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.

Subgraph Matching Algorithm

Input: Node URI.
Output: List of matching subgraphs, each element specifying the connections between nodes and their relations.
Operations: Utilizes Neo4j query by collecting and returning matched subgraphs containing node URIs, related node URIs, and their relational type. The objective of this algorithm is to return downstream subgraphs to the given node.

Pseudo-Code:

```
Function match_subgraph(pattern):
    Connect to Neo4j database
    Query = "Subgraph pattern matching query for specified relationships"
    Aggregate the results as subgraph matches
    Return matching subgraphs
```

Optimization: Minimizes data traversal by filtering nodes and relationships based on defined subgraph patterns, reducing database load.

Node Similarity Calculation Algorithm

Input: Two node URIs.
Output: Cosine similarity score between nodes based on shared connections.
Operations: Retrieves neighbors of each node by executing Neo4j query, constructs feature vectors via matrices representing the presence or absence of each neighbor for both nodes, computes the cosine similarity between the two feature vectors.

Pseudo-Code:

```
Function calc_similarity(uri1, uri2):
    Connect to Neo4j database
    Query1 = "Retrieve neighbors of node with uri1"
    Query2 = "Retrieve neighbors of node with uri2"
    Create feature vectors from the neighbors
    Calculate the cosine similarity between the vectors
    Return similarity score
```

Optimization: Utilizes cosine similarity to measure the degree of relation between vectors efficiently by leveraging vectorized operation computations for increased performance.

Find Similar Entities Algorithm

Input: `entity_uri`: The URI of the reference entity. `limit`: Max number of results (default = 10).
Output: List of similar entities containing:

- URI: URI of the similar entity.
- Label: Label or name of the similar entity.

Operations: The algorithm matches entities that share the same award relationship with the reference entity. A Cypher query fetches similar entities and their labels while excluding the reference entity itself.

Pseudo-Code:

```
Function find_similar_entities(entity_uri, limit=10):
    Query = """
        MATCH (e:Resource {uri: $entity_uri})-[:sch__award]->(award)<-
[:sch__award]-(similar:Resource)
        WHERE e <> similar
        RETURN similar.uri AS uri, similar.rdfs__label AS label
        LIMIT $limit
    """
    Execute Query with entity_uri and limit
    Extract URI and Label for each result
    Return list of similar entities
```

Optimization: Leverages Neo4j's indexing for efficient node and relationship retrieval. Limits results with a `LIMIT` clause to optimize performance for large datasets.

Relationship-Specific Search Algorithm

Input: start_uri: The URI of the starting entity. relationship_type: Type of relationship to explore. depth: Maximum traversal depth (default = 2).

Output: List of connected entities containing:

- End URI: URI of the connected entity.
- Traversal Depth: Number of hops to reach the entity.

Operations: Traverses the graph using the specified relationship type up to the defined depth. A Cypher query matches paths and retrieves connected entities.

Pseudo-Code:

```
Function relationship_specific_search(start_uri, relationship_type,
depth=2):
    Query = """
        MATCH path = (start:Resource {uri: $start_uri})-
[:{relationship_type}*1..{depth}]-(end)
        RETURN DISTINCT end.uri AS uri, length(path) AS depth
    """
    Execute Query with start_uri and relationship_type
    Extract URI and Depth for each result
    Return list of connected entities
```

Optimization: Restricts traversal depth to reduce computational load. Uses indexed properties for rapid entity matching.

Explore Multi-Hop Connections Algorithm

Input: `start_uri`: The URI of the starting entity. `max_hops`: Maximum number of hops to explore (default = 3). `limit`: Maximum number of results to return (default = 50).

Output: List of multi-hop connections containing:

- Connected Entity: URI of the connected entity.
- Hops: Number of hops to reach the entity.

Operations: Explores connections to the starting entity up to the specified number of hops. A Cypher query retrieves paths and connected entities.

Pseudo-Code:

```
Function explore_multi_hop(start_uri, max_hops=3, limit=50):
        Query = """
            MATCH path = (start:Resource {uri: $start_uri})-
    [*1..{max_hops}]-(end)
            RETURN DISTINCT end.uri AS uri, length(path) AS depth
            LIMIT $limit
        """
        Execute Query with start_uri, max_hops, and limit
        Extract Connected Entity and Hops for each result
        Return list of multi-hop connections
```
Optimization: Limits the number of results and traversal depth to manage computational resources. Uses Neo4j's optimized graph traversal algorithms.

Search Subgraph Algorithm
Input: Keyword: A string used to search the graph. A limit (int, default=10).
Output: List of subgraph results containing entity labels, URIs, relationships, and related entity details.
Operations: Queries the database for entities and their relationships matching the keyword.

Pseudo-code:
```
Function search_subgraph(keyword, limit=10):
    Query = """
        MATCH (entity)-[relation]-(relatedEntity)
        WHERE entity.rdfs__label CONTAINS $keyword
        RETURN entity, relation, relatedEntity
        LIMIT $limit
    """
    Execute Query with parameters keyword and limit.
    For each record in the result:
        Extract entity label, URI, relation type, and related entity details.
        Format and append the details into a result list.
    Return the formatted result list.
```
Optimization: The algorithm leverages Neo4j's indexing on the rdfs__label property for rapid query execution**.**

Result Ranking Algorithm
Input: Subgraph results from the Knowledge Graph Search algorithm.
Output: Ranked results based on relationship type.
Operations: Sort the subgraph results by the relationship type alphabetically.

Pseudo-code:
```
Function rank_results(subgraph):
    Sort subgraph by the `Relation` property
    Return sorted subgraph
```
Optimization: Simplified ranking logic ensures minimal computational overhead

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.

```
Checking neighbor counts...
Neighbors of 'http://yago-knowledge.org/resource/State_of_Bahrain': 2
neighbors
Neighbors of 'http://yago-
knowledge.org/resource/Order_of_King_Abdulaziz': 68 neighbors
```

- **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.

```
The nodes 'http://yago-knowledge.org/resource/State_of_Bahrain' and
'http://yago-knowledge.org/resource/Order_of_King_Abdulaziz' are not
within 2 connections.
```

- **Detailed Shortest Path**: Provides the shortest path between two nodes, along with its length, enabling efficient path analysis in the graph.

```
Built-in shortest path:
Shortest path details:
Path length: 4
Path nodes (URIs):
http://yago-knowledge.org/resource/State_of_Bahrain
http://yago-knowledge.org/resource/Bahrain
http://yago-knowledge.org/resource/Arabic_generic_instance
http://yago-knowledge.org/resource/Hussein_of_Jordan
http://yago-knowledge.org/resource/Order_of_King_Abdulaziz
```

- **Subgraph Matching**: Returns a list of dictionaries where each dictionary represents a matched subgraph based on the provided pattern. Each dictionary contains the Node URI, Related Node URI, and the Relationship Type.
  - **Node Similarity Calculation**: Node similarity calculation provides a numerical value representing the cosine similarity score between the vectors of two nodes. This score is a floating-point number normalized between 0 and 1.
- **Search Subgraph Results:** The method returns subgraph data in a user-friendly format. For example:

```
Entity Label: Albert Einstein
Entity URI: http://yago-knowledge.org/resource/Albert_Einstein
Relation: sch__award
Related Entity Label: médaille Matteucci
Related Entity URI: http://yago-knowledge.org/resource/Matteucci_Medal
```

- **Find Similar Entities:** Returns entities sharing the same awards as the reference entity.
- **Relationship-Specific Search:** Identifies entities connected by specific relationships and traversal depth.
- **Explore Multi-Hop Connections:** Discovers entities reachable within a certain number of hops.

```
Finding similar entities...
{'URI': 'http://yago-knowledge.org/resource/Manuel_Cardona', 'Label':
'Manuel Cardona'}
{'URI': 'http://yago-knowledge.org/resource/Franco_Rasetti', 'Label':
'Franco Rasetti'}
Searching specific relationship...
{'End URI': 'http://yago-knowledge.org/resource/Sandro_Pertini',
'Traversal Depth': 2}
{'End URI': 'http://yago-knowledge.org/resource/Herbert_Kelman',
'Traversal Depth': 2}
Exploring multi-hop connections...
{'Connected Entity': 'http://yago-
knowledge.org/resource/Matteucci_Medal', 'Hops': 1}
{'Connected Entity': 'http://yago-
knowledge.org/resource/Manuel_Cardona', 'Hops': 2}
```

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, subgraph matching, node similarity calculations, 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. The algorithms use Neo4j's native pattern-matching capabilities that allow for an efficcheck for direct relationships between two nodes by querying indexed URIs, which allows for very quick retrieval without the need for extensive traversal. This is powerful for very large graphs, and as our database scales.
- **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.
- **Neighbor-Based Queries:** Efficient for large datasets due to indexing and optimized relationship traversal.
- **Shortest Path Algorithm:** Uses Neo4j's built-in shortest path function, scalable for millions of nodes.
- **Multi-Hop Exploration:** Scales well for typical graph traversal but may face memory constraints for highly dense graphs.
  Future Improvements: Implement caching for frequently queried paths.
- Optimize subgraph exploration by prioritizing high-relevance nodes.

Source Code:

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

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.