Project Statement for Milestone 3

Big Lens Massive Solutions

Antonio Barber, Adam Shtrikman, Sher Chhi Ly

Overview:

At this stage of the project, teams should have implemented and tested their algorithms on the prepared data set. The algorithms should address the problem statement and project goals. The algorithms should leverage NoSQL database and Hadoop / Spark data processing framework, and should scale with data. Some algorithms should be database queries or Hadoop/Spark function calls, while others should be built from the ground up.

In this report, the teams are expected to report on their algorithms and results.

Report Topics:

The report should cover the following subtopics and answer the questions listed:

Data Files:

A. In addition to the NoSQL database, are you using any. non-CSV data files (eg. Parquet files, HDFS files) for data storage? If so, describe the data file and data transformation steps.

Our primary dataset, in RDF format and structured as triples (subject-predicate-object), is ideally suited for graph databases like Neo4j, enabling direct ingestion of RDF triples. The dataset, stored in Turtle format (`.ttl`), is compatible with Neo4j via the `n10s` plugin, which facilitates efficient querying and data processing. It includes multi-lingual labels, comments, and entity attributes, with nodes representing entities and edges capturing relationships. This structure allows Neo4j to maintain the graph model natively without additional transformation, effectively supporting complex data relationships.

B. Include sample data, if applicable, in the non-CSV file format.

```
yago:Candyman_3_u003A__Day_of_the_Dead rdfs:label "Candyman 3: Day of the
Dead"@en .
yago:Candyman_3_u003A__Day_of_the_Dead rdfs:comment "1999 film directed by
Turi Meyer"@en .
yago:Mentuhotep__u0028_queen_u0029_ schema:birthPlace yago:Ancient_Egypt .
yago:Angelo_Parisi  rdfs:label      "安杰洛·帕里西"@zh       .
```

Algorithm Description:

A. Give a formal description of each of the algorithms you have implemented. It should consists of (1) input; (2) output, and (3) (computing) operations.
B. Provide pseudo-code of the algorithms.
C. Discuss any optimization techniques you have implemented.

Neighbor Counting Algorithm

Input: Node URI.
Output: Integer count of the neighbors for the given node.
Operations: Uses a simple Cypher query to match nodes connected to the given node and counts the edges.

Pseudo-Code:

```
Function get_neighbors_count(node_uri):
      Query = "MATCH (n:Resource {uri: $node_uri})-[]-(neighbor:Resource)
      RETURN count(neighbor) AS neighborCount"
      Execute Query with node_uri
      Return neighbor count
```

Optimization: This algorithm leverages Neo4j's native indexing on nodes to optimize neighbor retrieval.

Direct Neighbor Check Algorithm

Input: Two node URIs.
Output: Boolean indicating if nodes are direct neighbors.
Operations: Matches nodes by their URIs and checks if an edge exists directly between them.

Pseudo-Code:

```
Function are_neighbors(uri1, uri2):
      Query = "MATCH (n1:Resource {uri: $uri1})-[]-(n2:Resource {uri: $uri2})
      RETURN count(n2) > 0 AS areNeighbors"
      Execute Query with uri1 and uri2
      Return whether nodes are direct neighbors
```

Optimization: Uses efficient pattern matching to minimize data traversal, filtering only relevant relationships.

Common Neighbor Check Algorithm

Input: Two node URIs.
Output: Boolean indicating if nodes share a common neighbor.
Operations: Checks if a common neighbor exists for two nodes by using a two-hop relationship query.

Pseudo-Code:

```
Function has_common_neighbors(uri1, uri2):
      Query = "MATCH (n1:Resource {uri: $uri1})-[]-(neighbor:Resource)-[]-
      (n2:Resource {uri: $uri2}) RETURN count(neighbor) > 0 AS
      hasCommonNeighbors"
      Execute Query with uri1 and uri2
      Return if common neighbors exist
```

Optimization: Limits traversal depth to two hops to prevent performance degradation.

Built-in Shortest Path Algorithm in Neo4j

Input: Start and end node URIs.
Output: Path length and list of node URIs in the path.
Operations: Uses Neo4j's `shortestPath` function to find the shortest path between two nodes and extracts the path details.

Pseudo-Code:

```
Function detailed_shortest_path(start_uri, end_uri):
```

```
Query = "MATCH p = shortestPath((start:Resource {uri: $start_uri})-[*]-
(end:Resource {uri: $end_uri})) RETURN p, length(p) AS pathLength, [node
IN nodes(p) | node.uri] AS pathNodes"
Execute Query with start_uri and end_uri
Return path length and path nodes
```

Optimization: The Neo4j's built-in shortest path function is optimized for performance, returning only the shortest path and limiting unnecessary traversals.

Algorithm Results:

A. Provide algorithm results (output) using sample data (inputs). Also, include the performance metrics (execution time, accuracy, etc.) for the sample data.

B. Describe whether you are storing the results of the algorithms in a database and/or data files. Describe how you plan to present the results to the user. Perhaps you plan to run the algorithm on-demand and present it to the user, or you plan to execute the algorithms off-line, store their results, and present the result on-demand.

Algorithm Outputs:

- **Neighbor Count**: Returns the total number of neighbors for a specified node, providing insight into the connectivity of individual nodes.
- **Direct Neighbor Check**: Determines if two nodes are directly connected, useful for assessing direct relationships.
- **Common Neighbor Check**: Identifies whether two nodes share any neighbors, aiding in understanding indirect associations.
- **Detailed Shortest Path**: Provides the shortest path between two nodes, along with its length, enabling efficient path analysis in the graph.

Performance Metrics:

- Execution time for each algorithm varies with data size, but all algorithms were optimized to run efficiently.
- Testing on a reduced dataset of ~1 million nodes returned results within milliseconds for neighbor checks and seconds for shortest path queries.

Data Storage:

- To support scalability and efficient resource management, results are queried dynamically on-demand without storing intermediate data in the database. This approach not only reduces storage overhead but also allows results to be presented immediately to the user, minimizing latency. The algorithms can be run as needed, making it feasible to scale up data volume without compromising performance.

Algorithm Scalability:

A. Have you implemented the algorithm with scalability in mind? If so, describe how your algorithm will scale with data.

Scalability Considerations:

- **Neighbor Count and Direct Neighbor Check**: These operations scale efficiently with data volume due to the use of indexed nodes, allowing rapid access even as the dataset grows..
- **Shortest Path Algorithm**: Neo4j's native shortest path function is optimized for large datasets and performs well within available resources, though system memory could become a limiting factor on very large graphs.
- **Future Improvements**: To enhance performance for massive datasets, implementing caching for frequently queried paths and introducing batch querying for multiple paths are promising strategies. These approaches would reduce redundant calculations and improve query efficiency.

Source Code:

A. Provide source code of your algorithms in a Zip file.

Zip File will be included. As a placeholder, here is the code for all algorithms.

```
from neo4j import GraphDatabase

class ShortestPathTester:
    def __init__(self, uri, user, password):
        # Initialize connection to Neo4j database
        self.driver = GraphDatabase.driver(uri, auth=(user, password))

    def close(self):
        # Close the Neo4j database connection
        if self.driver:
            self.driver.close()

    def get_neighbors_count(self, node_uri):
        # Get the count of neighbors for a specific node URI
        with self.driver.session() as session:
            result = session.run(
                """
                MATCH (n:Resource {uri: $node_uri})-[]-(neighbor:Resource)
                RETURN count(neighbor) AS neighborCount
                """,
                node_uri=node_uri
            )
            # Extract the neighbor count from the query result
            neighbor_count = result.single()["neighborCount"]
            print(f"Neighbors of '{node_uri}': {neighbor_count} neighbors")
            return neighbor_count
```

```
    def are_neighbors(self, uri1, uri2):
        # Check if two nodes are direct neighbors
        with self.driver.session() as session:
            result = session.run(
                """
                MATCH (n1:Resource {uri: $uri1})-[]-(n2:Resource {uri: $uri2})
                RETURN count(n2) > 0 AS areNeighbors
                """,
                uri1=uri1,
                uri2=uri2
            )
            # Return whether nodes are direct neighbors
            return result.single()["areNeighbors"]

    def has_common_neighbors(self, uri1, uri2):
        # Check if two nodes have any common neighbors
        with self.driver.session() as session:
            result = session.run(
                """
                MATCH (n1:Resource {uri: $uri1})-[]-(neighbor:Resource)-[]-
(n2:Resource {uri: $uri2})
                RETURN count(neighbor) > 0 AS hasCommonNeighbors
                """,
                uri1=uri1,
                uri2=uri2
            )
            # Return whether nodes have common neighbors
            return result.single()["hasCommonNeighbors"]

    def detailed_shortest_path(self, start_uri, end_uri):
        # Find the shortest path between two nodes using Neo4j's built-in
function
        with self.driver.session() as session:
            result = session.run(
                """
                MATCH p = shortestPath(
                    (start:Resource {uri: $start_uri})-[*]-(end:Resource {uri:
$end_uri})
                )
                RETURN p, length(p) AS pathLength, [node IN nodes(p) | node.uri]
AS pathNodes
                """,
                start_uri=start_uri,
                end_uri=end_uri
```

```
            )

            # Extract path details and print them if a path is found
            path_record = result.single()
            if path_record:
                path_length = path_record["pathLength"]
                path_nodes = path_record["pathNodes"]

                print("\nBuilt-in shortest path:")
                print("Shortest path details:")
                print(f"Path length: {path_length}")
                print("Path nodes (URIs):")
                for node_uri in path_nodes:
                    print(node_uri)
            else:
                print(f"No path found between '{start_uri}' and '{end_uri}'.")

# Usage example
if __name__ == "__main__":
    # Neo4j connection details
    uri = "bolt://localhost:7687"
    user = "neo4j"
    password = "12345678"

    # Initialize the ShortestPathTester instance
    tester = ShortestPathTester(uri, user, password)

    try:
        # Define start and end URIs for testing
        start_uri = "http://yago-knowledge.org/resource/State_of_Bahrain"
        end_uri = "http://yago-knowledge.org/resource/Order_of_King_Abdulaziz"

        # Get and print neighbor counts for both URIs
        print("Checking neighbor counts...")
        tester.get_neighbors_count(start_uri)
        tester.get_neighbors_count(end_uri)

        # Check if URIs are direct neighbors
        if tester.are_neighbors(start_uri, end_uri):
            print(f"\nThe nodes '{start_uri}' and '{end_uri}' are direct
neighbors. Path is shorter than 2.")
        else:
            # Check if URIs have common neighbors
            if tester.has_common_neighbors(start_uri, end_uri):
```

```
                print(f"\nThe nodes '{start_uri}' and '{end_uri}' have a common
neighbor. Path is shorter than 2.")
            else:
                print(f"\nThe nodes '{start_uri}' and '{end_uri}' are not within
2 connections.")

        # Run and display detailed shortest path information
        tester.detailed_shortest_path(start_uri, end_uri)

    finally:
        # Close the Neo4j connection
        tester.close()
```

Peer Evaluation:

Each team member should complete the CATME Peer Evaluation survey.

Grading:

- 50 pts: Team has successfully implemented algorithms with good results. Team is taking a good approach to storing, if applicable, and presenting the results of the algorithms to the user. The project milestone document provides all information with relevant description, pseudo code / code snippets and diagrams / images.