Project Phase 1

Data Systems – Spring 2026

Due Date: 23:59 Hrs, 4th February, 2026

1 Instructions

- This is a team project. You will be working on this project in a team of three.
- All commits for this phase are to be done in the `main` branch of the repository which is added to your team on GitHub Classroom. You may create separate branches, but you are supposed to merge them into the main branch by the due date.
- No personal queries will be entertained. All queries regarding the project must be posted in the doubts document. **Only doubts clarified in the doubts document will be considered for grading.**
- The late days form must be filled on the day of submission if any commit is made past the due date in the main branch. You must provide the correct number of late days taken.
- Plagiarism checks will be conducted on all submissions. Copied code, whether partially or fully, will be heavily penalized.
- Not following any of the instructions will result in a penalty.

Important Constraints and Implementation Notes

The following constraints must be strictly adhered to during this phase of the project:

- The overall program flow must not be modified. The reference flow can be found in `docs/flow.png`.
- The `cursor` and `bufferManager` classes must be used for handling reading and writing to the pages.
- Private variables of any class must not be changed to public. Access should be provided only through appropriate getter and setter methods.
- The values of `BLOCK_SIZE` and `BLOCK_COUNT` must remain unchanged.
- For all operations, you may store intermediate data structures in memory, however, when accessing the actual data files, **at most two blocks read** from these files may be held in memory at any time.
- The implemented commands should run in a reasonable amount of time.
- Any additional changes or extensions must not affect the correctness or behavior of already implemented table queries.

2 Task: Graph Handling

Extend `SimpleRA` to handle graphs. The application currently supports only tables. Graphs consist of nodes and edges, both of which can have associated attributes.

Each graph is represented using two CSV files:

- A node file, which stores node identifiers and node attributes
- An edge file, which stores the source node identifier, destination node identifier, edge weights, and edge attributes

File Naming Convention

For a graph named `GraphName`:

- Directed graph:
  - `GraphName_Nodes_D.csv`
  - `GraphName_Edges_D.csv`
- Undirected graph:
  - `GraphName_Nodes_U.csv`
  - `GraphName_Edges_U.csv`

Node File Format

The node file `GraphName_Nodes_D.csv` / `GraphName_Nodes_U.csv` stores the node identifier followed by boolean attributes.

The first row of the CSV contains the header `NodeID, A1, A2, ...`. Each subsequent row corresponds to a single node, where `NodeID` uniquely identifies the node and the values of $A_1, A_2, \ldots$ are boolean (0 or 1), specifying whether the corresponding attribute is true (1) or false (0) for that node. The number of attribute columns is not fixed and may vary across graphs.

Format:

```
NodeID, A1, A2, A3, A4
```

Edge File Format

The edge file stores the source node, destination node, edge weight, followed by boolean attributes. The first row of the CSV contains the header `Src_NodeID, Dest_NodeID, Weight, B1, B2, ...`. Each subsequent row represents a single edge, where `Src_NodeID` and `Dest_NodeID` identify the endpoints, `Weight` is a non-negative integer value, and $B_1, B_2, \ldots$ are boolean (0 or 1) edge attributes. The number of attribute columns is not fixed and may vary across graphs. **For undirected graphs, edges are assumed to be bidirectional.**

Format:

```
Src_NodeID, Dest_NodeID, Weight, B1, B2, ...
```

Here:

- `Weight` denotes the cost or weight of an edge and always satisfies `Weight` ≥ 0.

To load a directed graph A into the system, files named `A_Nodes_D.csv` and `A_Edges_D.csv` must be present in the `/data` directory. Similarly, to load an undirected graph A into the system, files named `A_Nodes_U.csv` and `A_Edges_U.csv` must be present in the `/data` directory. You will be tested on large graphs with $2 \leq |V| \leq 10^8$ and $1 \leq |E| \leq 10^8$. **You cannot store the whole graph in memory using any data structure (if the graph does not fit in main memory) for any of the commands.** In particular, graph processing must be performed using disk-based structures, and loading the full graph into in-memory data structures to answer any graph query from memory is not allowed.

Example Input

Node File (`GraphName_Nodes_D.csv`):

```
NodeID, A1, A2, A3, A4
1,0,1,1,1
2,1,1,0,1
3,1,1,1,1
4,1,1,1,1
```

Edge File (`GraphName_Edges_D.csv`):

```
Src_NodeID, Dest_NodeID, Weight, B1, B2, B3, B4
1,2,10,0,1,0,1
1,3,12,1,1,1,1
2,4,6,0,0,1,1
3,4,20,1,1,1,1
```

3 Commands

3.1 LOAD GRAPH

Syntax: `LOAD GRAPH <graph_name> U/D`
Sample Input: `LOAD GRAPH A D`
Sample Output: `Loaded Graph.Node Count:4,Edge Count:4`

- Similar to LOAD, which is used for tables, implement a function to load a graph.
- Given the name of a graph stored in `/data`, and the type of the graph (directed or undirected), this command should load the contents of the corresponding node and edge CSV files and **store them as blocks in the `/data/temp` directory.**
- If a graph referenced does not exist in the `/data` directory, the system should report a semantic error analogous to the table case, indicating that the file does not exist. (`SEMANTIC ERROR: Data file doesn't exist`)
- Similar to tables, the system should not allow two graphs with the same name to coexist. Attempting to load a graph whose name already exists in the system should result in an error, indicating that the graph already exists. (`SEMANTIC ERROR: Graph already exists`). However, a graph and a table may share the same name, and you are expected to design and implement an appropriate mechanism to support this behavior.
- **Note:** The `.csv` extension, the Nodes/Edges suffix, and the graph type must be appended to the graph file name in the code itself and should not be provided in the query.

Please note : The complete graph must be stored in a form that enables efficient traversal queries. This representation may differ from the storage format used for node and edge data. The objective is to minimize the number of block accesses; therefore, an efficient page design should be chosen while loading the graph, keeping in mind the requirements of other operations that need to be supported.

3.2 EXPORT GRAPH

Syntax: `EXPORT GRAPH <graph_name>`
Sample Input: `EXPORT GRAPH A`

- Exports the data of graph `A` (if A is already loaded into the system) and stores it as two files - `A_Nodes_D.csv` and `A_Edges_D.csv` in the `/data` directory, in the **same format as the input files.**
- If a graph referenced does not exist in the system, the system should report a semantic error analogous to the table case, indicating that the graph does not exist. (`SEMANTIC ERROR: Graph doesn't exist`)

3.3 PATH

Syntax:

```
RES <- PATH <graph_name> <src_NodeID> <dest_NodeID> WHERE <conditions>
```

- Checks whether a path exists between source node `src_NodeID` and destination node `dest_NodeID` in graph `graph_name`.
- If a graph referenced does not exist in the system, the system should report a semantic error analogous to the table case, indicating that the graph does not exist. (`SEMANTIC ERROR: Graph doesn't exist`)
- If either the source and/or the destination nodes do not exist in the graph, an error (`Node does not exist`) should be reported.
- If a path exists, the resulting graph is stored as `RES`, i.e., the nodes of the graph will be stored in `RES_Nodes_U/D.csv` and the edges of the graph will be stored in `RES_Edges_U/D.csv`.
- The command outputs `TRUE` or `FALSE`, indicating whether a path exists, along with the total weight of the least weighted path (if there is such a path), separated by a space.
- The least weighted path is the path with the least sum of edge weights.
- Any number of conditions (`Number of Conditions` ≥ 0) may be specified in the `WHERE` clause, and each condition is separated with the `AND` keyword.

Condition Format

Each condition applies to either all nodes (N) or all edges (E) in the path.

General Form:

```
ATTRIBUTE(N|E) == NUMBER
```

- `N` indicates that the condition applies to all nodes in the path.
- `E` indicates that the condition applies to all edges in the path.
- `NUMBER` is either 0 or 1, where `== 1` requires the attribute to be true for all nodes (or edges) in the path, and `== 0` requires it to be false.

- The `== NUMBER` part of a condition may be omitted only when the condition requires all nodes (or edges) in the path to have an identical boolean value for the specified attribute. In this case, the attribute must be uniform across the path, but the value itself may be either 0 or 1. For example, `A1(N)` requires that every node in the path has the same boolean value for attribute `A1`.
- `ATTRIBUTE` may also be `ANY`. This condition checks whether all nodes (or edges) in the path share the same boolean value for any attribute. For example, `ANY(N) == 1` requires every node in the path to have a boolean value of 1 for some attribute, and the same attribute must be consistent across all nodes in the path.

Examples

Assume the following input files for a graph **G**:

G_Nodes_D.csv:

```
NodeID,A1,A2,A3,A4
1,0,1,1,1
2,1,1,0,1
3,1,1,1,1
4,1,1,1,1
```

G_Edges_D.csv:

```
Src_NodeID,Dest_NodeID,Weight,B1,B2,B3,B4
1,2,10,0,1,0,1
1,3,12,1,1,1,1
2,4,6,0,0,1,1
3,4,20,1,1,1,1
```

Example 1: Check if there is a path between nodes `1` and `4` of graph `G` such that attribute `A3` of all nodes is `1` and attribute `B2` of all edges is the same.
Input: `RES_GRAPH <- PATH G 1 4 WHERE A3(N) == 1 AND B2(E)`
Output: `TRUE 32`
Explanation:
The only valid path in which all nodes have a boolean value of `1` for the node attribute `A3`, and all edges share the same boolean value (here, 1) for the edge attribute `B2`, is:

$$1 \rightarrow 3 \rightarrow 4$$

Therefore, the total path weight is computed as the sum of the weights of the edges `1-3` and `3-4`:

$$12 + 20 = 32$$

Note that the edge `2-4` is not considered during path computation because its `B2` attribute has the boolean value `0`. Since the condition is defined over `B2(E)`, all edges in a valid path must share the same `B2` value; in this case, All the other edges have `B2 = 1`. As a result, the edge `2-4`, which has `B2 = 0`, is excluded. The node 2 also has an A3 value of 0, another reason for it to not be considered.

Example 2: Check if there is a path between nodes `1` and `4` of graph `G` such that `A1` of all nodes is 1, `A3` of all nodes is 0 and all edges have a boolean value of 1 for some attribute.
Input: `RES_GRAPH <- PATH G 1 4 WHERE A1(N) == 1 AND A3(N) == 0 AND ANY(E) == 1`
Output: `FALSE`
Explanation:
There is no such path from 1 to 4 with the value of the node attribute A1 = 1 because the value of the node attribute A1 for the start node does not satisfy the condition.

Example 3: Check if there is a path between nodes `1` and `4` of graph `G` such that attribute `A2` of all nodes is `1`.
Input: `RES_GRAPH <- PATH G 1 4 WHERE A2(N) == 1`
Output: `TRUE 16`
Explanation:
There exist two valid paths from node `1` to node `4` in which all nodes have a boolean value of `1` for the node attribute `A2`.

$1 \rightarrow 2 \rightarrow 4$ and $1 \rightarrow 3 \rightarrow 4$

The total weight of the first path is:

$10 + 6 = 16$

The total weight of the second path is:

$12 + 20 = 32$

Since both paths satisfy the given attribute constraints, the path with the minimum total weight is selected. Therefore, the output is `TRUE 16`.

3.4 DEGREE

Syntax: `DEGREE <graph_name> <node_id>`
Example: `DEGREE A 3`

- Computes and outputs the degree of the specified node in graph `graph_name`.
- For an undirected graph, the degree is defined as the number of edges incident on the given node.
- For a directed graph, the degree is defined as the sum of the in-degree and out-degree of the given node.
- The graph must be loaded into the system before this command is executed; otherwise, an error should be reported.
- If the given node does not exist in the graph, an error `Node does not exist` should be reported.
- If a graph referenced does not exist in the system, the system should report a semantic error analogous to the table case, indicating that the graph does not exist. (`SEMANTIC ERROR: Graph doesn't exist`)

Example: Compute the degree of node `3` in the directed graph `G` above.
Input: `DEGREE G 3`
Output: `2`
Explanation:
In the directed graph G, node `3` has one incoming edge ($1 \rightarrow 3$) and one outgoing edge ($3 \rightarrow 4$).

Therefore, the degree of node `3`, defined as the sum of its in-degree and out-degree, is:

$$1 + 1 = 2$$

3.5 PRINT GRAPH

Syntax: `PRINT GRAPH <graph_name>`
Example: `PRINT GRAPH A`

- Prints the contents of the graph.
- The graph metadata (number of nodes, number of edges, and graph type (directed or undirected)) must be printed, one below the other, with a blank line at the end of the metadata.
- Additionally, both the node and edge files corresponding to the graph must be printed in order of their occurrence in the path (or order of occurrence in the loaded files if the graph is not a result of the PATH command), with nodes and edges separated by a blank line.
- If a graph referenced in a query does not exist in the system, the system should report a semantic error analogous to the table case, indicating that the graph does not exist. (`SEMANTIC ERROR: Graph doesn't exist`)
- Output includes:
  - `GraphName_Nodes_D/U`
  - `GraphName_Edges_D/U`

PRINT GRAPH Output for Resultant Graph

If the resulting nodes and edges from Example 1 above are stored in `RES_GRAPH_Nodes_D.csv` and `RES_GRAPH_Edges_D.csv` respectively, the output of `PRINT GRAPH RES_GRAPH` will be:

```
3
2
D

1,0,1,1,1
3,1,1,1,1
4,1,1,1,1

1,3,12,1,1,1,1
3,4,20,1,1,1,1
```

4 Report

Submit a `Report.md` in the `docs` directory including:

- Detailed explanation of implementation for each command (logic, page design, block access, error handling).
- A clear justification for the chosen page design, explaining why it is appropriate and efficient for the operations supported.
- Assumptions (under a separate **Assumptions** heading).

- Contribution of each team member.

5 Evaluation

Evaluation will consist of two components:

- Automated scripts will be run to assess correctness, performance, and adherence to the specified constraints. Outputs that do not strictly follow the specified format will be considered incorrect.
- An in-person viva to assess understanding of the implemented code.

You will be evaluated on the optimizations used to minimize block accesses while ensuring that memory constraints are not violated.