DBMS Term Project

LARGE SCALE GRAPH PROCESSING

**DBMS144**

Group Members

Banoth Karthik - 21CS10013 Chappa Jayanth - 21CS10017

Parala Siva Sai Yeswanth - 21CS30034

Kandula Revanth - 21CS10035

Nuthakki Varun - 21CS10045

# OBJECTIVE

* The aim of this project is to utilize a graph processing system like ApacheGraph, Pregel (GoldenOrb), Giraph, or Stanford GPS to handle large graphs within a database. It involves tasks such as importing a sizable graph dataset from the Stanford SNAP large graph repository, establishing an interface to execute basic graph queries, and optionally computing PageRank.
* The project focuses on optimizing performance metrics like query execution time, memory consumption, and disk usage to ensure efficient graph processing. The overarching objective is to facilitate the analysis of large graphs for extracting insights and knowledge from the data.
* The practical applications of processing large graphs in a database span various fields including social network analysis, web analytics, bioinformatics, and cybersecurity. For example, in social network analysis, analyzing large graphs can aid in pinpointing influential individuals, identifying communities or clusters based on shared interests or behaviors, and forecasting the propagation of information or diseases across the network.
* The Stanford Network Analysis Project (SNAP) houses a repository of extensive graph datasets openly available for research endeavors. These datasets encompass social networks, web graphs, communication networks, and diverse graph types.
* The SNAP repository serves as a valuable asset for researchers and developers seeking to assess and benchmark their graph processing systems using real-world datasets.
* To accomplish the task of processing large graphs within a database, the selection of a suitable graph processing system is crucial. Options such as ApacheGraph, Pregel (GoldenOrb), Giraph, or Stanford GPS should be considered based on specific project requirements and the characteristics of the graph dataset. Once a system is chosen, loading a large graph from the SNAP repository into the database can be facilitated using tools like SNAP.py or SNAP-Loader.
* Following the loading of the graph dataset, the implementation of an interface to execute basic graph queries becomes necessary. These queries may include tasks such as determining the shortest path between two nodes, identifying all nodes connected to a given node, or computing the degree centrality of a node. Moreover, advanced graph algorithms like PageRank, which measures the significance of each node in the graph, can be implemented.
* Performance profiling is essential to assess the efficiency of the system. Metrics such as query execution time, memory utilization, and disk usage can be monitored using profiling tools like JProfiler or YourKit. By analyzing performance metrics, potential bottlenecks can be identified, and optimizations can be explored through experimentation with different configurations and hardware setups.
* In summary, processing large graphs within a database offers myriad applications across various fields. Leveraging graph processing systems and large graph datasets, such as those found in the SNAP repository, enables the extraction of valuable insights and knowledge. Efficient query interfaces and performance optimization techniques play a crucial role in supporting decision-making and advancing research endeavors across diverse domains.

# METHODOLOGY

## Apache Spark Server:

* Apache Spark stands as a distributed computing framework tailored for handling extensive datasets. Its versatile interface facilitates distributed data processing, accommodating tasks ranging from batch and stream processing to machine learning and graph processing. Both industry and academia widely embrace Apache Spark for its utility in big data analytics, data science, and machine learning endeavors.
* A primary advantage of Apache Spark lies in its horizontal scalability across multiple machines, enabling it to manage substantial data volumes and execute computations in parallel. This scalability is achieved by partitioning data and processing each partition on distinct nodes within the cluster. Additionally, Apache Spark ensures fault tolerance, enabling recovery from node failures without data loss or disruption to ongoing computations.
* In the project's context aimed at processing large graphs within a database, Apache Spark, coupled with the GraphX library, proves instrumental. GraphX furnishes an API tailored for creating, loading, and manipulating graphs through distributed computing methodologies. Leveraging Apache Spark alongside GraphX facilitates efficient and scalable graph processing tasks on expansive graph datasets, encompassing operations like PageRank computation ,triangle counting, identification of connected comdddddponents.

## pagerank.scala:

The page.scala code uses the Apache Spark GraphX library to compute the PageRank algorithm on a graph dataset. The graph dataset is loaded from an edge list file using the GraphLoader.edgeListFile method, which creates a Graph object with vertex IDs of type Long, edge attributes of type Int, and vertex attributes of type Int.

The PageRank algorithm is then run on the graph using the PageRank.run method with the specified number of iterations (10 in this case). The resulting PageRank graph is a new graph with vertex attributes of type Double, representing the PageRank values of each vertex.

Finally, the top 10 vertices with the highest PageRank values are extracted from the PageRank graph using the vertices.top method, which returns an array of tuples sorted in descending order by the PageRank values. The results are then printed to the console using the println method.

This code snippet demonstrates a simple implementation of the PageRank algorithm on a graph dataset using the Apache Spark GraphX library. However, the performance and scalability of this implementation may depend on various factors such as the size and complexity of the graph dataset, the number of iterations, and the available hardware resources. Therefore, it is important to profile and optimize the implementation to achieve efficient and accurate results on large-scale graph datasets.

## profile\_performance.scala:

The p.scala code is used to profile the performance of loading a graph using Apache Spark GraphX. It starts by importing necessary libraries and initializing variables for profiling the performance.

The ThreadMXBean class is used to get information about the CPU utilization of the current thread. The System.nanoTime() function is used to get the start and end time of the graph loading process, which is used to calculate the elapsed time. The Files class is used to get information about the size of the graph file.

Next, the code loads the graph file using GraphLoader.edgeListFile() function and measures the end time. It then calculates the elapsed time and CPU utilization by subtracting the start time from the end time and calculating the CPU time used by the thread during this period. Finally, the code prints out the disk utilization, elapsed time, and CPU utilization of the graph loading process.

This code can be used to profile the performance of loading large-scale graph datasets using Apache Spark GraphX. By measuring the elapsed time and CPU utilization, it is possible to optimize the performance of the graph loading process by adjusting the cluster resources, partitioning the graph data, or using more efficient algorithms for graph processing.

## query\_processing.scala:

The quer.scala code uses the Apache Spark GraphX library to achieve our goal. The code performs the following steps:

Load the graph:- The code loads a graph from a file using the GraphLoader.edgeListFile method of the GraphX library. This method reads a file containing edges of the graph in the format "source\_id target\_id" and creates a graph with integer vertex and edge attributes.

Provide a menu of options:- The code provides a menu of options to the user, using the println statement to display the different query options that the user can choose from. The user is prompted to select an option by entering an integer between 1 and 7.

Perform graph queries based on user's choice: The code uses a switch-case statement to handle the different options selected by the user. Depending on the user's choice, the code performs one of the following graph queries:

* Option 1: The code counts the number of vertices in the graph using the graph.vertices.count() method.
* Option 2: The code counts the number of edges in the graph using the graph.edges.count() method.
* Option 3: The code finds the neighbors of a vertex, given its ID, by filtering the edges of the graph and selecting the destination vertices that match the given ID. The result is printed using the println statement.
* Option 4: The code finds the number of triangles in the graph using the graph.triangleCount() method. This method

returns a new graph with vertex attributes equal to the number of triangles each vertex belongs to, and the code sums up these values to get the total number of triangles.

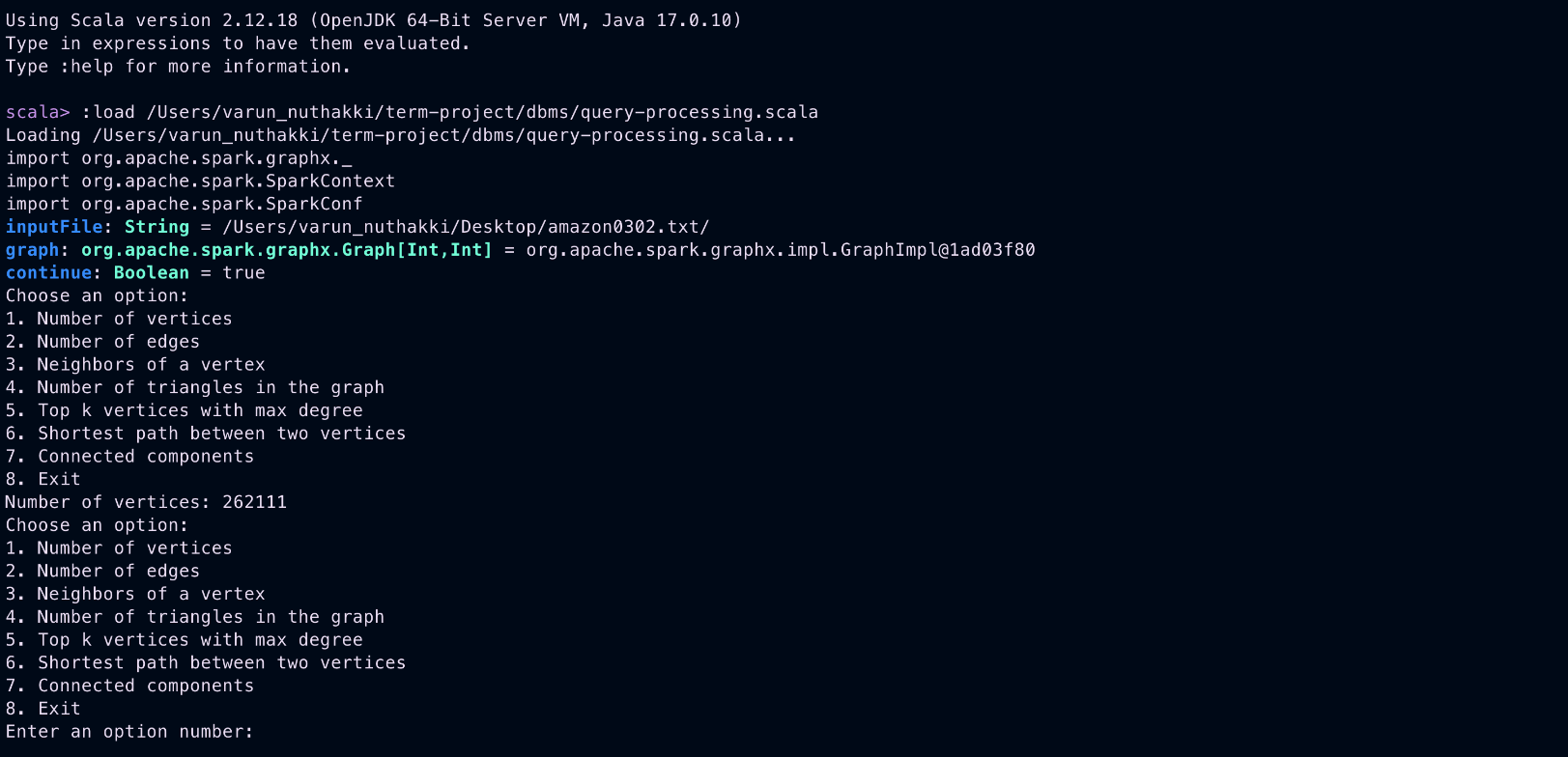
* Option 5: The code finds the top k vertices with the maximum degree using the graph.degrees.top(k) method. This method returns an array of pairs of vertex ID and degree, sorted in decreasing order of degree, and the code selects the first k elements of this array and prints them using the foreach method and the println statement.
* Option 6: The code finds the shortest path between two vertices, given their IDs, using the Pregel algorithm, which is a message-passing algorithm for computing shortest paths in graphs. The code first initializes the vertex attributes to positive infinity, except for the source vertex, which is initialized to 0.0. Then, it sends messages along the edges to update the distances, and iterates until convergence, using the graph.mapVertices and graph.pregel methods of the GraphX library. Finally, the code selects the shortest path and prints it using the println statement.
* Option 7: The code finds the connected components of the graph using the graph.connectedComponents() method. This method returns a new graph with vertex attributes equal to the IDs of the connected components, and the code prints the result using the foreach method and the println statement.

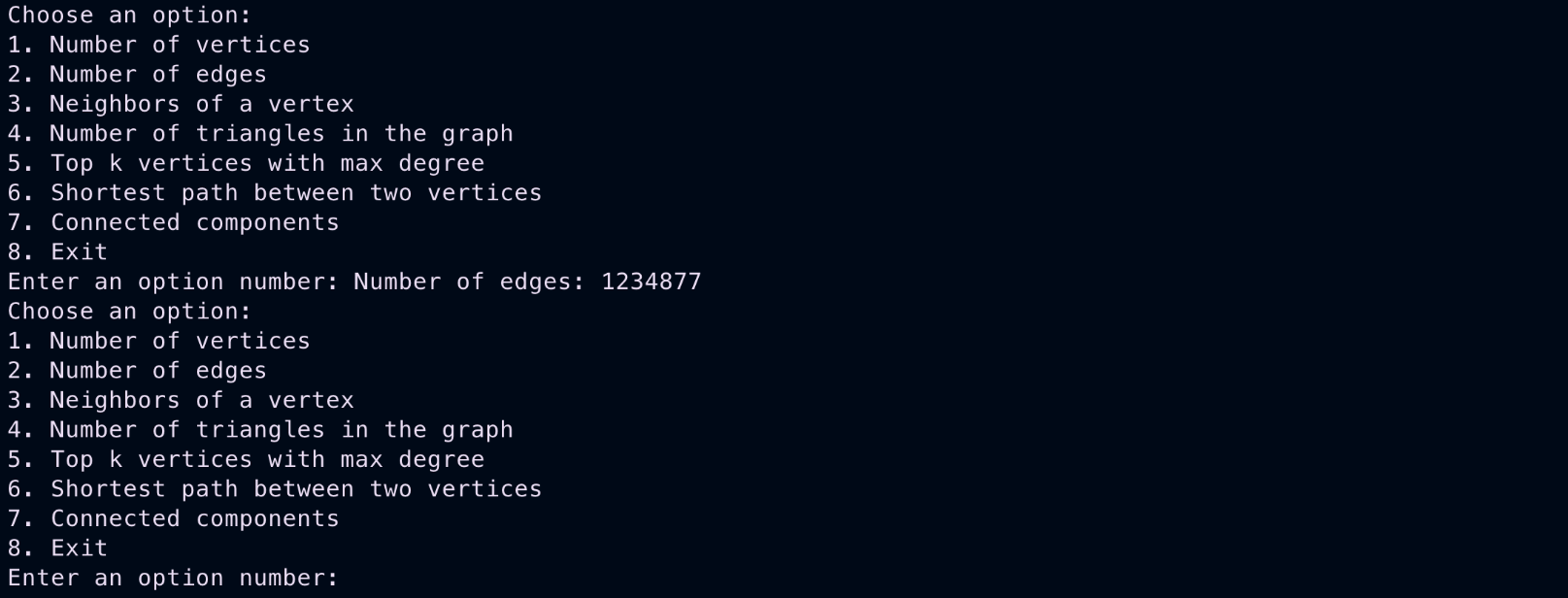
Profile Performance: The code does not explicitly profile the performance, but the use of the GraphX library and the efficient algorithms provided by the library, such as the Pregel algorithm, ensure that the graph queries are performed efficiently and can handle large graphs.

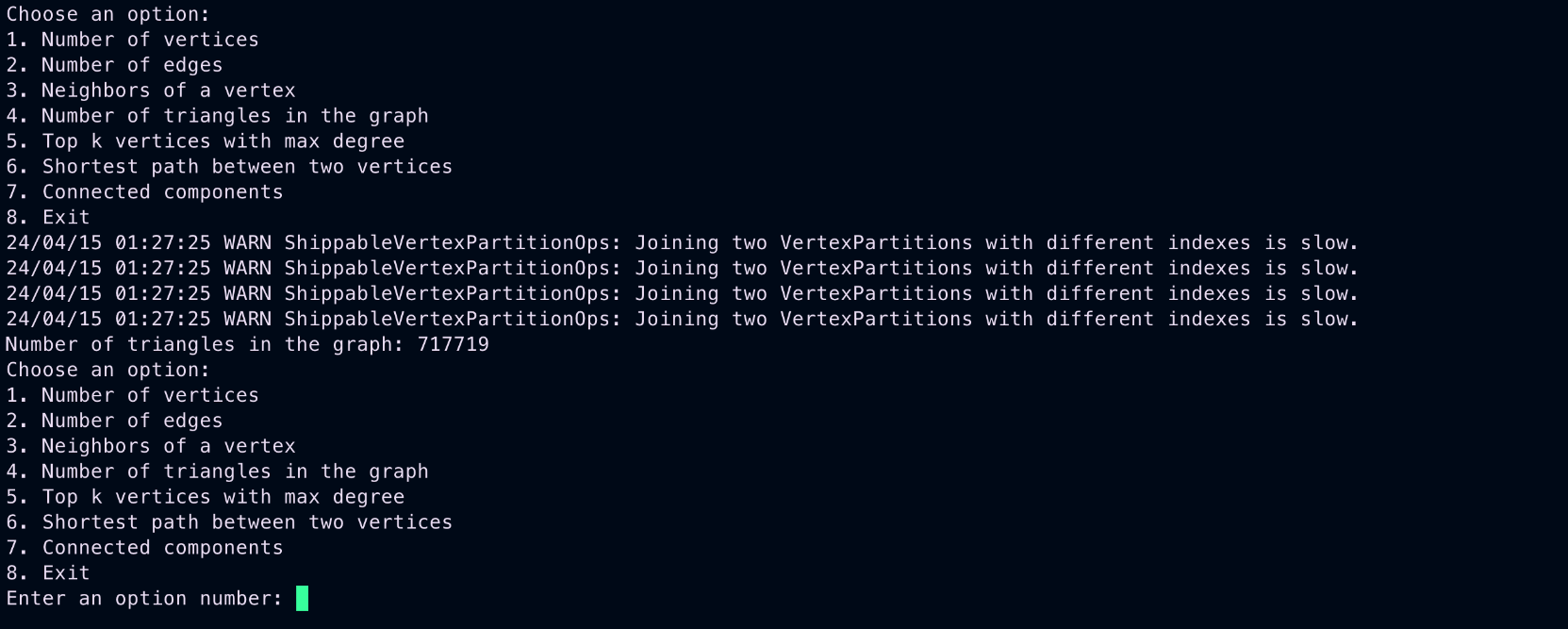
Overall, the code provides a simple and efficient interface for processing large graphs and performing various graph queries. The use of the GraphX library and efficient algorithms ensures that the code can handle large graphs and provides fast performance. However, the code only provides basic graph queries, and more advanced

# RESULTS/SCREENSHOTS

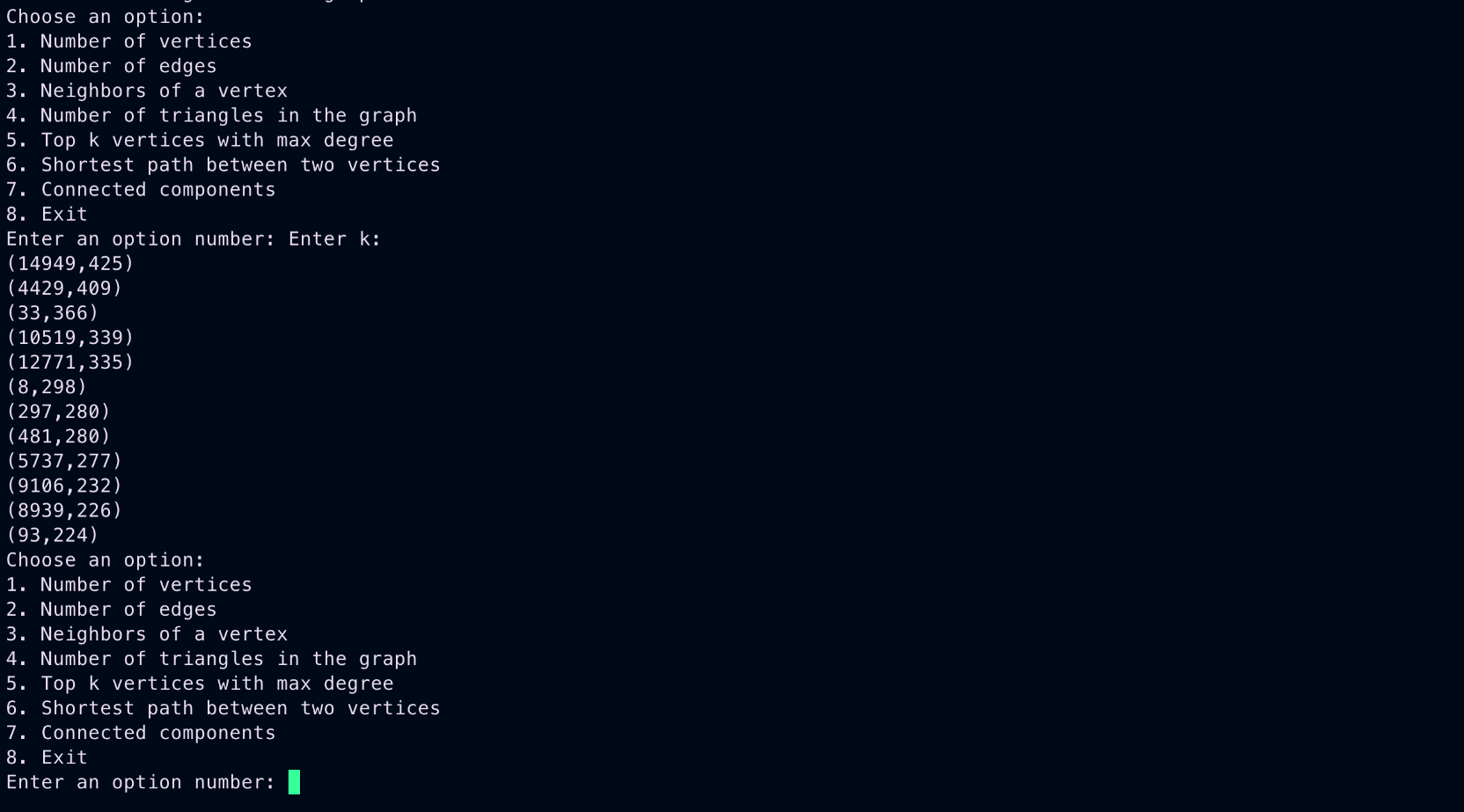
Query-1:

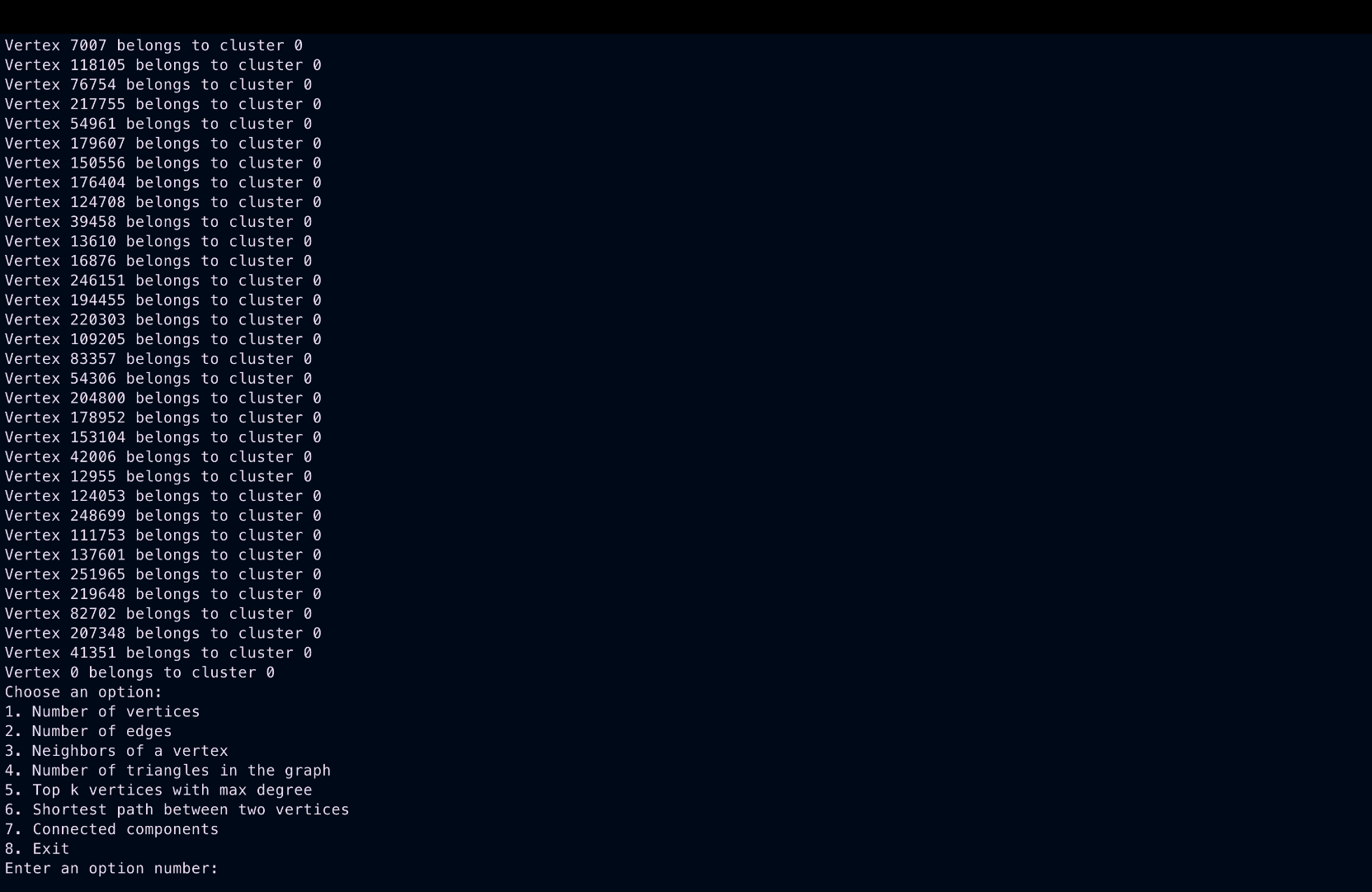


Query-2: 

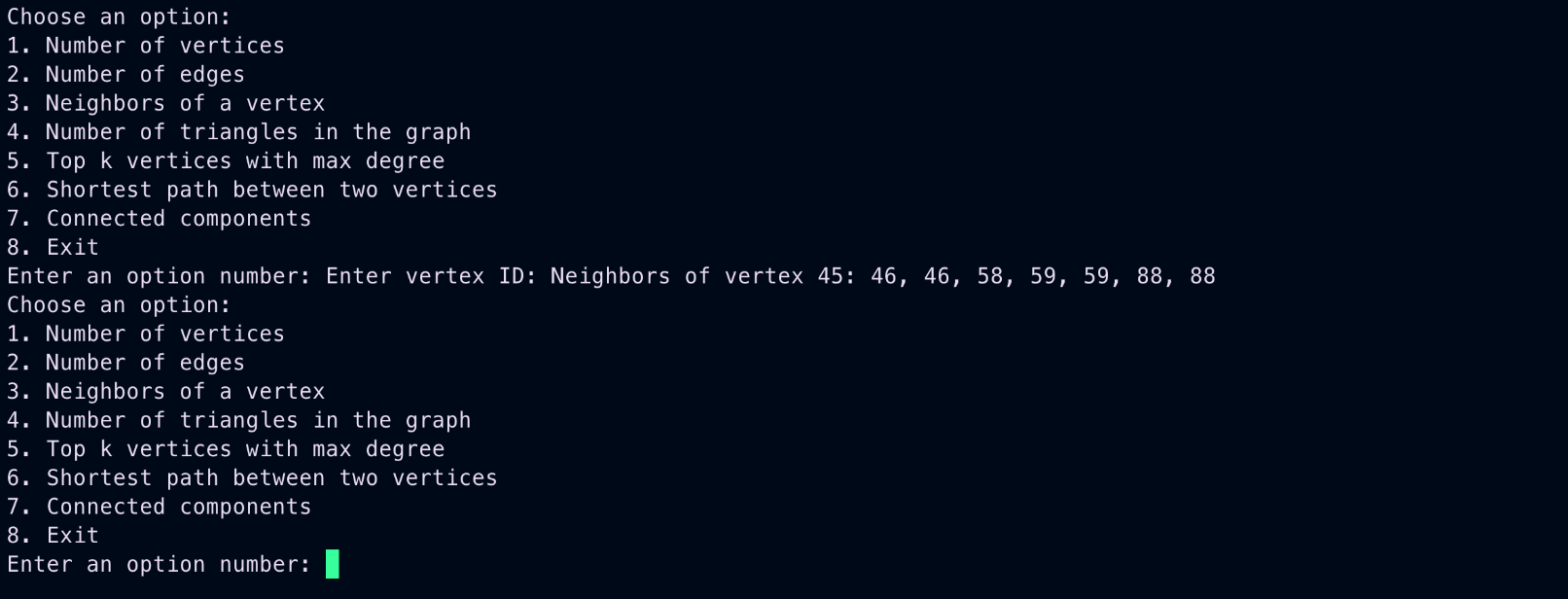
Query-3: 

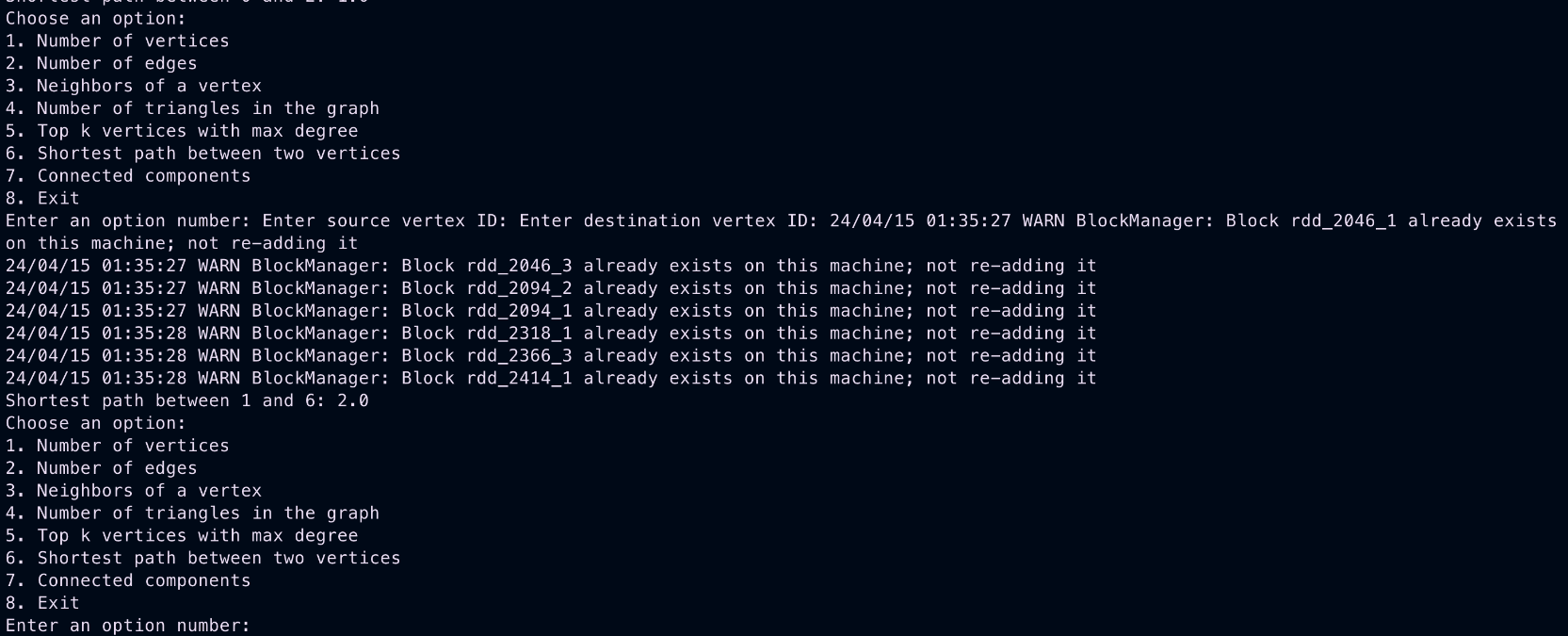
Query-4:

Query-5:

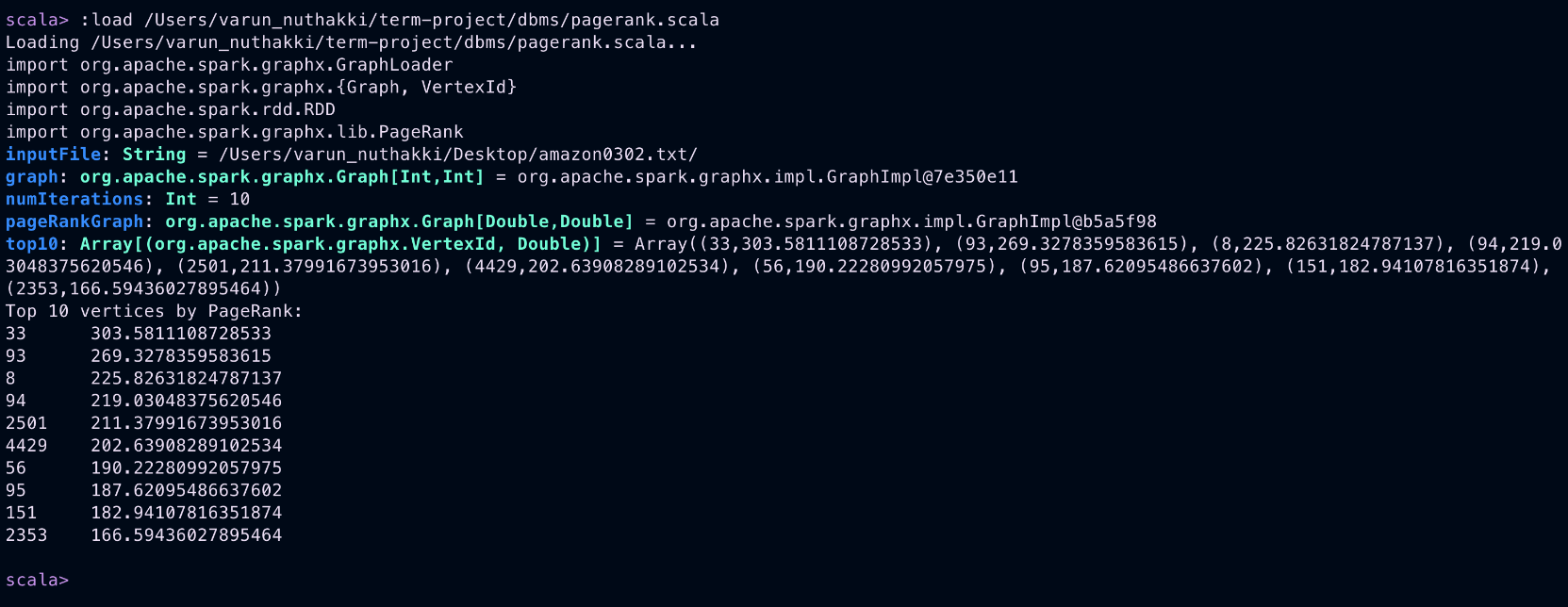


Query-6:

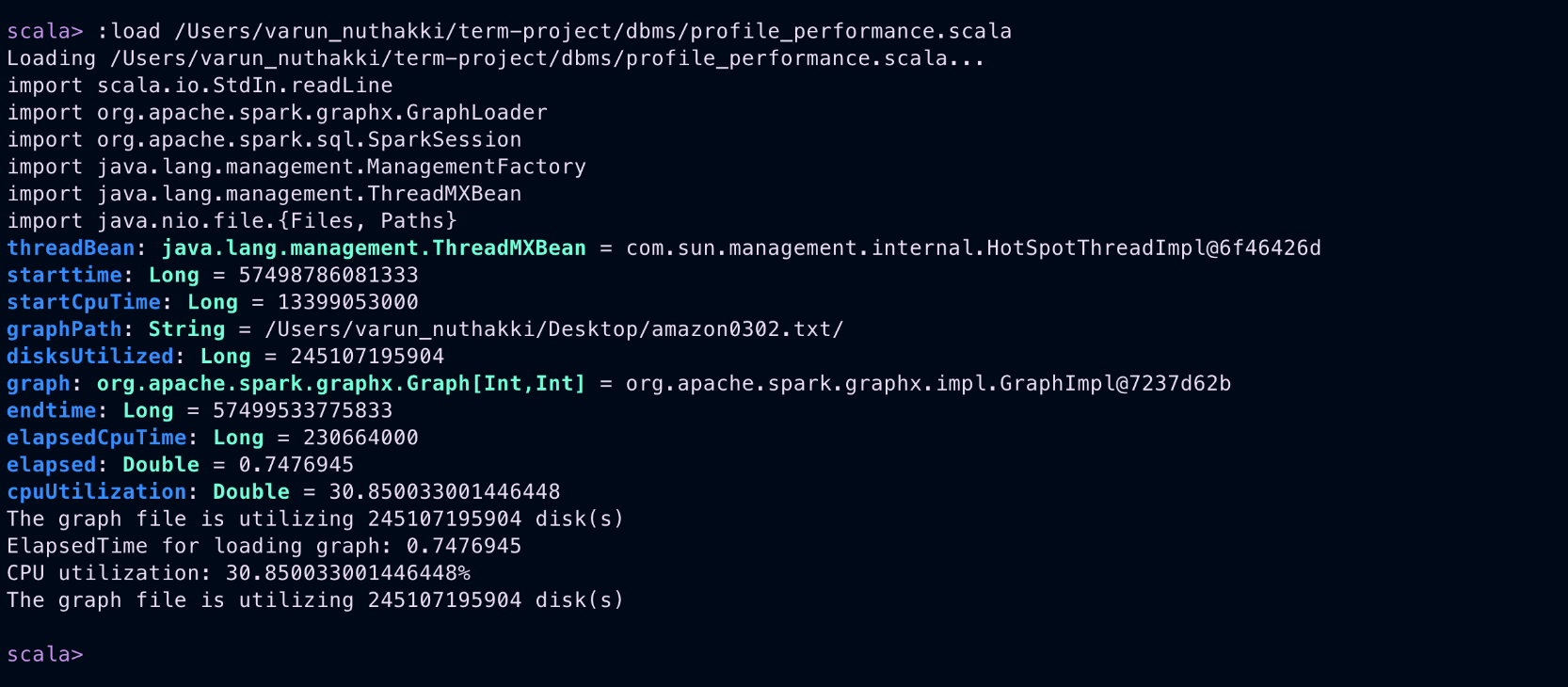
Query-7:

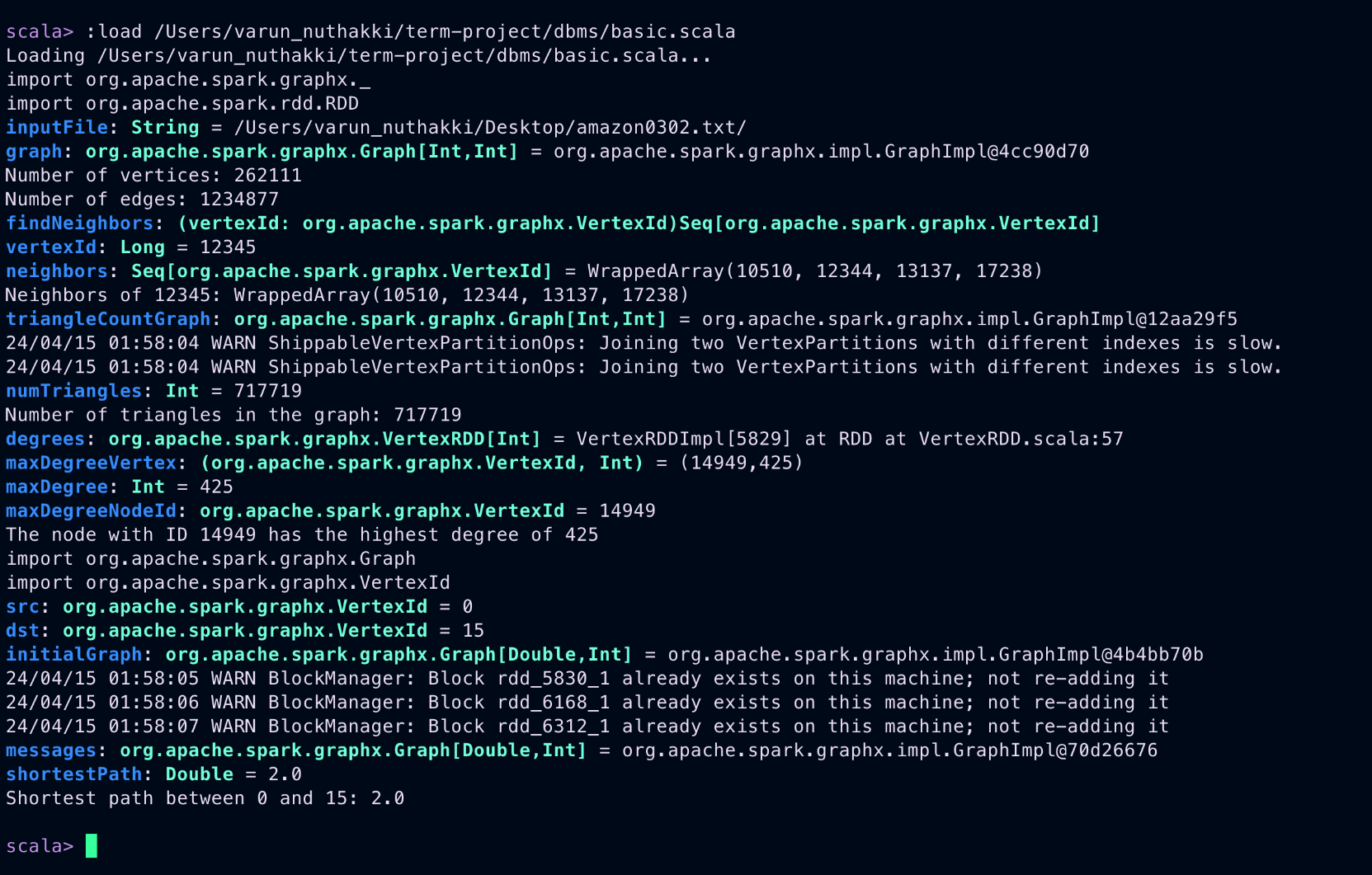


Page Rank:



Profile Performance:



basic.scala:

# REFERENCES

<https://phoenixnap.com/kb/install-spark-on-ubuntu>

https://github.com/gleahk/Large-Scale-Graph-processing-system