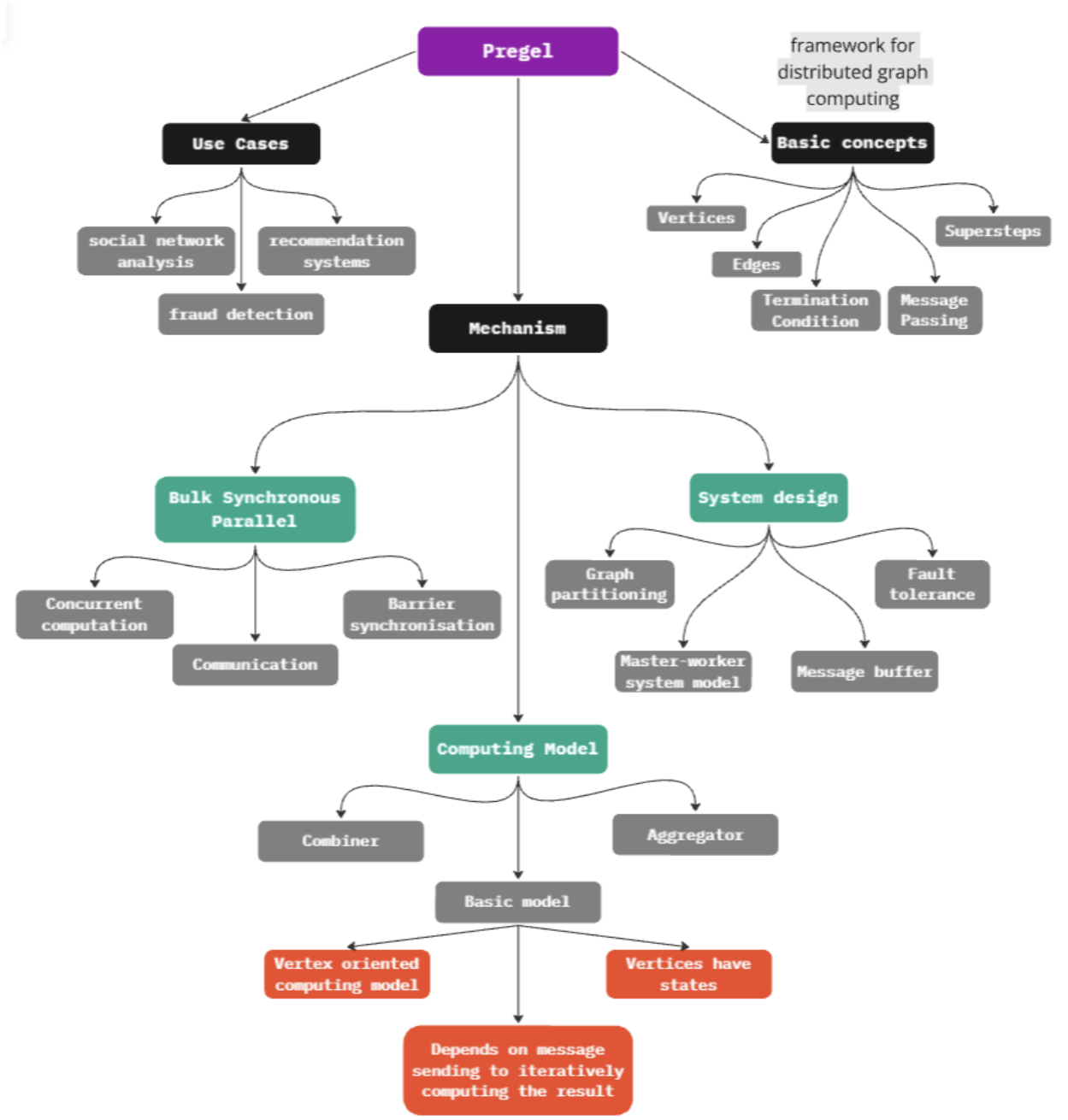

Figure 1: Pregel MindMap

Launching Spark

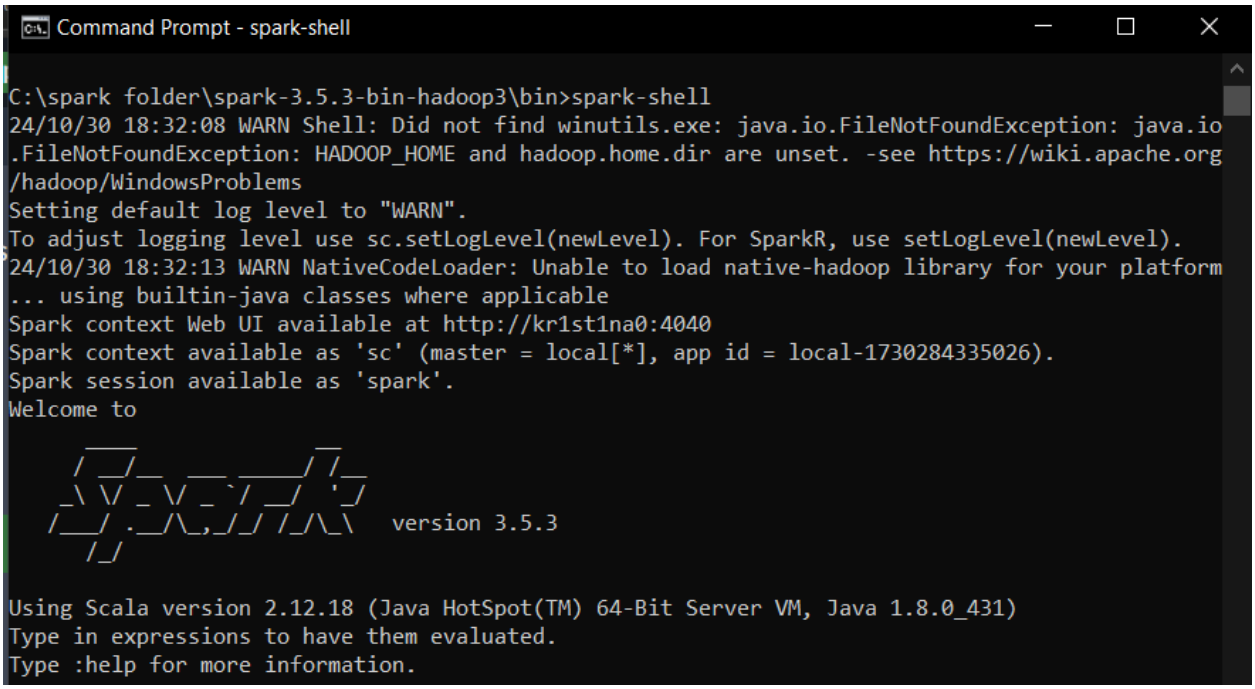

Figure 2: Spark-Shell

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:

```
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

val edgesFile = sc.textFile( “C:/spark folder/spark-3.5.3-bin-hadoop3/bin/graph_data.txt” )

val edges: RDD[Edge[Int]] = edgesFile.map { line =>
  val parts = line.split(" ")
  Edge(parts(0).toLong, parts(1).toLong, 1)
}

val graph = Graph.fromEdges(edges, defaultValue = 1)

graph.vertices.collect.foreach(println)
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.

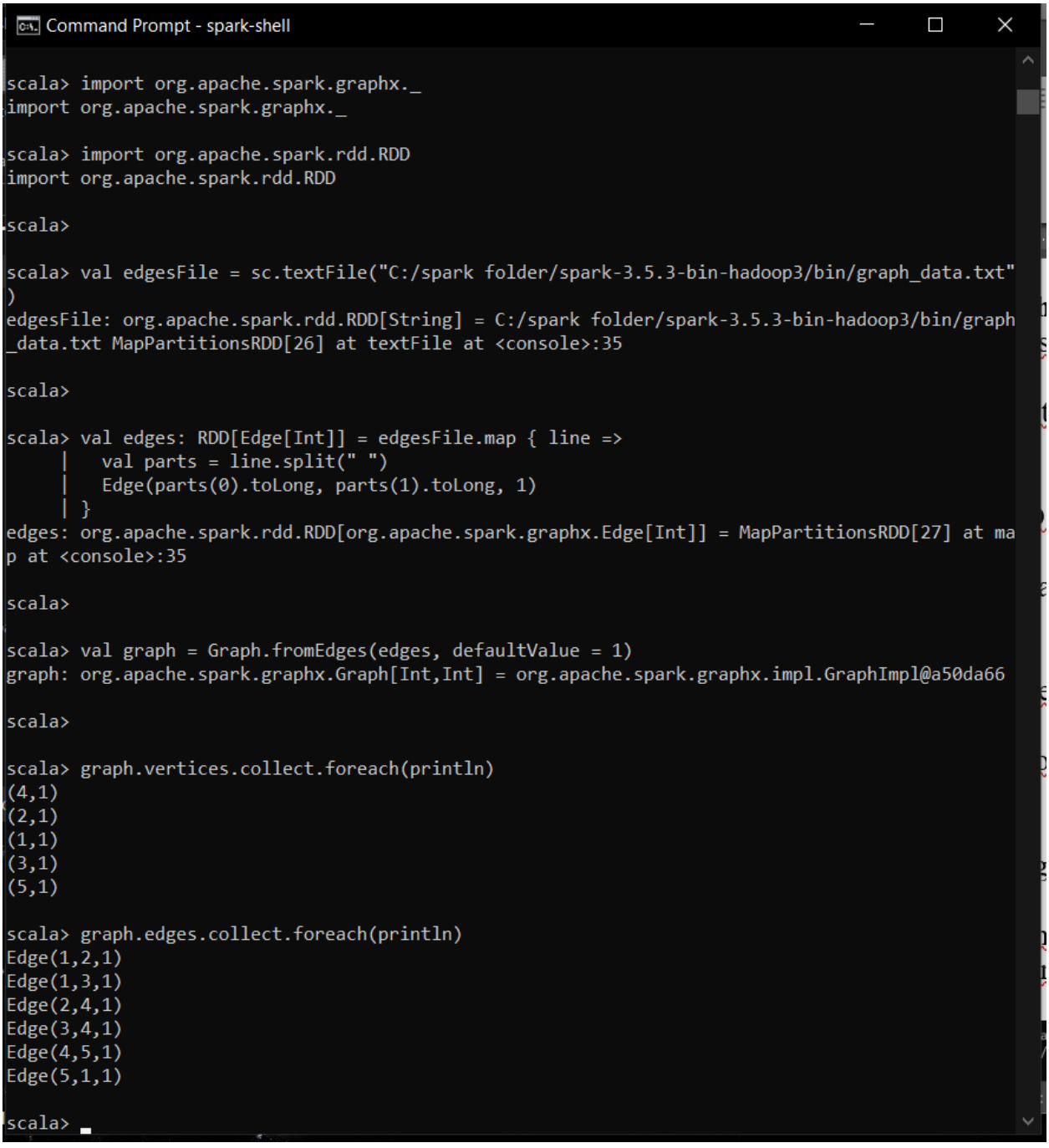

Figure 3: Execution 1

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:

```
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

//Vertex Definition (Vertex ID and Seed Values)
val vertices: RDD[(VertexId, Int)] = sc.parallelize(Seq(
  (1L, 0), (2L, 0), (3L, 0), (4L, 0), (5L, 0)
))

//Rib Definition (Source, Purpose, Weight)
val edges: RDD[Edge[Int]] = sc.parallelize(Seq(
  Edge(1L, 2L, 1), Edge(2L, 3L, 1), Edge(3L, 4L, 1),
  Edge(4L, 5L, 1), Edge(5L, 1L, 1)
))

//Graph Creation
val graph = Graph(vertices, edges)

//A seed value for each vertex (for example, a message)
val initialMessage = 0

//Vertex Program function that updates the vertex value
def vprog(vertexId: VertexId, value: Int, message: Int): Int = math.max(value, message)

//Send messages from the source to target vertices
def sendMessage(edge: EdgeTriplet[Int, Int]): Iterator[(VertexId, Int)] = {
  Iterator((edge.dstId, edge.srcAttr + 1))
}

//Message Merge Feature
def messageCombiner(a: Int, b: Int): Int = math.max(a, b)

//Running the Pregel algorithm
val pregelGraph = graph.pregel(initialMessage, maxIterations = 5)(
  vprog, sendMessage, messageCombiner
)

//Output of final vertex values
pregelGraph.vertices.collect.foreach { case (id, value) =>
  println(s"Vertex $id has final value $value")
}
```

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.

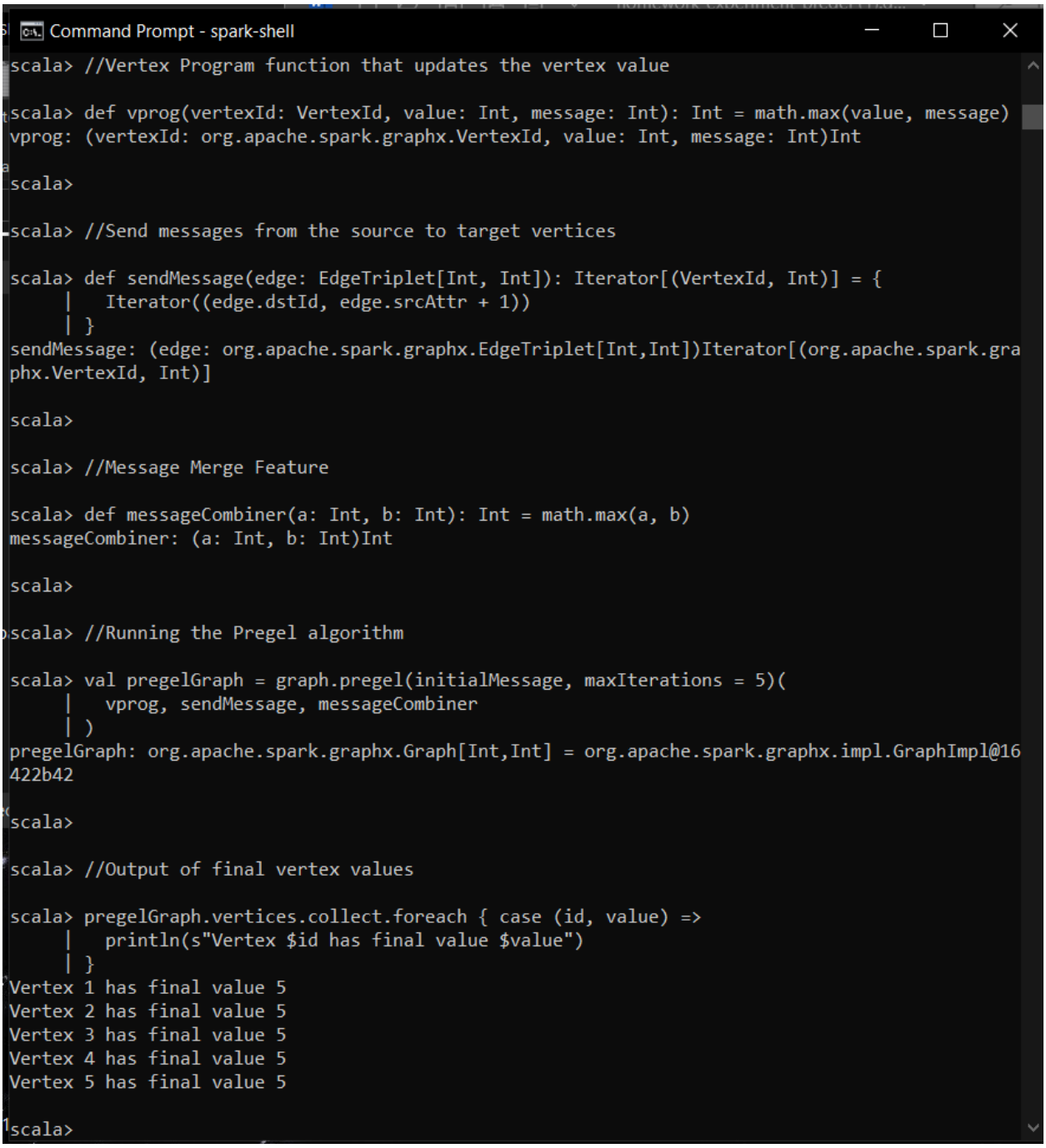

Figure 4: Execution 2

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:

```
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

//Vertex Definition (Vertex ID and Seed Values)
val vertices: RDD[(VertexId, Double)] = sc.parallelize(Seq(
  (1L, Double.PositiveInfinity), (2L, Double.PositiveInfinity),
  (3L, Double.PositiveInfinity), (4L, Double.PositiveInfinity),
  (5L, Double.PositiveInfinity)
))

//Rib Definition (Source, Purpose, Weight)
val edges: RDD[Edge[Double]] = sc.parallelize(Seq(
  Edge(1L, 2L, 1.0), Edge(2L, 3L, 1.0), Edge(3L, 4L, 1.0),
  Edge(4L, 5L, 1.0), Edge(5L, 1L, 1.0)
))

//Graph Creation
val graph = Graph(vertices, edges)

//The initial message is infinity
val initialMessage = Double.PositiveInfinity

//The starting vertex from which the search begins (for example, vertex 1)
val sourceId: VertexId = 1L

//Update the vertex value if the new message is less than the current value
def vprog(id: VertexId, dist: Double, newDist: Double): Double = {
  math.min(dist, newDist)
}

//Send a message only if a shorter path is found
def sendMessage(edge: EdgeTriplet[Double, Double]): Iterator[(VertexId, Double)] = {
  if (edge.srcAttr + edge.attr < edge.dstAttr) {
    Iterator((edge.dstId, edge.srcAttr + edge.attr))
  } else {
    Iterator.empty
  }
}

//Function for merging messages (looking for minimum distances)
def messageCombiner(a: Double, b: Double): Double = math.min(a, b)

Set the initial distance for the original vertex
val initialGraph = graph.mapVertices((id, _) =>
  if (id == sourceId) 0.0 else Double.PositiveInfinity
)
```

```
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.

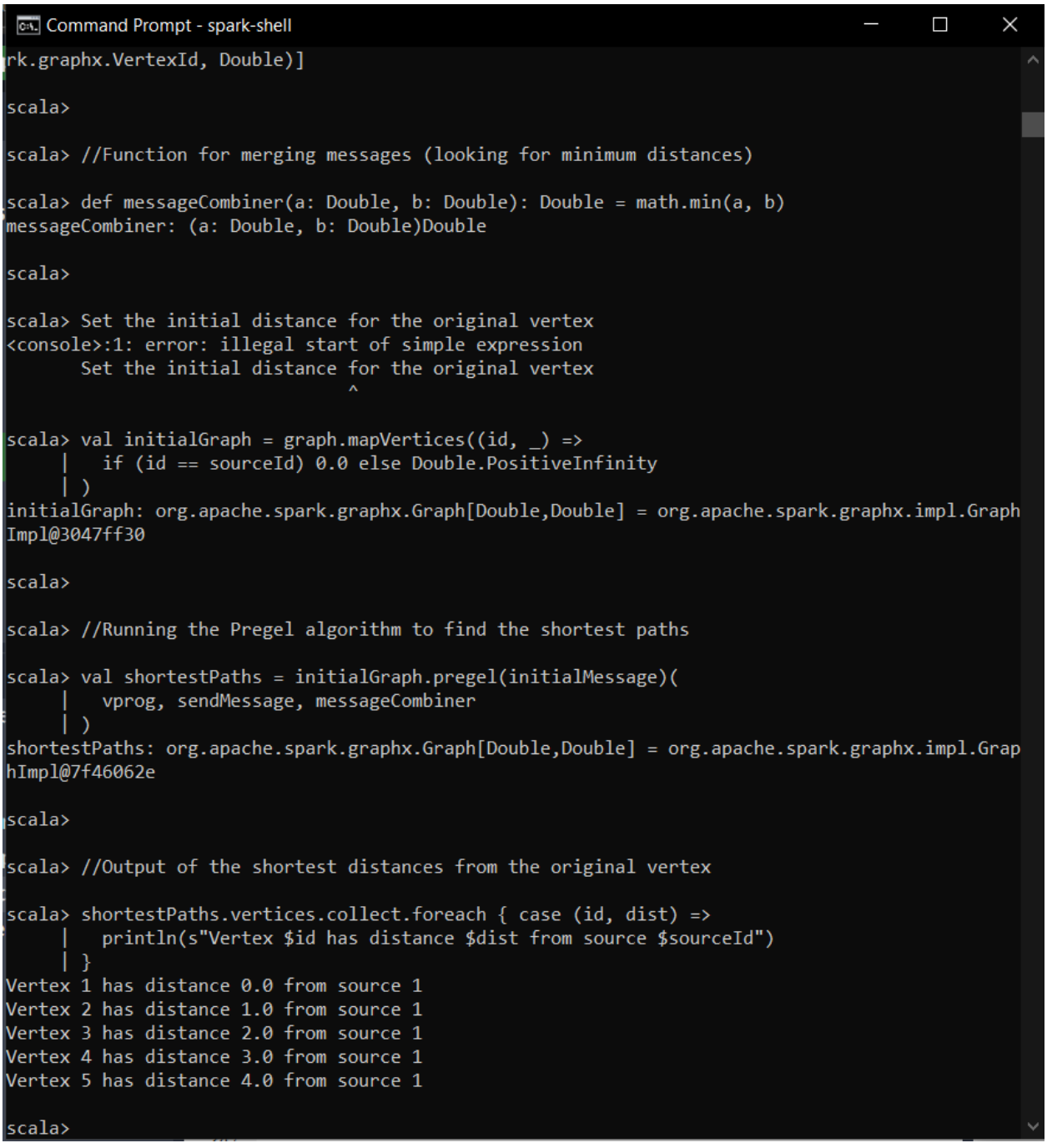

Figure 5: Execution 3

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:

```
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

//Define the vertices (VertexId, user name)
val users: RDD[(VertexId, String)] = sc.parallelize(Seq(
  (1L, "Alice"), (2L, "Bob"), (3L, "Charlie"),
  (4L, "David"), (5L, "Edward")
))

//Define the edges (source node, target node, link weight)
val relationships: RDD[Edge[Int]] = sc.parallelize(Seq(
  Edge(1L, 2L, 1), Edge(2L, 3L, 1), Edge(3L, 4L, 1),
  Edge(4L, 5L, 1), Edge(5L, 1L, 1), Edge(3L, 1L, 1),
  Edge(4L, 2L, 1)
))

//Graph Creation
val graph = Graph(users, relationships)

//Running PageRank with 10 iterations
val ranks = graph.pageRank(0.0001).vertices

Attaching computed ranks to user names
val ranksByUsername = users.join(ranks).map {
  case (id, (username, rank)) => (username, rank)
}

//Sorting and displaying users by rank
ranksByUsername.sortBy(_._2, ascending = false).collect.foreach {
  case (username, rank) => println(s"$username has rank: $rank")
}
```

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.

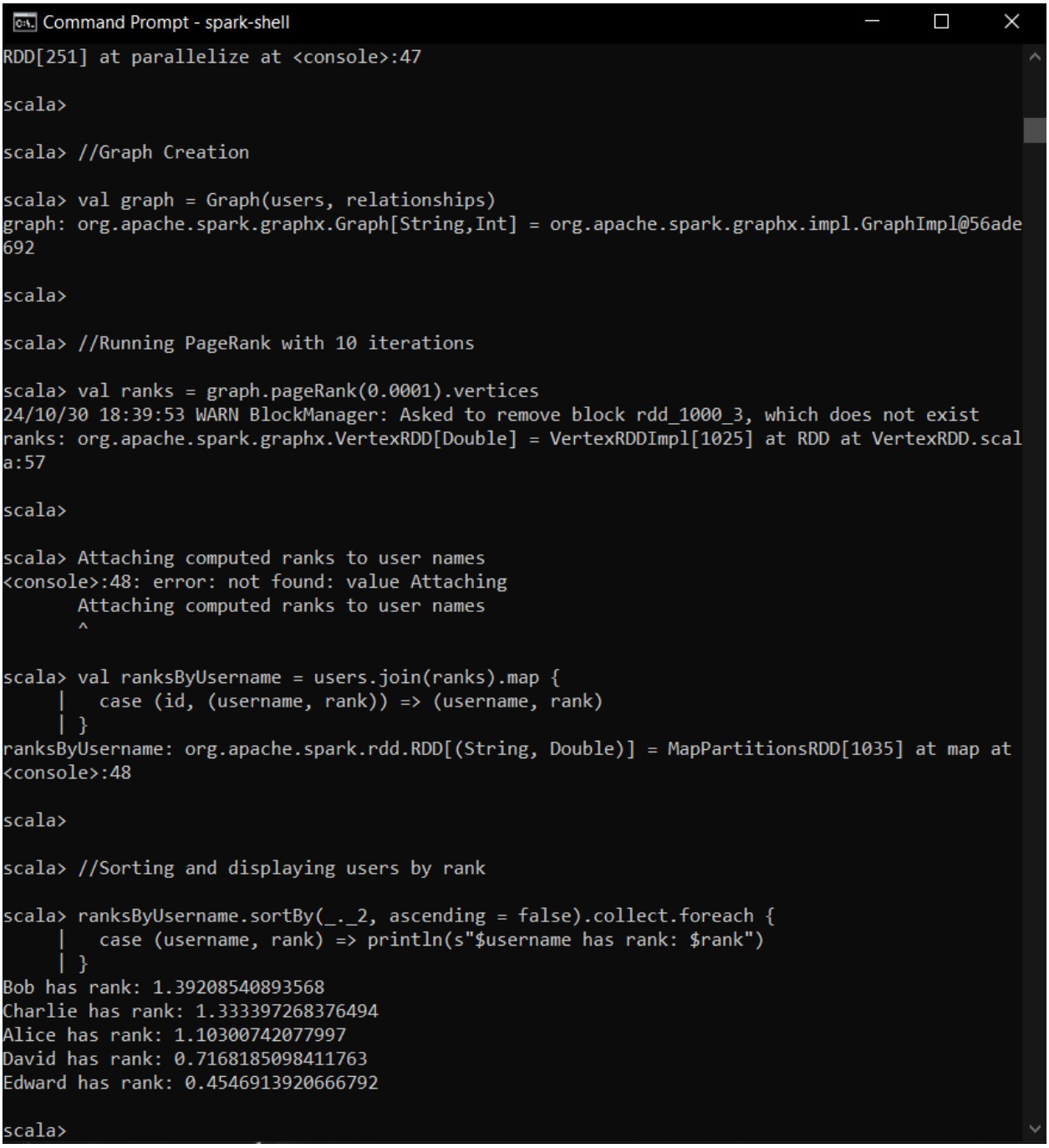

Figure 6: Execution 4