Big Data Processing (IE494)

Distributed Graph Processing using Apache Spark

(Project Report)

Divyesh Ramani - 202201241
Manan Patel - 202201310

Prof. P.M. Jat
18 November, 2024

Problem and Solution

The project addresses the challenge of **Graph Processing at Scale**. As datasets grow in size and complexity, they can be represented as graphs consisting of nodes and edges. Querying such large-scale graphs on a centralized system becomes increasingly infeasible due to the limitations of memory and processing power.

To tackle this, we implemented a solution proposed by **Prof. Rajshekhar Sunderraman** and **Dr. Janani Balaji**, leveraging the in-memory processing capabilities and distributed architecture of **Apache Spark**. The approach involves decomposing a graph query into smaller components, enabling parallel processing across clusters. This method significantly enhances scalability and efficiency, making it well-suited for large-scale graph analysis.

Our implementation demonstrates how distributed systems like Apache Spark can effectively manage and process massive graphs, enabling advanced querying and insights on complex datasets.

Approach/Algorithm Implemented

1. Node and Edge File Parsing

- Objective: Read graph and query information from files and construct graph structures.

  A. Node File Parsing:

  - Each line represents a node labeled as NodeLabel_NodeType.
  - The mapping of nodes to their types is stored in a dictionary.

```
FUNCTION read_nodes_from_file(file_path):
    INITIALIZE an empty dictionary `node`

    OPEN file at `file_path` in read mode with encoding 'utf-8-sig'
        FOR EACH line in the file:
            REMOVE leading and trailing whitespaces from `line`

            IF `line` is not empty AND `line` is not "Exit":
                IF "_" exists in `line`:
                    SPLIT `line` into `node_label` and `node_type` at
h"_"
                    ADD `node_label` as key and `node_type` as value
to `node`
                ELSE:
                    PRINT "Skipping invalid line:" followed by `line`

    RETURN `node`
```

B. Edge File Parsing:

- Edges are represented as SourceNode_TargetNode_EdgeLabel.
- The edges are stored in dictionaries for both outbound ('o') and inbound ('i') connections in a temp_graph dictionary.

```
FUNCTION read_edges_from_file(file_path, temp_graph):
    OPEN file at `file_path` in read mode with encoding 'utf-8-sig'

    FOR EACH line in the file:
        REMOVE leading and trailing whitespaces from `line`

        IF `line` is not empty AND `line` is not "Exit":
            IF `line` starts with '\ufeff':
                REMOVE the BOM character from `line`

            SPLIT `line` into `user_input` at "_"

            IF both `user_input[0]` and `user_input[1]` exist in
            `temp_graph`:
                # Handle outgoing edges
                IF edge label `(user_input[2], 'o')` exists in
                `temp_graph[user_input[0]][1]`:
                    APPEND `user_input[1]` (converted to integer) to
                    the list of values
                ELSE:
                    CREATE a new list with `user_input[1]` (converted
                    to integer) for the edge label

                # Handle incoming edges
                IF edge label `(user_input[2], 'i')` exists in
                `temp_graph[user_input[1]][1]`:
                    APPEND `user_input[0]` (converted to integer) to
                    the list of values
                ELSE:
                    CREATE a new list with `user_input[0]` (converted
                    to integer) for the edge label
```

2. Graph and Query Graph Construction

- Objective: Convert parsed nodes and edges into a distributed graph representation for both the main graph and the query graph.

A. Graph Representation:

- Nodes, their types, and edge details are organized into tuples and parallelized into RDDs (GraphRDD and QueryRDD).

B. Persistence:

- RDDs are persisted in memory for efficient reuse.

```
INITIALIZE an empty list `graph`

FOR EACH `key` in `temp_graph`:
    APPEND a tuple `(int(key), temp_graph[key][0], temp_graph[key]
[1])` to `graph`

CREATE `GraphRDD` by parallelizing `graph` using
`sc.parallelize(graph)`

PERSIST `GraphRDD` in memory for efficient reuse

INITIALIZE an empty list `query`

FOR EACH `key` in `temp_query`:
    APPEND a tuple `(int(key), temp_query[key][0], temp_query[key].
    [1])` to `query`

CREATE `QueryRDD` by parallelizing `query` using
`sc.parallelize(query)`

PERSIST `QueryRDD` in memory for efficient reuse
```

3. Query Graph Segmentation

- Objective: Divide the query graph into segments for sequential processing.
  - A. Extract each node's label and associated edges.
  - B. Create segments:
    - For start and end nodes, include the node and all edges.
    - For intermediate nodes, include individual edges as separate segments.

```
SET `NumQuery` = the number of elements in `QueryRDD` (using
`QueryRDD.count()`)

INITIALIZE an empty list `segments`

SET `data` = the collected elements of `QueryRDD` (using
`QueryRDD.collect()`)

FOR `i` from 0 to `NumQuery - 1`:
    SET `node_label` = the label of the current query node
    (`data[I][1]`)
    SET `edge_list` = the keys of the edge dictionary for the current
    query node (`list(data[i][2].keys())`)

    INITIALIZE `tmp1` = a list containing `node_label`
    APPEND `edge_list` to `tmp1`

    INITIALIZE `tmp2` = a list containing `node_label`
    FOR EACH `edge` in `edge_list`:
        APPEND `[edge]` to `tmp2` (single-element list)

   IF `i` is the first node (`i == 0`) OR the last node
   (`I == NumQuery - 1`):
        APPEND `tmp1` to `segments`
    ELSE:
        APPEND `tmp2` to `segments`
```

4. **Valid Candidate Initialization**

- Objective: Initialize the search space with the entire graph.

A. Create validRDD, a transformation of GraphRDD that pairs each node with an initially empty list for path tracking.

5. Iterative Subgraph Matching

- Objective: Iteratively refine the search space to match the query graph structure.

1. Node Label Filtering:
   - Filter nodes in validRDD that match the current query segment's node label.

B. Edge Filtering:
   - Further filter based on edges (label and direction) in the query segment.

C. Broadcast Shortlisted Nodes:
   - Broadcast the filtered results to improve parallelism in the next steps.

D. Neighbor Expansion:
   - Explore neighbors of shortlisted nodes using their edges.
   - Add paths of matched nodes to track query subgraph matches.

```
FOR `i` from 0 to `NumQuery - 1`:
    # Step 1: Extract the node label and edges from the current query
      segment
    SET `node_label` = the first element of `segments[i]` (current
    query node label)
    SET `edge_list` = all elements of `segments[i]` starting from the
    second (current query edges)
```

```
# Step 2: Filter valid nodes by node label
SET `shortlistedRDD` = `validRDD` filtered to only include nodes
with label `node_label`

# Step 3: Extract edge labels and directions from the edge list
INITIALIZE empty lists `e_lab` and `e_dir`
FOR EACH `edge` in `edge_list`:
    APPEND the edge label (`edge[0][0]`) to `e_lab`
    APPEND the edge direction (`edge[0][1]`) to `e_dir`

# Step 4: Further filter shortlisted nodes based on edges
FOR `j` from 0 to `len(e_lab) - 1`:
    UPDATE `shortlistedRDD` to only include nodes with
    `(e_lab[j], e_dir[j])` as keys in their edge dictionary

# Step 5: Collect shortlisted nodes and broadcast them
SET `shortlisted_data` = collected elements of `shortlistedRDD`
BROADCAST `shortlisted_data` as `shortlisted_broadcast`

# Step 6: Create a new RDD for next possible nodes (neighbors)
INITIALIZE `newRDD` as an empty RDD
FOR EACH `node` in `shortlisted_data`:
    SET `current_label` = the ID of the current node
    (`node[0][0]`)
    FOR `k` from 0 to `len(e_lab) - 1`:
        UNION the result of filtering `GraphRDD` for neighbors
        reachable via `(e_lab[k], e_dir[k])`:
            - Filter nodes where `x[0]` exists in the adjacency
              list `(e_lab[k], e_dir[k])` of `node`
            - Map the filtered nodes to `[x,
              path_with_current_label]`, where
              `path_with_current_label` is the previous path
              extended with `current_label`

# Step 7: Persist the new RDD
PERSIST `newRDD` to optimize further computations

# Step 8: Update `validRDD` for the next iteration
SET `validRDD` = `newRDD`
```

6. Result Collection

- Objective: Collect and print all matched subgraphs.

A. Final validRDD contains potential subgraph matches.

B. Extract and print paths from matched nodes.

Python Code

[Google Colab: Distributed Graph Processing Using Apache Spark]

Testing

1. Test Cases for `read_nodes_from_file`

- **Objective**: Verify that the function correctly parses nodes from the file and handles malformed data.
- **Test Inputs**:
  A. A file with valid node entries (NodeID_NodeLabel).
  B. A file with invalid entries.
  C. A file with a mix of valid and invalid lines.
- **Expected Outputs**:
  A. The function returns a dictionary with valid node-label mappings.
  B. It skips invalid lines and prints appropriate warnings.
- **Test Scenarios**:
  A. Empty file should return an empty dictionary.
  B. File with only invalid lines should return an empty dictionary with warnings.

2. Test Cases for `read_edges_from_file`

- **Objective**: Verify that edges are added correctly to temp_graph and handle invalid entries gracefully.
- **Test Inputs**:
  A. A file with valid edge entries (FromNodeID_ToNodeID_EdgeLabel).
  B. A file with missing nodes (e.g., 3_4_EdgeLabel where 3 or 4 is not in temp_graph).
  C. A file with invalid formats (e.g., 1-2-EdgeLabel).
- **Expected Outputs**:
  A. Valid edges are added correctly to the graph.
  B. Missing nodes are skipped, and warnings are generated.
- **Test Scenarios**:
  A. In case of valid file, graph edges should update as expected.
  B. In case of empty file graph remains unchanged.
  C. In case of file with invalid edges graph remains unchanged with appropriate warnings.

3. Test Cases for Graph Construction

- **Objective**: Validate that graph and query lists are constructed correctly from temp_graph and temp_query.

- **Test Inputs**:
  A. Manually created temp_graph and temp_query dictionaries.

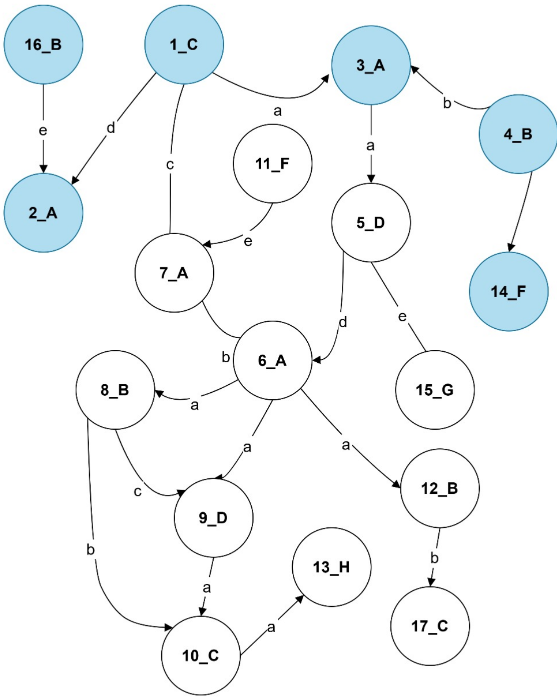
- **Expected Outputs**:
  A. Correct tuples are created for all nodes in temp_graph and temp_query.
- **Test Scenarios**:
  A. temp_graph with no nodes should return an empty RDD.
  B. temp_graph with multiple nodes should return an RDD with the correct structure.

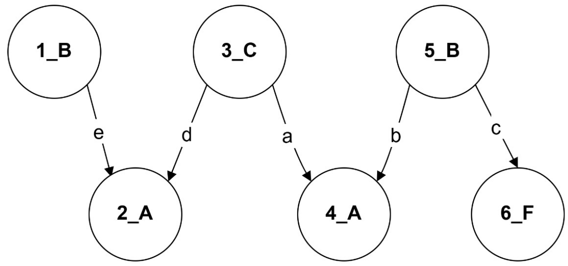
4. Test Cases for Query Segmentation

- **Objective**: Verify the correctness of the segmentation logic.
- **Test Inputs**:
  A. A QueryRDD with varying sizes (small, medium, large).
  B. Query nodes with multiple and zero edges.
- **Expected Outputs**:
  A. segments list correctly segments nodes and edges.
- **Test Scenarios**:
  A. Single query node with no edges should result in one segment with no edge information.
  B. Multiple query nodes with edges should result in segments correctly split nodes and edges.

5. Test Cases for Matching Logic

- **Objective**: Validate the iterative node matching and edge filtering.
- **Test Inputs**:
  A. Simple graphs with known matches.
  B. Complex graphs with cyclic dependencies.
- **Expected Outputs**:
  A. Correctly shortlisted nodes after each iteration.
  B. Accurate final validRDD with matched nodes and paths.
- **Test Scenarios**:
  A. Simple graph with direct matches → Final RDD contains expected matches.
  B. Graph with no matches → Final RDD is empty.
  C. Graph with multiple valid paths → All paths are identified.






```
1_B
2_A
3_C
4_A
5_B
6_F
Exit
```

Query Node

```
1_2_e
3_2_d
3_4_a
5_4_b
5_6_c
Exit
```

Query Edge

```
7_A
8_B
9_D
10_C
11_F
12_B
13_H
14_F
15_G
16_B
17_C
Exit
```

Graph Nodes

```
7_6_b
6_8_a
6_9_a
8_9_c
8_10_b
9_10_a
11_7_e
6_12_a
10_13_a
4_14_c
5_15_e
16_2_e
12_17_b
Exit
```

Graph Edges

```
[16, 2, 1, 3, 4, 14]
```

Output using the python code

References

Balaji, Janani, and Rajshekhar Sunderraman. "Distributed graph path queries using spark." *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. IEEE, 2016.