DBMS Term Project

LARGE SCALE GRAPH PROCESSING

DATABASE SQUASHERS

Group Members

Banoth Dinesh Karthik - 20CS30009
Pentamsetty Tharakeswar - 20CS10044
Saikat Moi - 20CS10050
Karamcheti Vamsy - 20CS30025
Dasari Giridhar Ram Chand - 20CS10022

OBJECTIVE

The objective of this project is to process large graphs in a database using a graph processing system such as ApacheGraph, Pregel (GoldenOrb), Giraph, or Stanford GPS. The project will involve loading a large graph dataset from the Stanford SNAP large graph repository, implementing an interface to run simple graph queries, and optionally computing PageRank. The performance of the system will be profiled and optimized to achieve efficient query execution time, low memory usage, and optimal disk usage. The ultimate goal of the project is to enable the processing and analysis of large graphs to derive insights and knowledge from the data.

The objective of processing large graphs in a database has several practical applications in various fields such as social network analysis, web analysis, bioinformatics, and cybersecurity. For instance, in social network analysis, processing and analyzing large graphs can help in identifying influential people in a network, detecting communities or clusters of people with similar interests or behaviors, and predicting the spread of information or disease through the network.

The Stanford Network Analysis Project (SNAP) is a repository of large graph datasets that are publicly available for research purposes. The datasets in SNAP include social networks, web graphs, communication networks, and other types of graphs. The SNAP repository provides a valuable resource for researchers and developers who want to test and benchmark their graph processing systems on real-world datasets.

To achieve the objective of processing large graphs in a database, we can choose a graph processing system such as ApacheGraph, Pregel (GoldenOrb), Giraph, or Stanford GPS, depending on our requirements and the nature of the graph dataset. Once we have chosen a system, we can load a large graph from the SNAP repository into our database, using tools such as SNAP.py or SNAP-Loader.

After loading the graph dataset, we can implement an interface to run simple graph queries, such as finding the shortest path between two nodes, finding all nodes connected to a given node, or finding the degree centrality of a node. Additionally, we can implement more advanced graph algorithms such as PageRank, which is a measure of the importance of each node in the graph.

To profile performance, we can measure various metrics such as query execution time, memory usage, and disk usage. We can use profiling tools such as JProfiler or YourKit to measure the performance of our system and identify bottlenecks. We can also experiment with different configurations and hardware setups to optimize performance.

Overall, the objective of processing large graphs in a database has numerous applications in various fields and can be achieved using graph processing systems and large graph datasets such as those available in the SNAP repository. By implementing efficient query interfaces and optimizing performance, we can extract valuable insights and knowledge from large graphs to support decision-making and advance research in various domains.

METHODOLOGY

Apache Spark Server:

Apache Spark is a distributed computing framework that is designed for processing large-scale data sets. It provides an interface for distributed data processing that can be used to perform various operations such as batch processing, stream processing, machine learning, and graph processing. Apache Spark is widely used in industry and academia for big data analytics, data science, and machine learning applications.

One of the key benefits of Apache Spark is its ability to scale horizontally across multiple machines, allowing it to handle large volumes of data and process computations in parallel. It achieves this by dividing the data into partitions and processing each partition on a separate node in the cluster. Apache Spark also provides fault tolerance, meaning that it can recover from node failures without losing data or interrupting computations.

In the context of the project to process large graphs in a database, Apache Spark can be used with the GraphX library to load and process large-scale graph datasets. The GraphX library provides an API for creating, loading, and processing graphs using distributed computing techniques. By using Apache Spark with GraphX, it is possible to perform efficient and scalable graph processing operations such as PageRank, triangle counting, and connected components on large graph datasets.

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:

```
scala> :load /home/dinesh/Downloads/code/quer.scala
Loading /home/dinesh/Downloads/code/quer.scala...
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
inputFile: String = /home/dinesh/spark/facebook_combined.txt
graph: org.apache.spark.graphx.Graph[Int,Int] = org.apache.spark.graphx.impl.GraphImpl@7f6f4644
continue: Boolean = true

Choose an option:
1. Number of vertices
2. Number of edges
3. Neighbors of a vertex
4. Number of triangles in the graph
5. Top k vertices with max degree
6. Shortest path between two vertices
7. Connected components
8. Exit
Number of vertices: 4039
```

Query-2:

```
Choose an option:
1. Number of vertices
2. Number of edges
3. Neighbors of a vertex
4. Number of triangles in the graph
5. Top k vertices with max degree
6. Shortest path between two vertices
7. Connected components
8. Exit
Enter an option number: Number of edges: 88234
```

Query-3:

```
Choose an option:
1. Number of vertices
2. Number of edges
3. Neighbors of a vertex
4. Number of triangles in the graph
5. Top k vertices with max degree
6. Shortest path between two vertices
7. Connected components
8. Exit
23/04/16 23:00:15 WARN ShippableVertexPartitionOps: Joining two VertexPartitions with different indexes is slow.
23/04/16 23:00:15 WARN ShippableVertexPartitionOps: Joining two VertexPartitions with different indexes is slow.
Number of triangles in the graph: 1612010
```

Query-4:

```
Choose an option:
1. Number of vertices
2. Number of edges
3. Neighbors of a vertex
4. Number of triangles in the graph
5. Top k vertices with max degree
6. Shortest path between two vertices
7. Connected components
8. Exit
Enter an option number: enter the value of k
(107,1045)
(1684,792)
(1912,755)
(3437,547)
(0,347)
(2543,294)
(2347,291)
(1888,254)
(1800,245)
(1663,235)
(1352,234)
(2266,234)
(483,231)
(348,229)
```

Query-5:

```
Choose an option:
1. Number of vertices
2. Number of edges
3. Neighbors of a vertex
4. Number of triangles in the graph
5. Top k vertices with max degree
6. Shortest path between two vertices
7. Connected components
8. Exit
Enter an option number: Vertex 3672 belongs to cluster 0
Vertex 3331 belongs to cluster 0
Vertex 2990 belongs to cluster 0
Vertex 3744 belongs to cluster 0
Vertex 3026 belongs to cluster 0
Vertex 2685 belongs to cluster 0
Vertex 3340 belongs to cluster 0
Vertex 2344 belongs to cluster 0
Vertex 3753 belongs to cluster 0
Vertex 2353 belongs to cluster 0
Vertex 2012 belongs to cluster 0
Vertex 137 belongs to cluster 0
Vertex 3107 belongs to cluster 0
Vertex 2021 belongs to cluster 0
Vertex 891 belongs to cluster 0
Vertex 1205 belongs to cluster 0
Vertex 550 belongs to cluster 0
Vertex 864 belongs to cluster 0
Vertex 146 belongs to cluster 0
Vertex 2775 belongs to cluster 0
Vertex 559 belongs to cluster 0
Vertex 3959 belongs to cluster 0
Vertex 218 belongs to cluster 0
Vertex 3995 belongs to cluster 0
Vertex 3654 belongs to cluster 0
Vertex 568 belongs to cluster 0
Vertex 3968 belongs to cluster 0
Vertex 3627 belongs to cluster 0
Vertex 227 belongs to cluster 0
Vertex 1752 belongs to cluster 0
Vertex 3322 belongs to cluster 0
Vertex 2981 belongs to cluster 0
Vertex 2640 belongs to cluster 0
Vertex 765 belongs to cluster 0
```

Query-6:

```
Choose an option:
1. Number of vertices
2. Number of edges
3. Neighbors of a vertex
4. Number of triangles in the graph
5. Top k vertices with max degree
6. Shortest path between two vertices
7. Connected components
8. Exit
Enter an option number: Neighbors of 44: [J@38b61eb3
```

Query-7:

```
continue: Boolean = true
Choose an option:
1. Number of vertices
2. Number of edges
3. Neighbors of a vertex
4. Number of triangles in the graph
5. Top k vertices with max degree
6. Shortest path between two vertices
7. Connected components
8. Exit
Enter an option number: Shortest path between 0 and 15: 1.0
Choose an option:
```

Page Rank:

```
import org.apache.spark.graphx.GraphLoader
import org.apache.spark.graphx.{Graph, VertexId}
import org.apache.spark.rdd.RDD
import org.apache.spark.graphx.lib.PageRank
inputFile: String = /home/dinesh/spark/facebook_combined.txt
graph: org.apache.spark.graphx.Graph[Int,Int] = org.apache.spark.graphx.impl.GraphImpl@8abb4a5
numIterations: Int = 10
pageRankGraph: org.apache.spark.graphx.Graph[Double,Double] = org.apache.spark.graphx.impl.GraphImpl@167394fd
top10: Array[(org.apache.spark.graphx.VertexId, Double)] = Array((1911,40.173020035189126), (3434,38.21196968977279), (2655,
,25.51131091705774), (1907,21.240971021533998), (3971,20.558757839388395), (2654,20.200635873097152), (1910,17.5513294849114
Top 10 vertices by PageRank:
1911    40.173020035189126
3434    38.21196968977279
2655    37.63024755334742
1902    37.243511110381164
1888    28.028255649535613
2649    25.51131091705774
1907    21.240971021533998
3971    20.558757839388395
2654    20.200635873097152
1910    17.551329484911424
```

Profile Performance:

```
import scala.io.StdIn.readLine
import org.apache.spark.graphx.GraphLoader
import org.apache.spark.sql.SparkSession
import java.lang.management.ManagementFactory
import java.lang.management.ThreadMXBean
import java.nio.file.{Files, Paths}
threadBean: java.lang.management.ThreadMXBean = com.sun.management.internal.HotSpotThreadImpl@71cf1b07
starttime: Long = 17687772306623
startCpuTime: Long = 13485538252
graphPath: String = /home/dinesh/spark/facebook_combined.txt
disksUtilized: Long = 94457470976
graph: org.apache.spark.graphx.Graph[Int,Int] = org.apache.spark.graphx.impl.GraphImpl@16cf13d0
endtime: Long = 17690311464419
elapsedCpuTime: Long = 1645612213
elapsed: Double = 2.539157796
cpuUtilization: Double = 64.80937165828665
The graph file is utilizing 94457470976 disk(s)
ElapsedTime for loading graph: 2.539157796
CPU utilization: 64.80937165828665%
The graph file is utilizing 94457470976 disk(s)
```

REFERENCES

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

Github link :-

https://github.com/THARAKESWAR-P/Large-scale-graph-processing.git