



# GML

## Graph Manipulation Language

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

The Burnt Garlic

DBMS Laboratory TERM PROJECT



# Overview

We have created **GML** (*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.



# Operations Supported

- Loading the graph
- Data about the graph
  - Number of nodes, edges
  - Nodes with max indegree, outdegree, total degree
  - Number of SCC's, size of largest SCC
  - Top 30 (or less) ranked pages by *PageRank*
  - Graph Diameter (for small graphs)
- Shortest path between two nodes
  - The length
  - The path
- Data about the nodes
  - Indegree, outdegree, list of out-neighbours
  - *PageRank* value and rank
  - SCC id and size
- Whether two nodes belong to the same SCC



# Documentation Of Commands

`ld`

Loads a Graph formatted as in Stanford SNAP database. The command takes one argument, the name of the file from which the graph has to be loaded.

`rd`

Reads metadata of a graph which has been loaded in the last `ld` command.

`ret`

Returns metadata about a node in the graph. The command takes one argument, the id of which the data has to be loaded.

`q_scc`

Checks if two nodes are in the same SCC. The command takes two arguments, the nodes for which it is to be checked if they are in the same SCC.

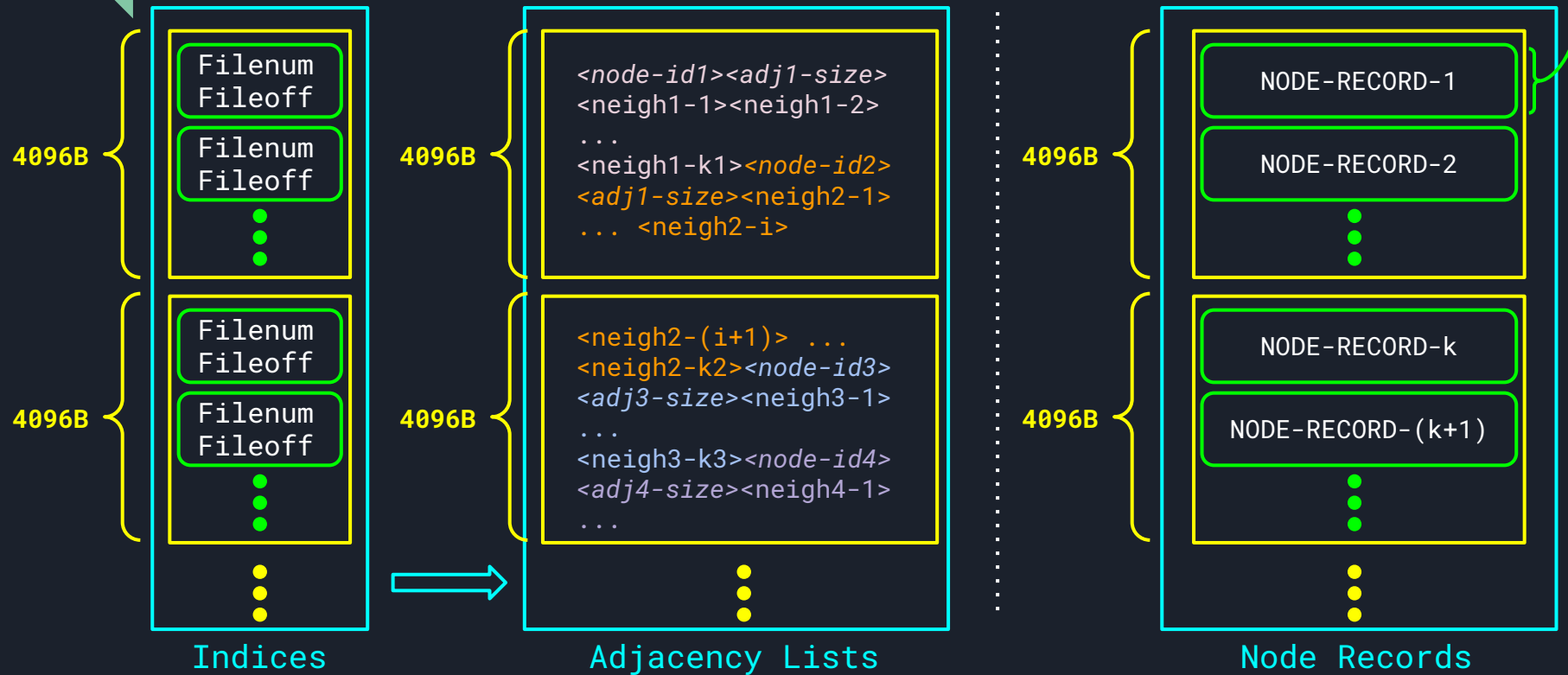
`s_path`

Finds the shortest path between two nodes. The command takes two arguments, the nodes between which the path is to be found.

`neigh`

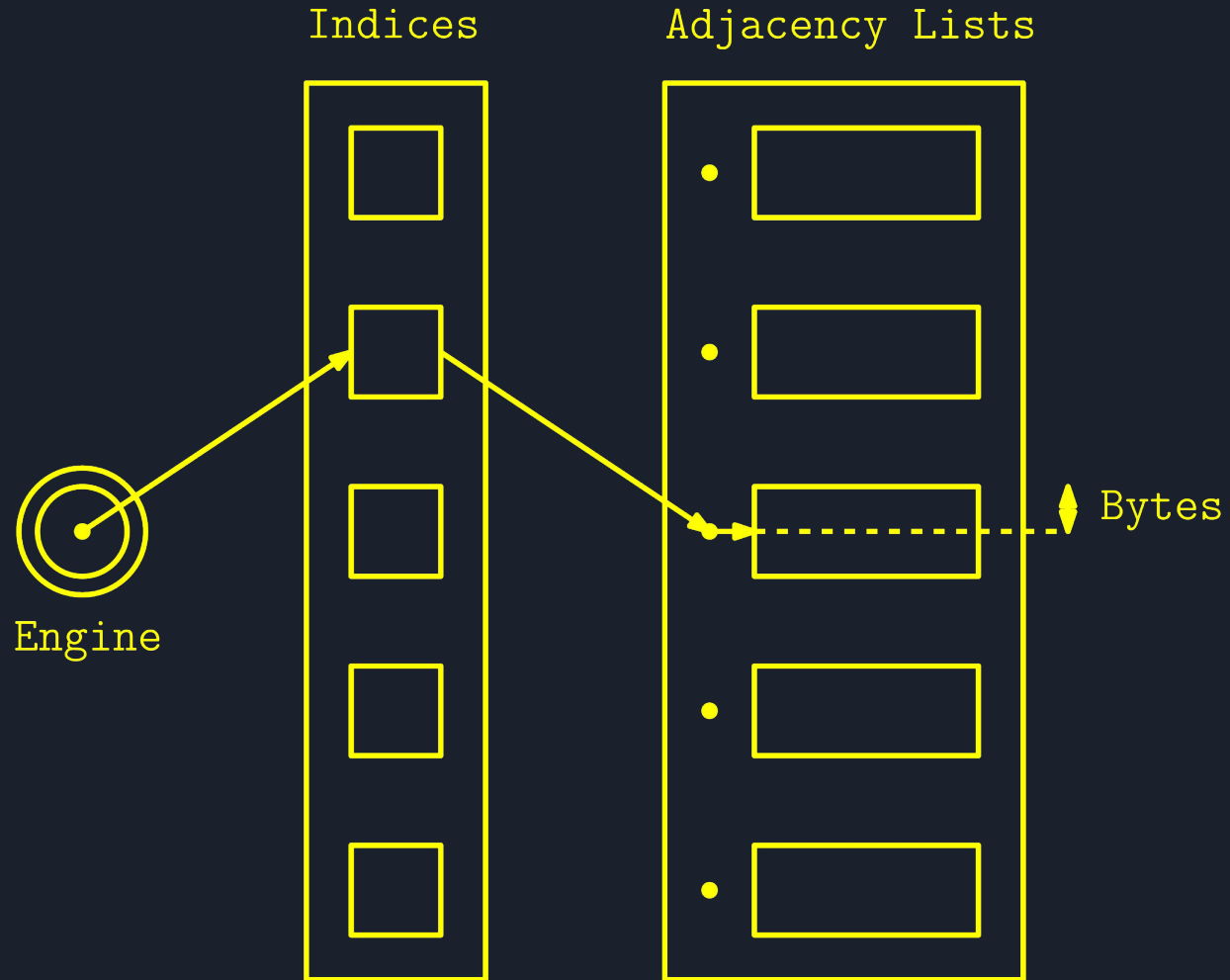
Finds the neighbours of a node in the graph. The command takes one argument, the id of the node of which the neighbours are to be found.

# File Structure



# Efficient Indexing

- *Ordered, Clustering, Primary* Index is used for efficient access start of adjacency list
- One disk access for *index file*, one for the *adjacency list*
- The index file stores the *file number* and the *offset* inside that file






# 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.

Software Stack used:

- Javascript (Ajax and JQuery)
  - HTML
  - CSS
- 



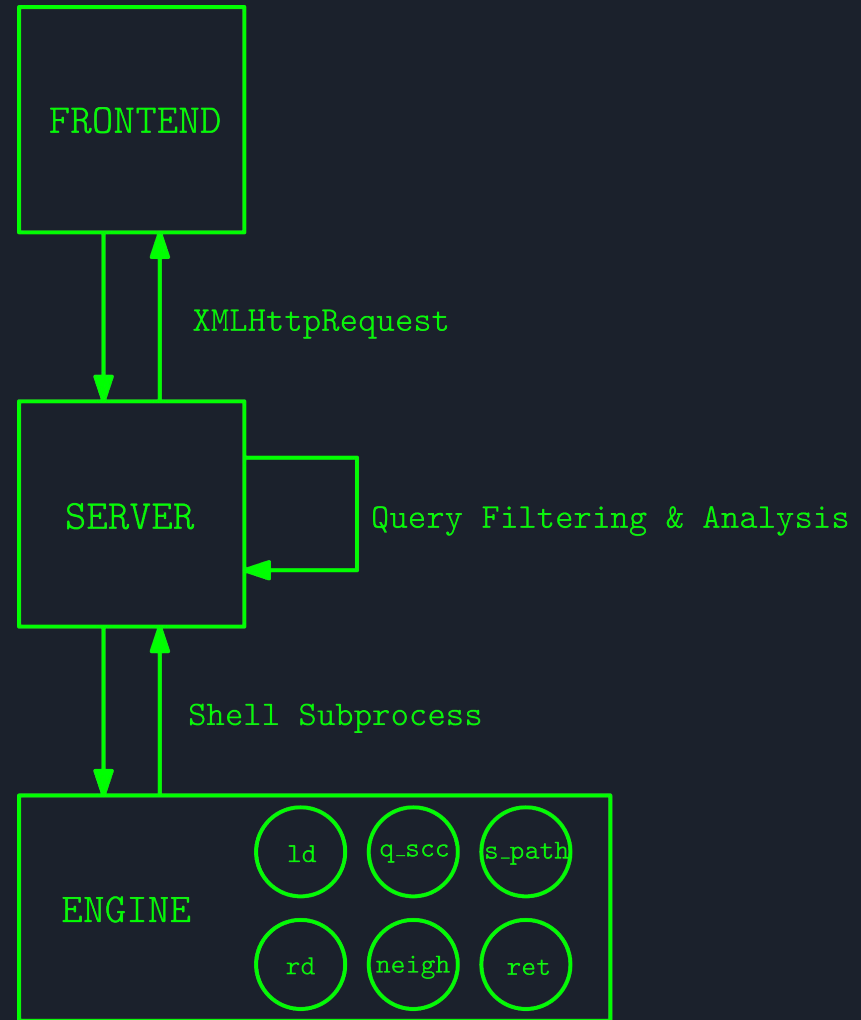
# Architecture

The architecture of our database consists of a frontend 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.***







## Architecture (Continued)

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.*



# 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 and neighbour printing may differ based on the exact queries.

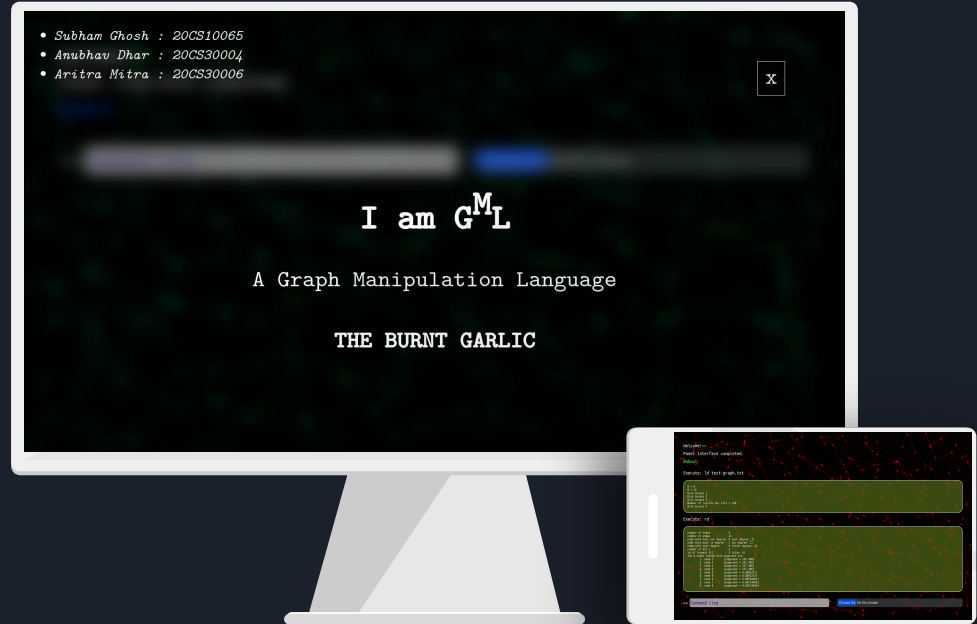
Method	Time Taken	Disk Block Access
Load table	13.321s	5282
Retrieve metadata	0.004s	1
Shortest path [0 → 4546]	0.107s	2366
Shortest path [2 → 33]	0.006s	3
Node data query [node 7]	0.003s	1
SCC info query	0.003s	2
Print neighbour [node 2]	0.004s	3
Print neighbour [node 7137]	0.006s	6



# Expandability

Future plans to expand the horizons and capabilities

- **Edit ability:** Giving users the ability to add/delete nodes/edges into a graph.
- **Complex queries:** Giving users the ability to write complex queries.
- **Clickable GUI:** We could make complicated applications using clickable GUI or use it as a subroutine for a larger project.





Thank you!

Our Github Repo

<https://github.com/frediff/GML>

