CS39202 : Database Management Systems Laboratory

Term Project Report | Large Scale Graph Processing

Members:

Subham Ghosh (20CS10065)
Anubhav Dhar (20CS30004)
Aritra Mitra (20CS30006)

Contents

1 Introduction

We have created G$^{\text{M}}$L *(Graph Manipulation Language)*, a query and manipulation language solely for graphs. This provides a command line interface to query, manipulate and retrieve different salient aspects of a graph, through simple and intuitive commands. This makes working with graphs easier in a database management context, as traditional query languages like SQL are not particularly well-suited for working with graphs.

1.1 Use cases

G$^{\text{M}}$L is a tool which can have many prospective use cases, as it works with a variety of graphs. Some of the use cases are as follows:

Social Networks: Social networks are a classic use case for graph databases, and a graph manipulation language can be used to query and manipulate data related to user relationships, content sharing, messaging, and more.

Recommendation Engines: Recommendation engines often use graph data to model user behavior and generate personalized recommendations. A graph manipulation language can be used to query the graph data and generate recommendations based on various criteria, such as user interests, demographics, and past behavior.

Fraud Detection: Graph databases can be used to model relationships between entities and detect fraudulent activities, such as money laundering or identity theft. A graph manipulation language can be used to query the graph data and identify suspicious patterns or behaviors.

Logistics and Supply Chain: Graph databases can be used to model and optimize logistics and supply chain networks. A graph manipulation language can be used to query the graph data and optimize various aspects of the network, such as route planning, inventory management, and supplier relationships.

Knowledge Graphs: Knowledge graphs are a powerful tool for organizing and querying complex information. A graph manipulation language can be used to query the graph data and answer complex questions related to the relationships between various entities and concepts.

IoT Networks: Graph databases can be used to model and optimize IoT networks. A graph manipulation language can be used to query the graph data and optimize various aspects of the network, such as device connectivity, data transmission, and energy usage.

1.2 Operations [Objective]

Some basic operations supported by G$^{\text{M}}$L are as follows:

§ Shortest Distance between queried nodes u and v

§ Longest Shortest Distance in the graph (Graph Diameter)

- § Size of Largest *strongest connected component (SCC)*
- § Size of SCC containing queried node u
- § Whether queried nodes u and v belong to the same SCC
- § Degree, indegree, outdegree of queried node u
- § Node with largest degree, indegree, outdegree
- § List of out-neighbours of a node u
- § PageRank value and the rank of a queried node u

1.3 Salient Features

Disk Access Files are kept of the size of disk blocks (4096B), to make it easier to keep track of disk access.

Indexing Adjacency lists are variable sized, we keep primary clustering ordered index on them.

Direct access Index entry and node records are fixed size. The blocks can be accessed directly.

2 Documentation of commands [Methodology]

2.1 `ld`

The `ld` command is used in G^ML to load a graph from a text file, formatted as in **Stanford SNAP**[1] database. The command takes one argument, the name of the file from which the graph has to be loaded.

Output:

In `stdout`:
N = ⟨number of nodes⟩
M = ⟨number of edges⟩

In `stderr`: Disk access ⟨access number⟩ for each disk access

Example command: `ld web-NotreDam.txt`

! This command creates files in three folders, viz. `adj_lists`, `indices`, `node_records`, and it also creates a file with the metadata of the graph loaded, named `metadata.txt`.

[1]Each line in the file is an edge, denoted by two whitespace-separated integers, the IDs of the source and the destination nodes.

Example file:
```
0 1
1 2
2 0
```

2.2 `rd`

The `rd` command is used in G^{M}L to retrieve metadata of a loaded graph. This command takes no arguments.

Output:

In `stdout`: metadata of the graph

Example command: `rd`

! This command does not interact with the files in `adj_lists`, `indices`, or `node_records`. It reads the data from `metadata.txt`.

2.3 `ret`

The `ret` command is used in G^{M}L to retrieve data of a particular node. The command takes one **necessary** argument, the id of the node whose data has to be retrieved. It also takes one *optional* argument, the type of data to be retrieved. If this argument is omitted, it retrieves all types of data.

Types of data:

`in-degree`: in-degree of a node

`out-degree`: out-degree of a node

`pagerank`: PageRank of a node

`scc-id`: ID of the SCC in which the node belongs

`scc-sz`: Size of the SCC in which the node belongs

`ranking`: the rank of the node according to pagerank

Output:

In `stdout`: metadata of a node

Example command: `ret 543; ret 2389 in-degree`

2.4 `q_scc`

The `q_scc` command is used in G^{M}L to check if two nodes belong to the same strongly connected component*(SCC)*. It takes in two arguments, the nodes for which checking would be done if they are in the same SCC. The **Kosaraju** algorithm is used to find out the SCCs, and this is performed during the `ld` command on the graph.

Output:

In `stdout`: If the two -nodes are in the same SCC or not

In `stderr`: Disk access ⟨access number⟩ for each disk access

Example command: `q_scc 5 653`

2.5 `s_path`

The `s_path` command is used in G$^{\text{M}}$L to find the shortest path between two nodes. It uses a Breadth First Search to find the shortest distance. During this, only the relevant adjacency lists are read (along with the rest of the block) from the disk. It takes two **necessary** arguments, the nodes of which the shortest path has to be found. It also takes an *optional* argument, indicating that the path has to be printed.

Output:

In `stdout`: If a path exists, then distance and the path if it is to be printed, or that no path exists.

In `stderr`: Disk access ⟨access number⟩ for each disk access

Example command: `s_path 5 2392; s_path 435 2 path`

2.6 `neigh`

The `neigh` command is used in G$^{\text{M}}$L to retrieve the out-neighbours of a node in the graph. It takes in one argument, the node for which neighbours are to be retrieved.

Output:

In `stdout`: List of neighbours of the node

In `stderr`: Disk access ⟨access number⟩ for each disk access

Example command: `neigh 64`

3 Data structures for efficient storage and indexing

3.1 Index structure

```
struct index_entry{
  int file_num;
  int file_off;
};
```

3.2 Record structure

```
struct node_record{
  int node_id;
  int in_deg;
  int out_deg;
  int scc_id;
  double pagerank;
};
```

3.3 Note

The index structure is $4{+}4 = 8$ bytes, and the record structure is $4{+}4{+}4{+}4{+}4{+}4{+}8 = 32$ bytes, both of which are factors of the block size (4096B). It is created that way so that the disk blocks are used efficiently, without the need to split a record or an index into two blocks.

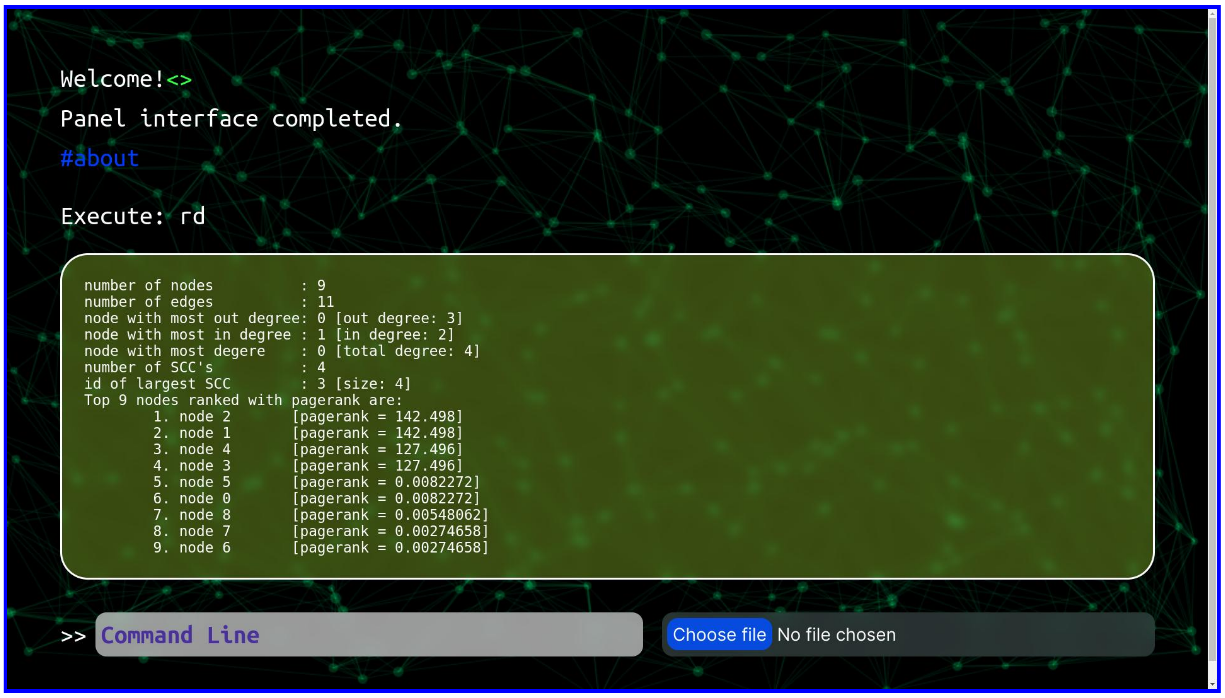
3.4 Adjacency List storage

The adjacency lists of the nodes are of variable length, and so they are allowed to cross the block boundaries, because otherwise the blocks would have been used very inefficiently, with as much as 50% space being wasted in the worst case.
In the implementation of GML, the adjacency lists are stored as follows:
⟨`node id`⟩⟨`length of its adjacency list`⟩⟨`id of neighbour 1`⟩⟨`id of neighbour 2`⟩...
⟨`next node id`⟩...

Remark: The files are all sized 4096B or less. Hence we can get an accurate measure of the number of disk-block accesses based on the number of file accesses required. This was not explicitly needed, but might be used for profiling performance.

4 Frontend

GML as a language does not need a front-end, but we have build a JavaScript-based front-end, deployed as a web-based Command-line Input (CLI) through which GML queries can be performed.



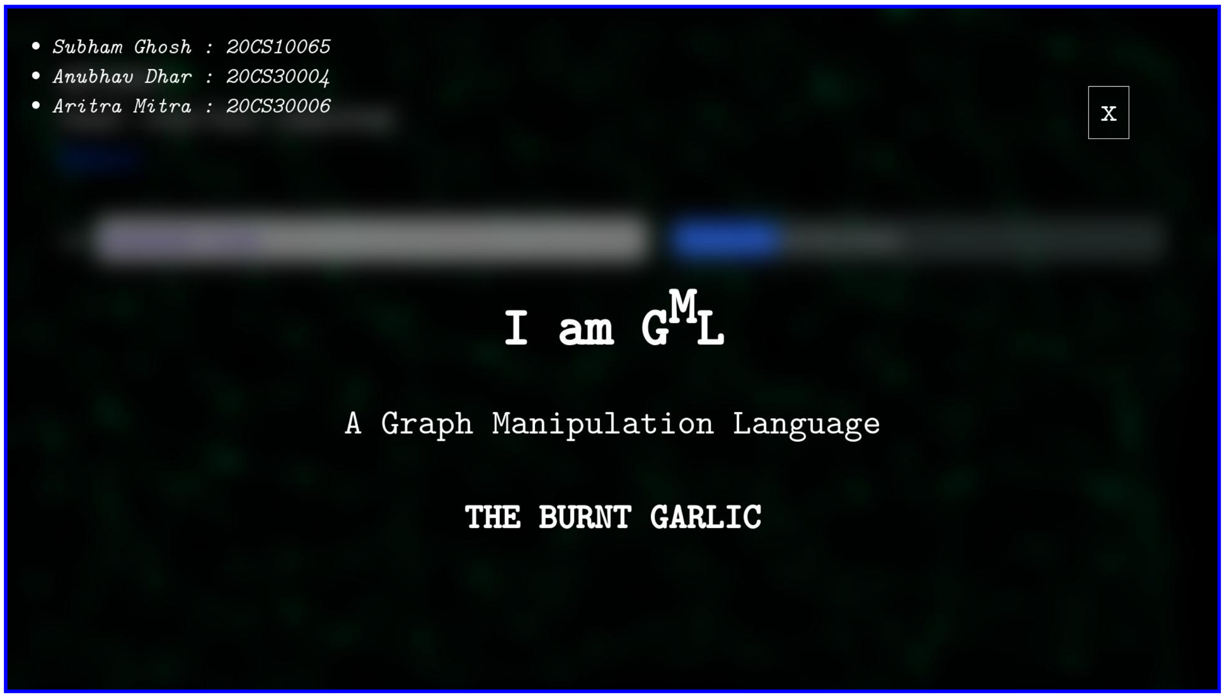
```
Welcome!<>
Panel interface completed.
#about

Execute: rd

number of nodes          : 9
number of edges          : 11
node with most out degree: 0 [out degree: 3]
node with most in degree : 1 [in degree: 2]
node with most degere    : 0 [total degree: 4]
number of SCC's          : 4
id of largest SCC        : 3 [size: 4]
Top 9 nodes ranked with pagerank are:
        1. node 2       [pagerank = 142.498]
        2. node 1       [pagerank = 142.498]
        3. node 4       [pagerank = 127.496]
        4. node 3       [pagerank = 127.496]
        5. node 5       [pagerank = 0.0082272]
        6. node 0       [pagerank = 0.0082272]
        7. node 8       [pagerank = 0.00548062]
        8. node 7       [pagerank = 0.00274658]
        9. node 6       [pagerank = 0.00274658]

>> Command Line
```

The Dashboard


- *Subham Ghosh : 20CS10065*
- *Anubhav Dhar : 20CS30004*
- *Aritra Mitra : 20CS30006*

I am G^ML

A Graph Manipulation Language

THE BURNT GARLIC

The About section

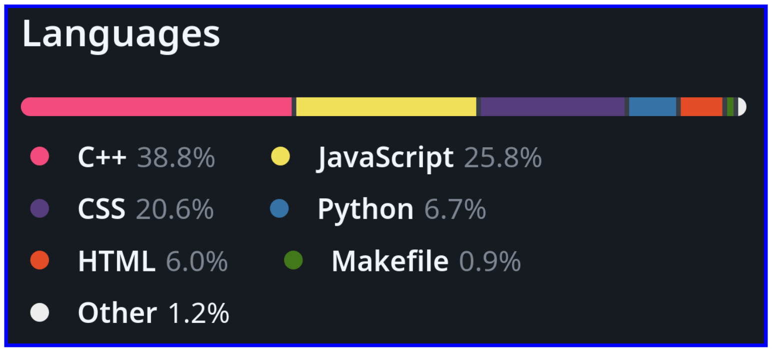
5 Architecture of the system

- The architecture of our database consists of a frontend built using **JavaScript (Ajax and JQuery), HTML and CSS** that sends queries and uploaded files via **XML-HttpRequest** to a **middleware HTTP server** built using **Python**.
- The server first sees if there is any file attachment in the query and if so, it saves the file in the same directory where the graph engine is placed.
- It then analyses the query by first stripping off all characters to the right of the first semicolon to prevent security breaches by any attacker that attempts to run shell scripts in the server by injecting them into the query. **This means GML is immune to injection.**
- Once the query is filtered it is checked against the valid commands that are supported by the engine else the query is flagged invalid.
- Once a valid query is parsed the server spawns a shell subprocess that invokes the corresponding executable module present in the same directory as the graph-engine and supplies the query arguments as command line arguments to the executable.
- The graph engine is essentially a directory that has a modular structure.
  - All the graph functionalities are divided into modules that are stored as executable object codes in the the engine.

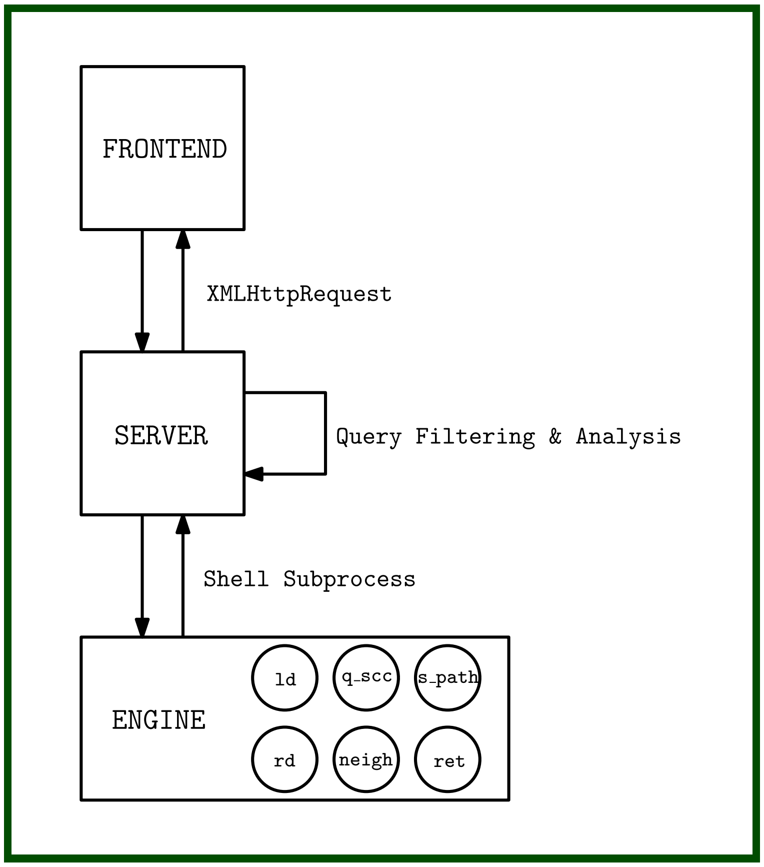
Example: To load a graph the server would load the './ld' executable.

- Once the subprocess ends the server sends back the result to the frontend interface that shows the result by appending a new box element to the document tree.

Note The executables in the graph engine are object code produced by the 'g++' compiler.



The Languages used



The Architecture

6 Efficiency and Performance

Following are the benchmarks measured against the Notre-Dame Web Graph from the Stanford SNAP database in a *intel core i5 (11th gen)* machine. These are just for some arbitrary queries, exact timings of shortest path may differ based on the exact queries.

method	time taken	disk blocks accessed
load table	27.256s	
retrieve metadata	0.012s	
shortest path [0 $\rightarrow$ 4546]	31.884s	
node data query	0.011s	
SCC info query	0.011s	

Table 1: Timing profile

7 Expandability

We would like to further develop GML, and add more functionalities to it. We would like to extend the abilities of GML into further frontiers as follows:

Manipulation Techniques: Giving the user ability to add/remove nodes/edges will be a big leap in the abilities of GML.

Complex Queries: Combining multiple queries, and letting the user perform more complex actions will increase the amount of work possible to do in the realm of GML.

Clickable GUI: We could make complicated applications using clickable GUI or use it as a sub routine for a larger project.

8 References

1. **Github repo of our implementation**
2. Stanford SNAP Database
3. Simple PageRank Algorithm
4. Kosaraju's Algorithm
5. DBMS course slides on file structure