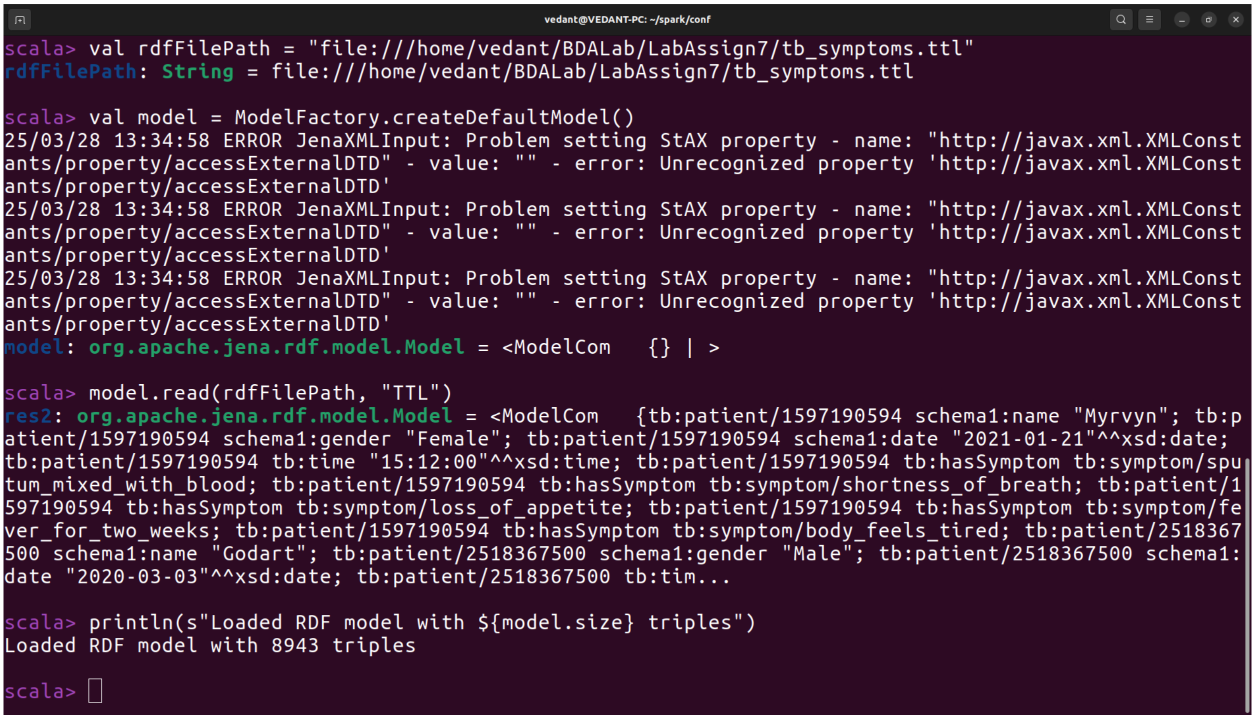
Name : Vedant Rahane

Roll No : MDE2024002

Load the RDF File Using Apache Jena

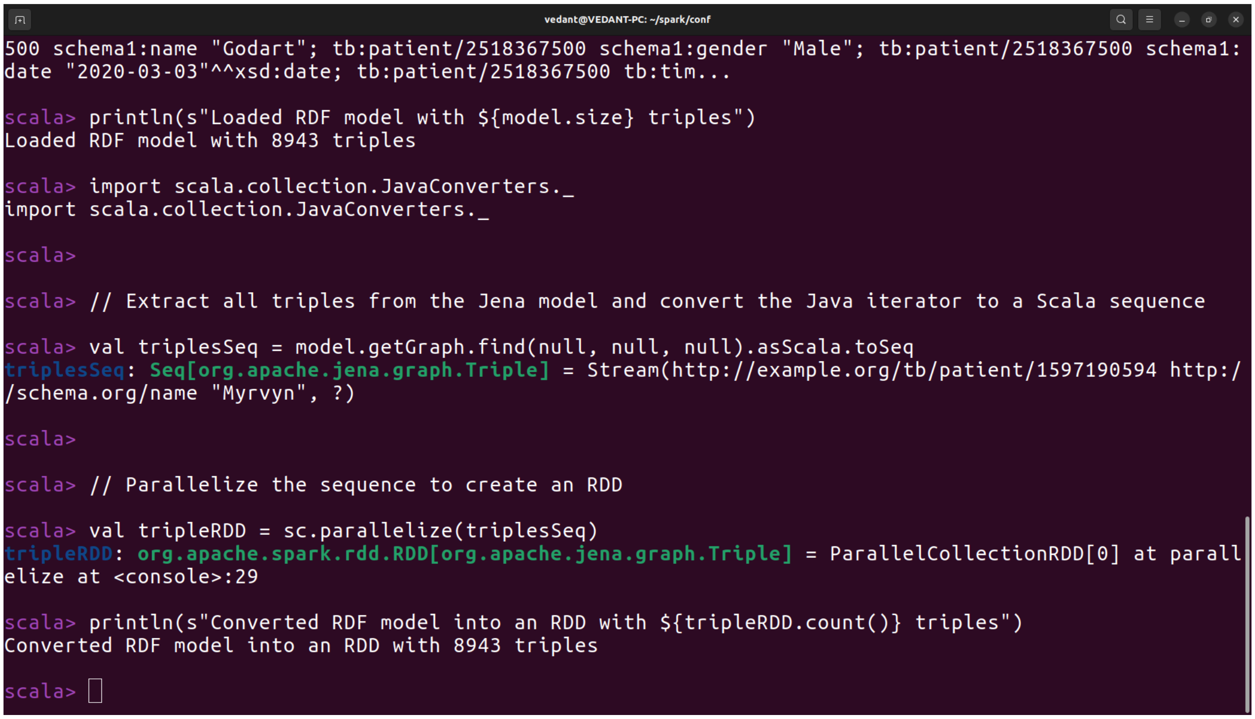
Screenshot -



```
scala> val rdfFilePath = "file:///home/vedant/BDALab/LabAssign7/tb_symptoms.ttl"
rdfFilePath: String = file:///home/vedant/BDALab/LabAssign7/tb_symptoms.ttl

scala> val model = ModelFactory.createDefaultModel()
25/03/28 13:34:58 ERROR JenaXMLInput: Problem setting StAX property - name: "http://javax.xml.XMLConst
ants/property/accessExternalDTD" - value: "" - error: Unrecognized property 'http://javax.xml.XMLConst
ants/property/accessExternalDTD'
25/03/28 13:34:58 ERROR JenaXMLInput: Problem setting StAX property - name: "http://javax.xml.XMLConst
ants/property/accessExternalDTD" - value: "" - error: Unrecognized property 'http://javax.xml.XMLConst
ants/property/accessExternalDTD'
25/03/28 13:34:58 ERROR JenaXMLInput: Problem setting StAX property - name: "http://javax.xml.XMLConst
ants/property/accessExternalDTD" - value: "" - error: Unrecognized property 'http://javax.xml.XMLConst
ants/property/accessExternalDTD'
model: org.apache.jena.rdf.model.Model = <ModelCom   {} | >

scala> model.read(rdfFilePath, "TTL")
res2: org.apache.jena.rdf.model.Model = <ModelCom   {tb:patient/1597190594 schema1:name "Myrvyn"; tb:p
atient/1597190594 schema1:gender "Female"; tb:patient/1597190594 schema1:date "2021-01-21"^^xsd:date;
tb:patient/1597190594 tb:time "15:12:00"^^xsd:time; tb:patient/1597190594 tb:hasSymptom tb:symptom/spu
tum_mixed_with_blood; tb:patient/1597190594 tb:hasSymptom tb:symptom/shortness_of_breath; tb:patient/1
597190594 tb:hasSymptom tb:symptom/loss_of_appetite; tb:patient/1597190594 tb:hasSymptom tb:symptom/fe
ver_for_two_weeks; tb:patient/1597190594 tb:hasSymptom tb:symptom/body_feels_tired; tb:patient/2518367
500 schema1:name "Godart"; tb:patient/2518367500 schema1:gender "Male"; tb:patient/2518367500 schema1:
date "2020-03-03"^^xsd:date; tb:patient/2518367500 tb:tim...

scala> println(s"Loaded RDF model with ${model.size} triples")
Loaded RDF model with 8943 triples

scala> 
```

Convert the RDF Model into an RDD of Triples

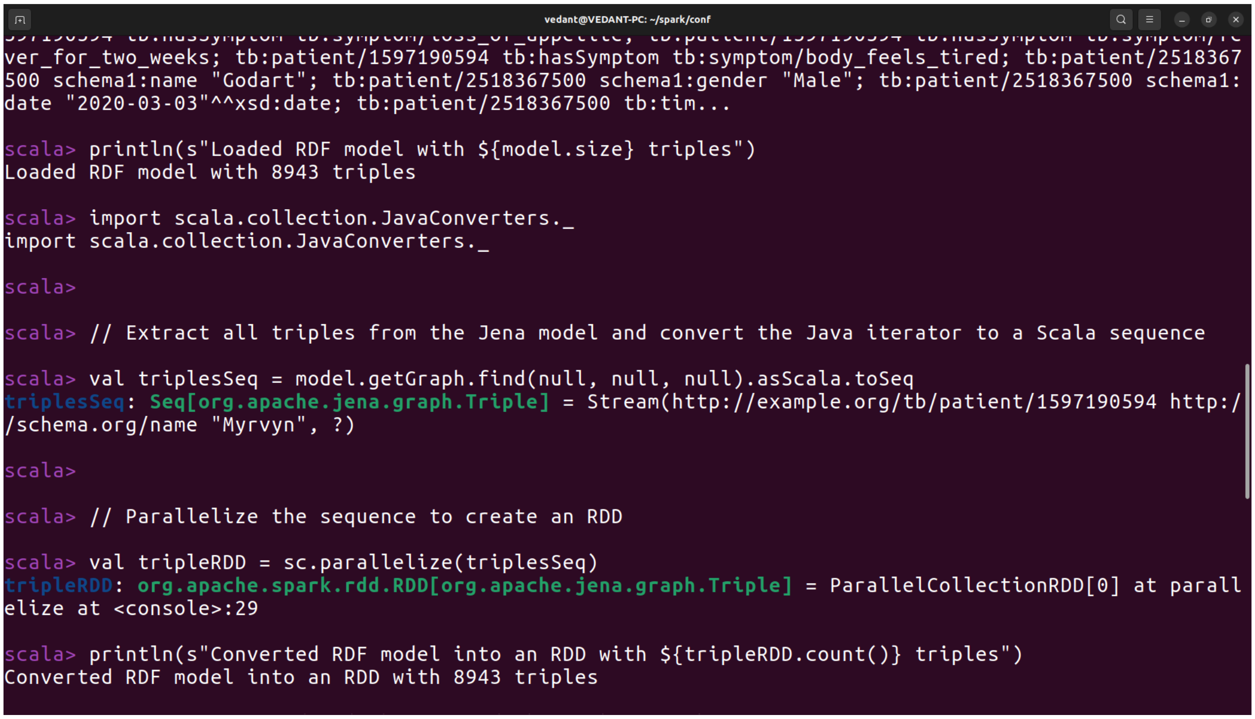
Screenshot -



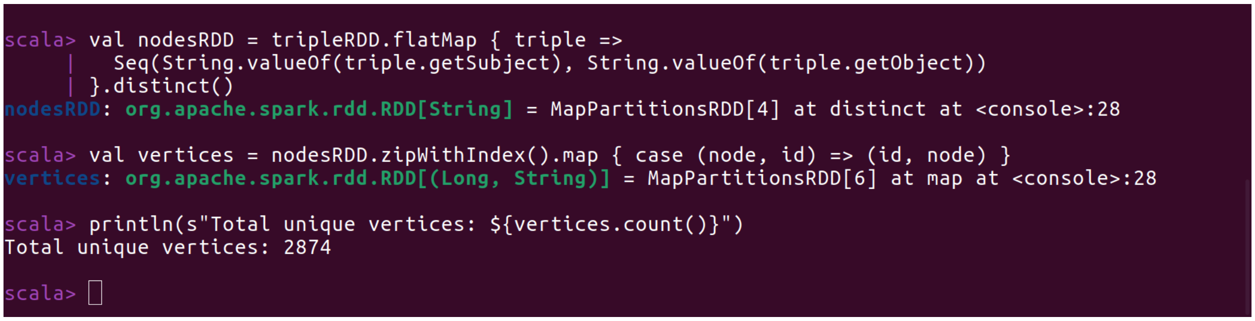
```
500 schema1:name "Godart"; tb:patient/2518367500 schema1:gender "Male"; tb:patient/2518367500 schema1:
date "2020-03-03"^^xsd:date; tb:patient/2518367500 tb:tim...

scala> println(s"Loaded RDF model with ${model.size} triples")
Loaded RDF model with 8943 triples

scala> import scala.collection.JavaConverters._
import scala.collection.JavaConverters._

scala>

scala> // Extract all triples from the Jena model and convert the Java iterator to a Scala sequence

scala> val triplesSeq = model.getGraph.find(null, null, null).asScala.toSeq
triplesSeq: Seq[org.apache.jena.graph.Triple] = Stream(http://example.org/tb/patient/1597190594 http:/
/schema.org/name "Myrvyn", ?)

scala>

scala> // Parallelize the sequence to create an RDD

scala> val tripleRDD = sc.parallelize(triplesSeq)
tripleRDD: org.apache.spark.rdd.RDD[org.apache.jena.graph.Triple] = ParallelCollectionRDD[0] at parall
elize at <console>:29

scala> println(s"Converted RDF model into an RDD with ${tripleRDD.count()} triples")
Converted RDF model into an RDD with 8943 triples

scala> 
```

Create Vertices RDD from RDF Triples

Screenshot -



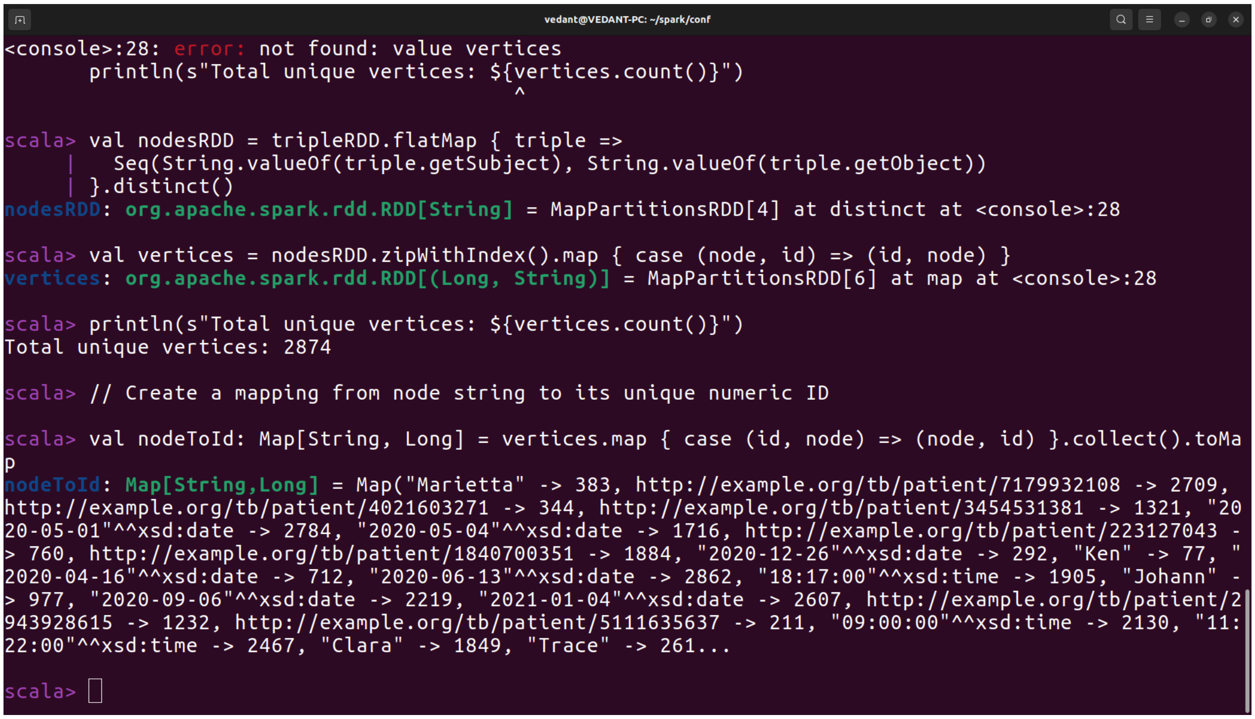
```
597190594 tb:hasSymptom tb:symptom/loss_of_appetite; tb:patient/1597190594 tb:hasSymptom tb:symptom/fe
ver_for_two_weeks; tb:patient/1597190594 tb:hasSymptom tb:symptom/body_feels_tired; tb:patient/2518367
500 schema1:name "Godart"; tb:patient/2518367500 schema1:gender "Male"; tb:patient/2518367500 schema1:
date "2020-03-03"^^xsd:date; tb:patient/2518367500 tb:tim...

scala> println(s"Loaded RDF model with ${model.size} triples")
Loaded RDF model with 8943 triples

scala> import scala.collection.JavaConverters._
import scala.collection.JavaConverters._

scala>

scala> // Extract all triples from the Jena model and convert the Java iterator to a Scala sequence

scala> val triplesSeq = model.getGraph.find(null, null, null).asScala.toSeq
triplesSeq: Seq[org.apache.jena.graph.Triple] = Stream(http://example.org/tb/patient/1597190594 http:/
/schema.org/name "Myrvyn", ?)

scala>

scala> // Parallelize the sequence to create an RDD

scala> val tripleRDD = sc.parallelize(triplesSeq)
tripleRDD: org.apache.spark.rdd.RDD[org.apache.jena.graph.Triple] = ParallelCollectionRDD[0] at parall
elize at <console>:29

scala> println(s"Converted RDF model into an RDD with ${tripleRDD.count()} triples")
Converted RDF model into an RDD with 8943 triples
```



```
scala> val nodesRDD = tripleRDD.flatMap { triple =>
     |   Seq(String.valueOf(triple.getSubject), String.valueOf(triple.getObject))
     | }.distinct()
nodesRDD: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[4] at distinct at <console>:28

scala> val vertices = nodesRDD.zipWithIndex().map { case (node, id) => (id, node) }
vertices: org.apache.spark.rdd.RDD[(Long, String)] = MapPartitionsRDD[6] at map at <console>:28

scala> println(s"Total unique vertices: ${vertices.count()}")
Total unique vertices: 2874

scala> 
```

Create a Mapping from Node URI to Vertex ID

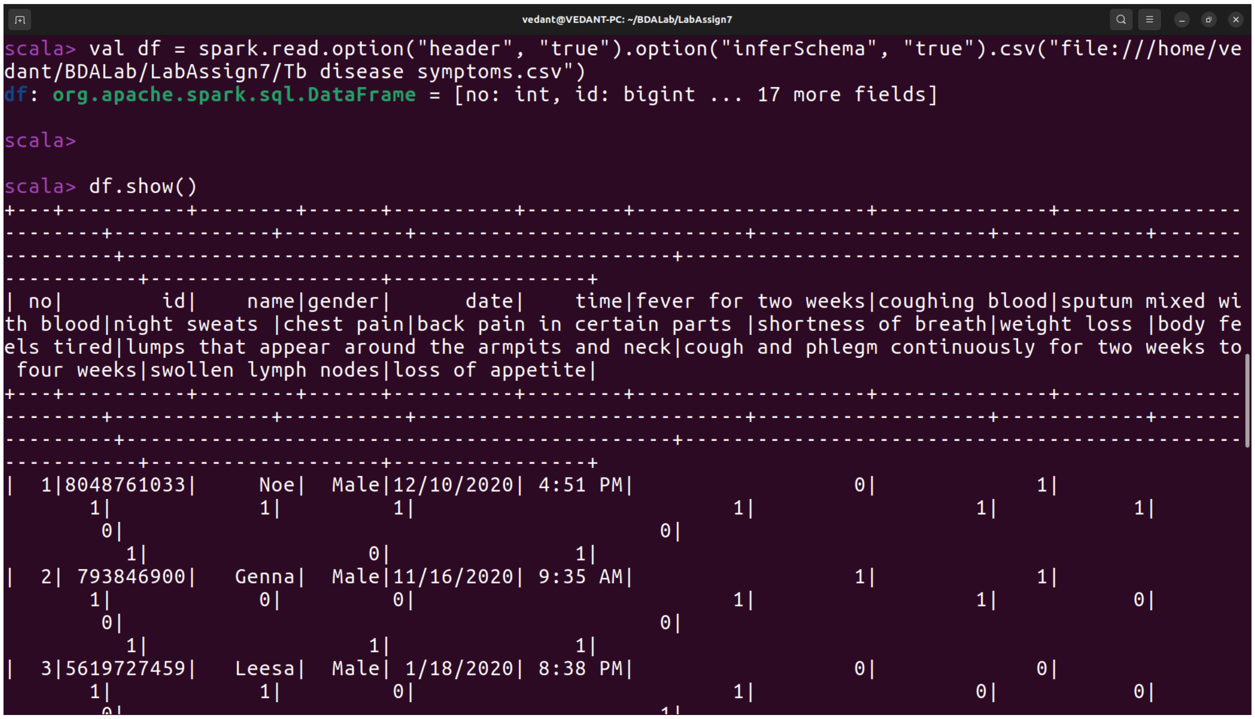
Screenshot -



```
<console>:28: error: not found: value vertices
       println(s"Total unique vertices: ${vertices.count()}")
                                          ^

scala> val nodesRDD = tripleRDD.flatMap { triple =>
     |   Seq(String.valueOf(triple.getSubject), String.valueOf(triple.getObject))
     | }.distinct()
nodesRDD: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[4] at distinct at <console>:28

scala> val vertices = nodesRDD.zipWithIndex().map { case (node, id) => (id, node) }
vertices: org.apache.spark.rdd.RDD[(Long, String)] = MapPartitionsRDD[6] at map at <console>:28

scala> println(s"Total unique vertices: ${vertices.count()}")
Total unique vertices: 2874

scala> // Create a mapping from node string to its unique numeric ID

scala> val nodeToId: Map[String, Long] = vertices.map { case (id, node) => (node, id) }.collect().toMa
p
nodeToId: Map[String,Long] = Map("Marietta" -> 383, http://example.org/tb/patient/7179932108 -> 2709,
http://example.org/tb/patient/4021603271 -> 344, http://example.org/tb/patient/3454531381 -> 1321, "20
20-05-01"^^xsd:date -> 2784, "2020-05-04"^^xsd:date -> 1716, http://example.org/tb/patient/223127043 -
> 760, http://example.org/tb/patient/1840700351 -> 1884, "2020-12-26"^^xsd:date -> 292, "Ken" -> 77, "
2020-04-16"^^xsd:date -> 712, "2020-06-13"^^xsd:date -> 2862, "18:17:00"^^xsd:time -> 1905, "Johann" -
> 977, "2020-09-06"^^xsd:date -> 2219, "2021-01-04"^^xsd:date -> 2607, http://example.org/tb/patient/2
943928615 -> 1232, http://example.org/tb/patient/5111635637 -> 211, "09:00:00"^^xsd:time -> 2130, "11:
22:00"^^xsd:time -> 2467, "Clara" -> 1849, "Trace" -> 261...

scala> 
```

Create GraphX Graph -

Screenshot -



```
scala> val df = spark.read.option("header", "true").option("inferSchema", "true").csv("file:///home/ve
dant/BDALab/LabAssign7/Tb disease symptoms.csv")
df: org.apache.spark.sql.DataFrame = [no: int, id: bigint ... 17 more fields]

scala> 

scala> df.show()
+---+----------+--------+------+----------+--------+-------------------+--------------+---------------
--------+-------------+----------+---------------------------+-------------------+------------+-------
---------+---------------------------------------------+----------------------------------------------
-----------+-------------------+----------------+
| no|        id|    name|gender|      date|    time|fever for two weeks|coughing blood|sputum mixed wi
th blood|night sweats |chest pain|back pain in certain parts |shortness of breath|weight loss |body fe
els tired|lumps that appear around the armpits and neck|cough and phlegm continuously for two weeks to
 four weeks|swollen lymph nodes|loss of appetite|
+---+----------+--------+------+----------+--------+-------------------+--------------+---------------
--------+-------------+----------+---------------------------+-------------------+------------+-------
---------+---------------------------------------------+----------------------------------------------
-----------+-------------------+----------------+
|  1|8048761033|     Noe|  Male|12/10/2020| 4:51 PM|                  0|             1|
       1|            1|         1|                          1|                  1|           1|
        0|                                            0|
          1|                  0|               1|
|  2| 793846900|   Genna|  Male|11/16/2020| 9:35 AM|                  1|             1|
       1|            0|         0|                          1|                  1|           0|
        0|                                            0|
          1|                  1|               1|
|  3|5619727459|   Leesa|  Male| 1/18/2020| 8:38 PM|                  0|             0|
       1|            1|         0|                          1|                  0|           0|
        0|                                            1|
```

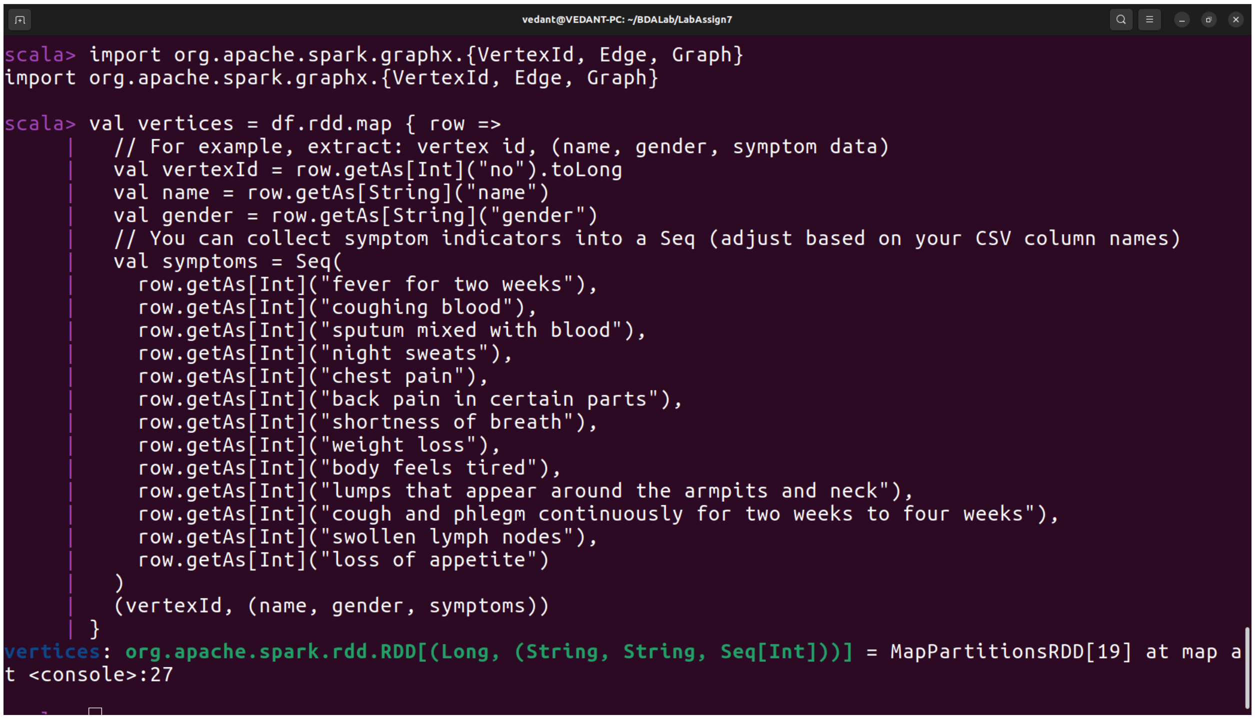
Create Vertices and Edges for the GraphX Graph :

Convert the DataFrame to an RDD and Create Vertices

Code -

```
import org.apache.spark.graphx.{VertexId, Edge, Graph}

// Create an RDD of vertices from the cleaned DataFrame.
val vertices = cleanedDF.rdd.map { row =>
  // Extract: vertex id, (name, gender, symptom data)
  val vertexId = row.getAs[Int]("no").toLong
  val name = row.getAs[String]("name")
  val gender = row.getAs[String]("gender")

  // Collect symptom indicators (adjusted column names)
  val symptoms = Seq(
          row.getAs[Int]("fever_for_two_weeks"),
          row.getAs[Int]("coughing_blood"),
          row.getAs[Int]("sputum_mixed_with_blood"),
          row.getAs[Int]("night_sweats"),
          row.getAs[Int]("chest_pain"),
          row.getAs[Int]("back_pain_in_certain_parts"),
          row.getAs[Int]("shortness_of_breath"),
          row.getAs[Int]("weight_loss"),
          row.getAs[Int]("body_feels_tired"),
          row.getAs[Int]("lumps_that_appear_around_the_armpits_and_neck"),
          row.getAs[Int]("cough_and_phlegm_continuously_for_two_weeks_to_four_weeks"),
          row.getAs[Int]("swollen_lymph_nodes"),
          row.getAs[Int]("loss_of_appetite")
  )

  (vertexId, (name, gender, symptoms))
}
```

Screenshot -



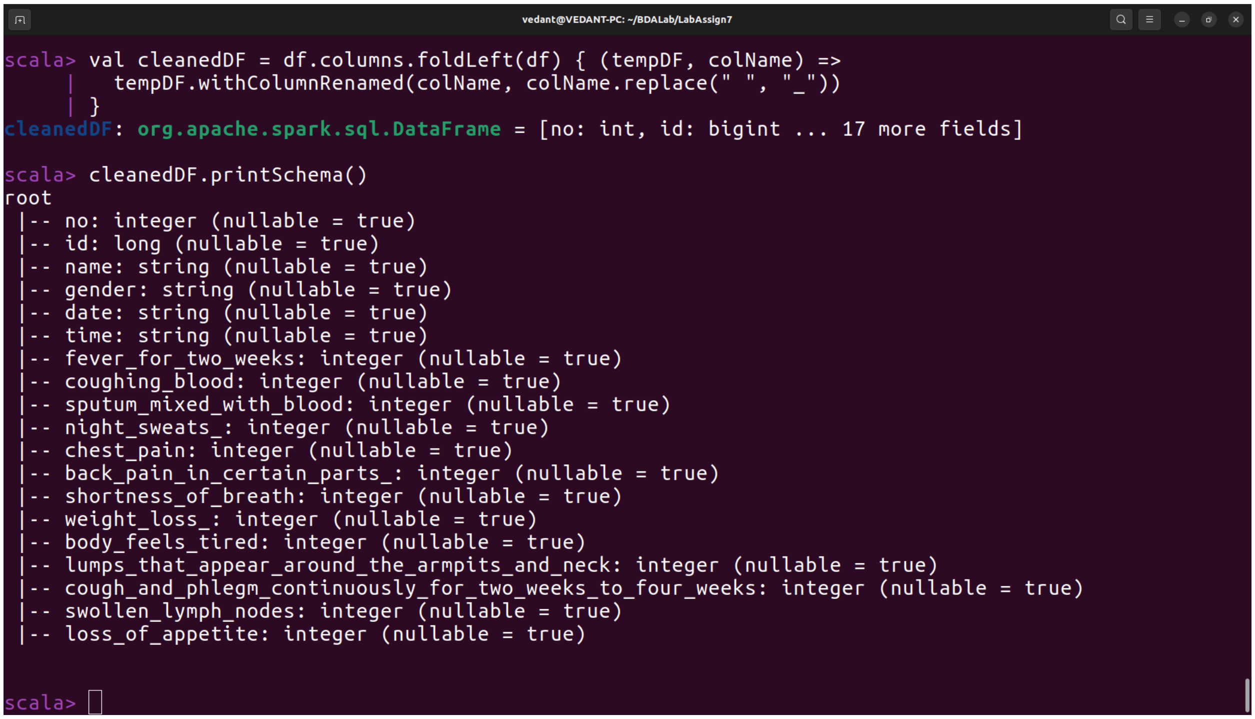
```
scala> import org.apache.spark.graphx.{VertexId, Edge, Graph}
import org.apache.spark.graphx.{VertexId, Edge, Graph}

scala> val vertices = df.rdd.map { row =>
     |   // For example, extract: vertex id, (name, gender, symptom data)
     |   val vertexId = row.getAs[Int]("no").toLong
     |   val name = row.getAs[String]("name")
     |   val gender = row.getAs[String]("gender")
     |   // You can collect symptom indicators into a Seq (adjust based on your CSV column names)
     |   val symptoms = Seq(
     |     row.getAs[Int]("fever for two weeks"),
     |     row.getAs[Int]("coughing blood"),
     |     row.getAs[Int]("sputum mixed with blood"),
     |     row.getAs[Int]("night sweats"),
     |     row.getAs[Int]("chest pain"),
     |     row.getAs[Int]("back pain in certain parts"),
     |     row.getAs[Int]("shortness of breath"),
     |     row.getAs[Int]("weight loss"),
     |     row.getAs[Int]("body feels tired"),
     |     row.getAs[Int]("lumps that appear around the armpits and neck"),
     |     row.getAs[Int]("cough and phlegm continuously for two weeks to four weeks"),
     |     row.getAs[Int]("swollen lymph nodes"),
     |     row.getAs[Int]("loss of appetite")
     |   )
     |   (vertexId, (name, gender, symptoms))
     | }
vertices: org.apache.spark.rdd.RDD[(Long, (String, String, Seq[Int]))] = MapPartitionsRDD[19] at map a
t <console>:27
```

Clean Column Names, Replace space with Underscore

Code -

```
val cleanedDF = df.columns.foldLeft(df)((tempDF, colName) =>
  tempDF.withColumnRenamed(colName, colName.trim.replaceAll("\\s+", "_").replaceAll("^_|_$", "").toLowerCase))

// Show schema after renaming
cleanedDF.printSchema()
```

Screenshot -



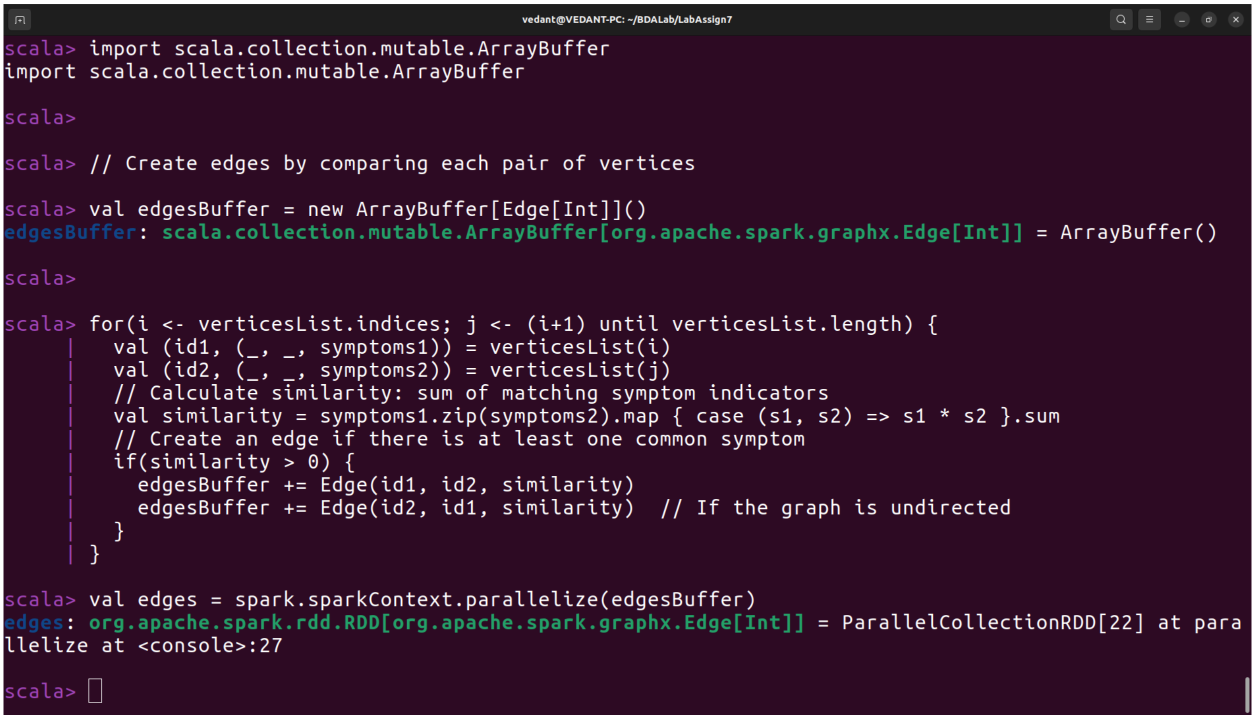
```
scala> val cleanedDF = df.columns.foldLeft(df) { (tempDF, colName) =>
     |   tempDF.withColumnRenamed(colName, colName.replace(" ", "_"))
     | }
cleanedDF: org.apache.spark.sql.DataFrame = [no: int, id: bigint ... 17 more fields]

scala> cleanedDF.printSchema()
root
 |-- no: integer (nullable = true)
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- date: string (nullable = true)
 |-- time: string (nullable = true)
 |-- fever_for_two_weeks: integer (nullable = true)
 |-- coughing_blood: integer (nullable = true)
 |-- sputum_mixed_with_blood: integer (nullable = true)
 |-- night_sweats_: integer (nullable = true)
 |-- chest_pain: integer (nullable = true)
 |-- back_pain_in_certain_parts_: integer (nullable = true)
 |-- shortness_of_breath: integer (nullable = true)
 |-- weight_loss_: integer (nullable = true)
 |-- body_feels_tired: integer (nullable = true)
 |-- lumps_that_appear_around_the_armpits_and_neck: integer (nullable = true)
 |-- cough_and_phlegm_continuously_for_two_weeks_to_four_weeks: integer (nullable = true)
 |-- swollen_lymph_nodes: integer (nullable = true)
 |-- loss_of_appetite: integer (nullable = true)


scala> 
```

Create an RDD of Edges

Code -

```
// Create a list of vertices (this approach works for a small dataset)
val verticesList = vertices.collect()
import scala.collection.mutable.ArrayBuffer

// Create edges by comparing each pair of vertices
val edgesBuffer = new ArrayBuffer[Edge[Int]]()

for(i <- verticesList.indices; j <- (i+1) until verticesList.length) {
  val (id1, (_, _, symptoms1)) = verticesList(i)
  val (id2, (_, _, symptoms2)) = verticesList(j)
  // Calculate similarity: sum of matching symptom indicators
  val similarity = symptoms1.zip(symptoms2).map { case (s1, s2) => s1 * s2 }.sum
  // Create an edge if there is at least one common symptom
  if(similarity > 0) {
        edgesBuffer += Edge(id1, id2, similarity)
        edgesBuffer += Edge(id2, id1, similarity)  // If the graph is undirected
  }
}
val edges = spark.sparkContext.parallelize(edgesBuffer)
```

Screenshot -



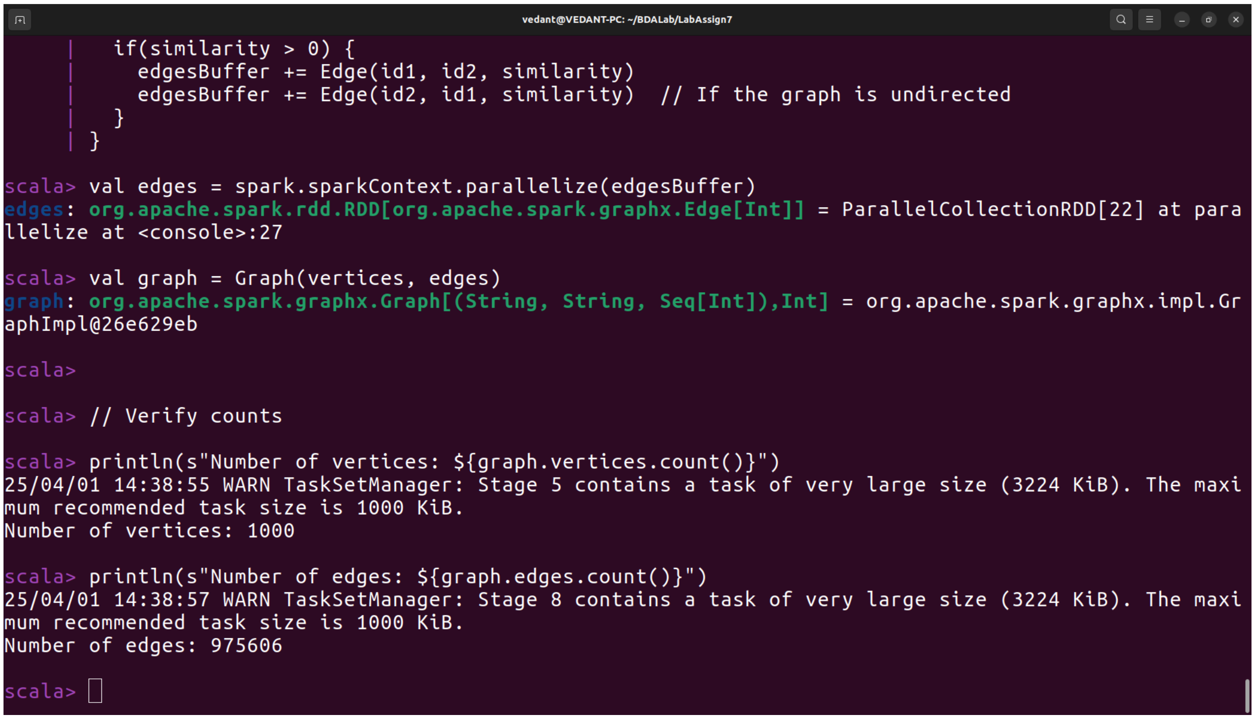
```
scala> import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.ArrayBuffer

scala>

scala> // Create edges by comparing each pair of vertices

scala> val edgesBuffer = new ArrayBuffer[Edge[Int]]()
edgesBuffer: scala.collection.mutable.ArrayBuffer[org.apache.spark.graphx.Edge[Int]] = ArrayBuffer()

scala>

scala> for(i <- verticesList.indices; j <- (i+1) until verticesList.length) {
     |   val (id1, (_, _, symptoms1)) = verticesList(i)
     |   val (id2, (_, _, symptoms2)) = verticesList(j)
     |   // Calculate similarity: sum of matching symptom indicators
     |   val similarity = symptoms1.zip(symptoms2).map { case (s1, s2) => s1 * s2 }.sum
     |   // Create an edge if there is at least one common symptom
     |   if(similarity > 0) {
     |     edgesBuffer += Edge(id1, id2, similarity)
     |     edgesBuffer += Edge(id2, id1, similarity)  // If the graph is undirected
     |   }
     | }

scala> val edges = spark.sparkContext.parallelize(edgesBuffer)
edges: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Int]] = ParallelCollectionRDD[22] at para
llelize at <console>:27

scala> 
```

Build the GraphX Graph

Code -

```
val graph = Graph(vertices, edges)

// Verify counts
println(s"Number of vertices: ${graph.vertices.count()}")
println(s"Number of edges: ${graph.edges.count()}")
```

Screenshot -



```
     |   if(similarity > 0) {
     |     edgesBuffer += Edge(id1, id2, similarity)
     |     edgesBuffer += Edge(id2, id1, similarity)  // If the graph is undirected
     |   }
     | }

scala> val edges = spark.sparkContext.parallelize(edgesBuffer)
edges: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Int]] = ParallelCollectionRDD[22] at para
llelize at <console>:27

scala> val graph = Graph(vertices, edges)
graph: org.apache.spark.graphx.Graph[(String, String, Seq[Int]),Int] = org.apache.spark.graphx.impl.Gr
aphImpl@26e629eb

scala>

scala> // Verify counts

scala> println(s"Number of vertices: ${graph.vertices.count()}")
25/04/01 14:38:55 WARN TaskSetManager: Stage 5 contains a task of very large size (3224 KiB). The maxi
mum recommended task size is 1000 KiB.
Number of vertices: 1000

scala> println(s"Number of edges: ${graph.edges.count()}")
25/04/01 14:38:57 WARN TaskSetManager: Stage 8 contains a task of very large size (3224 KiB). The maxi
mum recommended task size is 1000 KiB.
Number of edges: 975606

scala> 
```

Part 3: Graph Operations :

a) **PageRank**

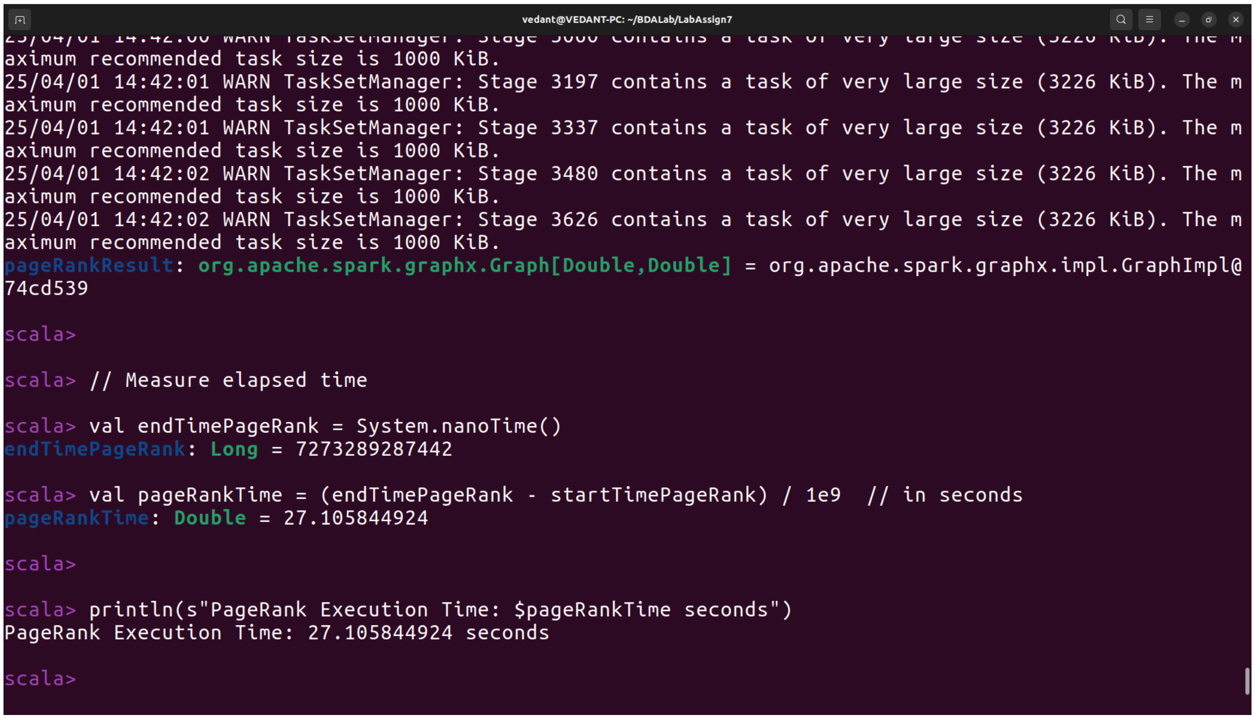
PageRank is used to determine the most influential nodes in a graph based on their connections. For this, you can apply `graph.pageRank` on the created graph.

Code -

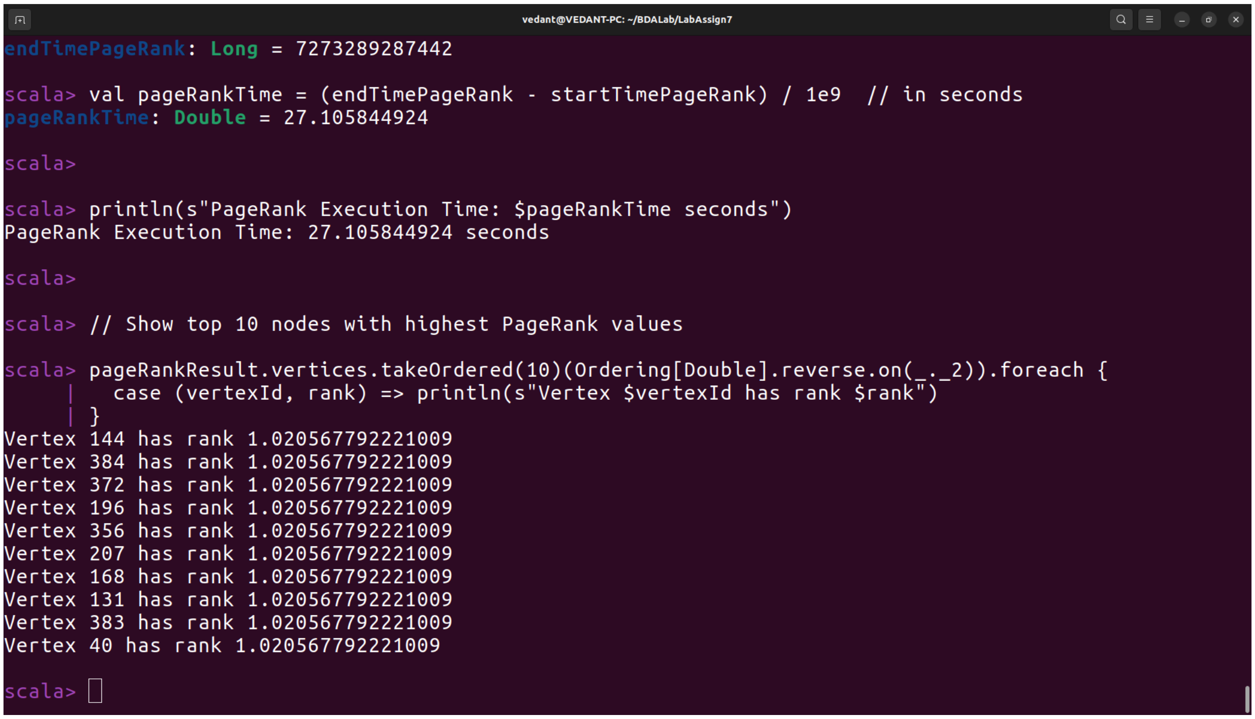
```
import org.apache.spark.graphx.PageRank

// Measure execution time for PageRank
val startTimePageRank = System.nanoTime()

// Perform PageRank
val pageRankResult = graph.pageRank(0.0001)  // Convergence threshold

// Measure elapsed time
val endTimePageRank = System.nanoTime()
val pageRankTime = (endTimePageRank - startTimePageRank) / 1e9  // in seconds

println(s"PageRank Execution Time: $pageRankTime seconds")
```

```
// Show top 10 nodes with highest PageRank values
pageRankResult.vertices.takeOrdered(10)(Ordering[Double].reverse.on(_._2)).foreach {
  case (vertexId, rank) => println(s"Vertex $vertexId has rank $rank")
}
```

Screenshot -



```
25/04/01 14:42:00 WARN TaskSetManager: Stage 3060 contains a task of very large size (3226 KiB). The m
aximum recommended task size is 1000 KiB.
25/04/01 14:42:01 WARN TaskSetManager: Stage 3197 contains a task of very large size (3226 KiB). The m
aximum recommended task size is 1000 KiB.
25/04/01 14:42:01 WARN TaskSetManager: Stage 3337 contains a task of very large size (3226 KiB). The m
aximum recommended task size is 1000 KiB.
25/04/01 14:42:02 WARN TaskSetManager: Stage 3480 contains a task of very large size (3226 KiB). The m
aximum recommended task size is 1000 KiB.
25/04/01 14:42:02 WARN TaskSetManager: Stage 3626 contains a task of very large size (3226 KiB). The m
aximum recommended task size is 1000 KiB.
pageRankResult: org.apache.spark.graphx.Graph[Double,Double] = org.apache.spark.graphx.impl.GraphImpl@
74cd539

scala>

scala> // Measure elapsed time

scala> val endTimePageRank = System.nanoTime()
endTimePageRank: Long = 7273289287442

scala> val pageRankTime = (endTimePageRank - startTimePageRank) / 1e9  // in seconds
pageRankTime: Double = 27.105844924

scala>

scala> println(s"PageRank Execution Time: $pageRankTime seconds")
PageRank Execution Time: 27.105844924 seconds

scala>
```



```
endTimePageRank: Long = 7273289287442

scala> val pageRankTime = (endTimePageRank - startTimePageRank) / 1e9  // in seconds
pageRankTime: Double = 27.105844924

scala>

scala> println(s"PageRank Execution Time: $pageRankTime seconds")
PageRank Execution Time: 27.105844924 seconds

scala>

scala> // Show top 10 nodes with highest PageRank values

scala> pageRankResult.vertices.takeOrdered(10)(Ordering[Double].reverse.on(_._2)).foreach {
     |   case (vertexId, rank) => println(s"Vertex $vertexId has rank $rank")
     | }
Vertex 144 has rank 1.020567792221009
Vertex 384 has rank 1.020567792221009
Vertex 372 has rank 1.020567792221009
Vertex 196 has rank 1.020567792221009
Vertex 356 has rank 1.020567792221009
Vertex 207 has rank 1.020567792221009
Vertex 168 has rank 1.020567792221009
Vertex 131 has rank 1.020567792221009
Vertex 383 has rank 1.020567792221009
Vertex 40 has rank 1.020567792221009

scala> 
```

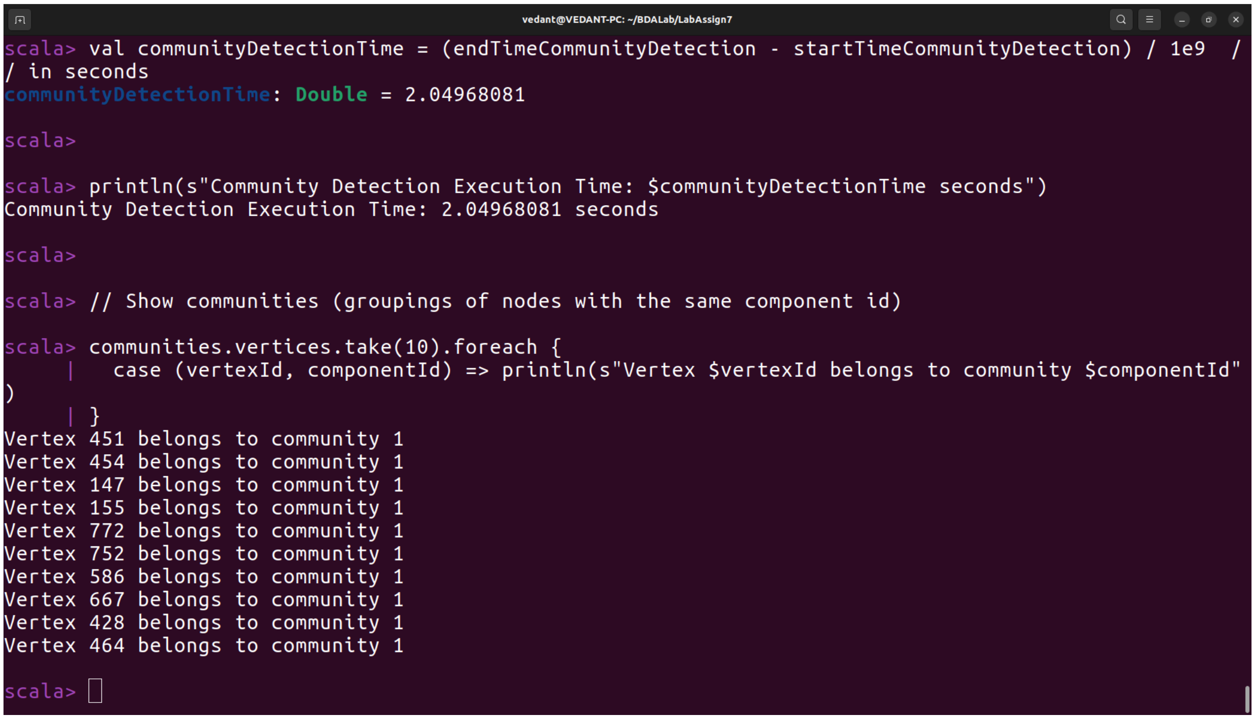
b) Community Detection

Community detection can be done using `GraphX`'s `connectedComponents` or more advanced methods like `Label Propagation`. For simplicity, let's use `connectedComponents` here:

Code -

```
// Measure execution time for Community Detection
val startTimeCommunityDetection = System.nanoTime()

// Perform community detection
val communities = graph.connectedComponents()

// Measure elapsed time
val endTimeCommunityDetection = System.nanoTime()
val communityDetectionTime = (endTimeCommunityDetection - startTimeCommunityDetection)
/ 1e9  // in seconds

println(s"Community Detection Execution Time: $communityDetectionTime seconds")

// Show communities (groupings of nodes with the same component id)
communities.vertices.take(10).foreach {
  case (vertexId, componentId) => println(s"Vertex $vertexId belongs to community
$componentId")
}
```

Screenshot -



```
scala> val communityDetectionTime = (endTimeCommunityDetection - startTimeCommunityDetection) / 1e9  /
/ in seconds
communityDetectionTime: Double = 2.04968081

scala>

scala> println(s"Community Detection Execution Time: $communityDetectionTime seconds")
Community Detection Execution Time: 2.04968081 seconds

scala>

scala> // Show communities (groupings of nodes with the same component id)

scala> communities.vertices.take(10).foreach {
     |   case (vertexId, componentId) => println(s"Vertex $vertexId belongs to community $componentId"
)
     | }
Vertex 451 belongs to community 1
Vertex 454 belongs to community 1
Vertex 147 belongs to community 1
Vertex 155 belongs to community 1
Vertex 772 belongs to community 1
Vertex 752 belongs to community 1
Vertex 586 belongs to community 1
Vertex 667 belongs to community 1
Vertex 428 belongs to community 1
Vertex 464 belongs to community 1

scala> 
```

c) Connected Components

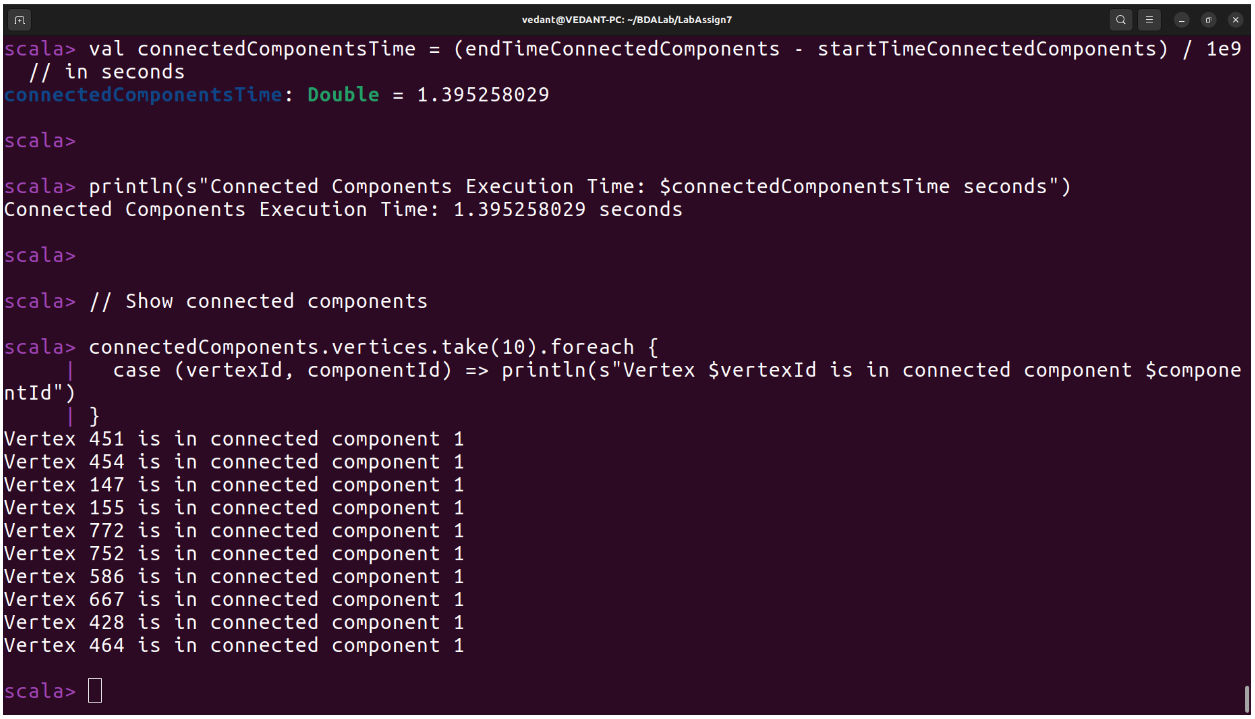
`connectedComponents` is used for identifying isolated groups (connected components) in the graph. This is already included in the community detection part, but you can also run it independently:

Code -

```
// Measure execution time for Connected Components
val startTimeConnectedComponents = System.nanoTime()

// Get connected components
val connectedComponents = graph.connectedComponents()

// Measure elapsed time
val endTimeConnectedComponents = System.nanoTime()
val connectedComponentsTime = (endTimeConnectedComponents -
startTimeConnectedComponents) / 1e9  // in seconds

println(s"Connected Components Execution Time: $connectedComponentsTime seconds")

// Show connected components
connectedComponents.vertices.take(10).foreach {
```

  case (vertexId, componentId) => println(s"Vertex $vertexId is in connected component $componentId")
}

Screenshot -



vedant@VEDANT-PC: ~/BDALab/LabAssign7

```
scala> val connectedComponentsTime = (endTimeConnectedComponents - startTimeConnectedComponents) / 1e9
  // in seconds
connectedComponentsTime: Double = 1.395258029

scala>

scala> println(s"Connected Components Execution Time: $connectedComponentsTime seconds")
Connected Components Execution Time: 1.395258029 seconds

scala>

scala> // Show connected components

scala> connectedComponents.vertices.take(10).foreach {
     |   case (vertexId, componentId) => println(s"Vertex $vertexId is in connected component $compone
ntId")
     | }
Vertex 451 is in connected component 1
Vertex 454 is in connected component 1
Vertex 147 is in connected component 1
Vertex 155 is in connected component 1
Vertex 772 is in connected component 1
Vertex 752 is in connected component 1
Vertex 586 is in connected component 1
Vertex 667 is in connected component 1
Vertex 428 is in connected component 1
Vertex 464 is in connected component 1

scala> 
```

d) Shortest Path

You can compute the shortest path using `GraphX`'s `shortestPaths` function. Here's an example for calculating the shortest path between two nodes:

Code -

import org.apache.spark.graphx.{Graph, VertexId}
import org.apache.spark.graphx.lib.ShortestPaths // Import the ShortestPaths library

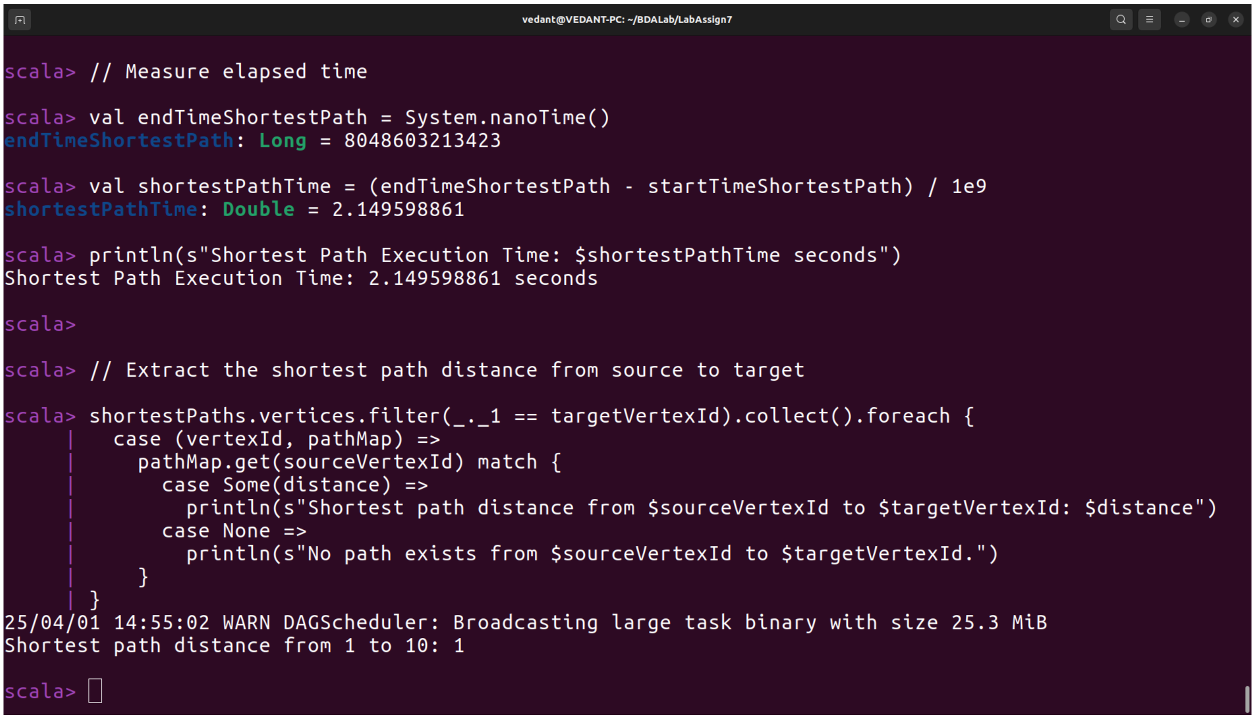
// Measure execution time for Shortest Path
val startTimeShortestPath = System.nanoTime()

// Specify source and target vertices
val sourceVertexId: VertexId = 1L
val targetVertexId: VertexId = 10L

// Compute shortest paths from the source vertex using ShortestPaths.run()

```
val shortestPaths = ShortestPaths.run(graph, Seq(sourceVertexId))

// Measure elapsed time
val endTimeShortestPath = System.nanoTime()
val shortestPathTime = (endTimeShortestPath - startTimeShortestPath) / 1e9
println(s"Shortest Path Execution Time: $shortestPathTime seconds")

// Extract the shortest path distance from source to target
shortestPaths.vertices.filter(_._1 == targetVertexId).collect().foreach {
  case (vertexId, pathMap) =>
            pathMap.get(sourceVertexId) match {
            case Some(distance) =>
            println(s"Shortest path distance from $sourceVertexId to $targetVertexId: $distance")
            case None =>
            println(s"No path exists from $sourceVertexId to $targetVertexId.")
            }
}
```

Screenshot -



```
scala> // Measure elapsed time

scala> val endTimeShortestPath = System.nanoTime()
endTimeShortestPath: Long = 8048603213423

scala> val shortestPathTime = (endTimeShortestPath - startTimeShortestPath) / 1e9
shortestPathTime: Double = 2.149598861

scala> println(s"Shortest Path Execution Time: $shortestPathTime seconds")
Shortest Path Execution Time: 2.149598861 seconds

scala>

scala> // Extract the shortest path distance from source to target

scala> shortestPaths.vertices.filter(_._1 == targetVertexId).collect().foreach {
     |   case (vertexId, pathMap) =>
     |     pathMap.get(sourceVertexId) match {
     |       case Some(distance) =>
     |         println(s"Shortest path distance from $sourceVertexId to $targetVertexId: $distance")
     |       case None =>
     |         println(s"No path exists from $sourceVertexId to $targetVertexId.")
     |     }
     | }
25/04/01 14:55:02 WARN DAGScheduler: Broadcasting large task binary with size 25.3 MiB
Shortest path distance from 1 to 10: 1

scala> 
```

Execution Time :

Operation	Execution Time
PageRank	27.105 sec
Community Detection	2.049 sec
Connected Components	1.395 sec
Shortest Path	2.149 sec