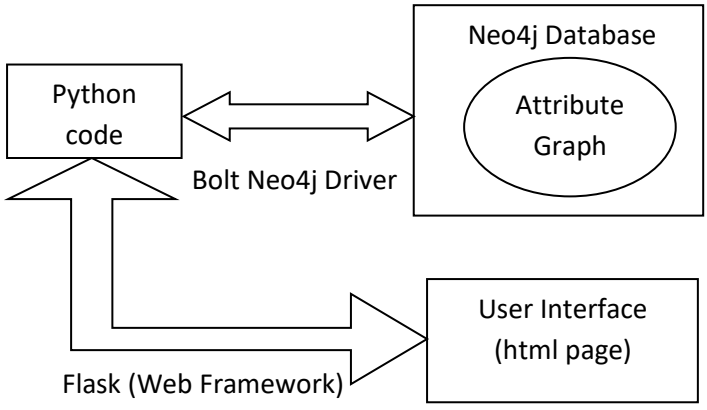
Attribute Graph Implementation in Neo4j

Problem Motivation:

Security has become an important component in today's Databases. Wherein people are facilitating malicious attacks by having access to the entire database. To prevent this, access control policies were invented. The Attribute Based Access Control (ABAC), is one of the most powerful access control policies available. It encompasses popular models, such as the Role-Based Access Control (RBAC) model and can also enforce dynamic policies where authorizations depend on values of user, resource, or environment attributes. These attributes are stored in the form of a graph for faster access. The attribute graph has relations to link nodes within an attribute and in between two different attributes. In this paper, I have built an attribute graph in Neo4j and found the degree of similarity of the valid paths between the source and target nodes provided by the User. Neo4j is a noSQL (graph) database, based on graph theory where the nodes are equivalent to entities and the edges are relationships. These graph databases are efficient in dealing with huge volumes of interconnected data. Due to a plethora of interconnections present in the attribute graph, Neo4j Database would be considered ideal to implement attribute graph, but it has no provisions for accessing individual paths out of all the paths available between source and target nodes. Hence, the implementation of Python code to find valid path out of all the paths available and a user-interface for acquiring the source and target nodes.

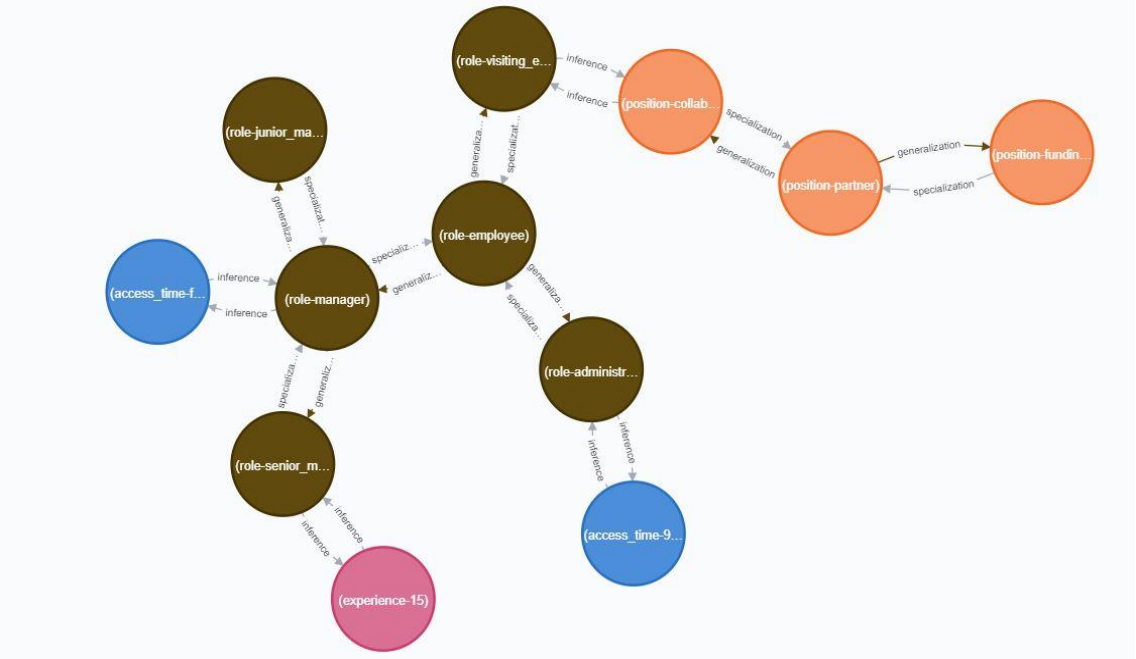
Methodology:

Overview:

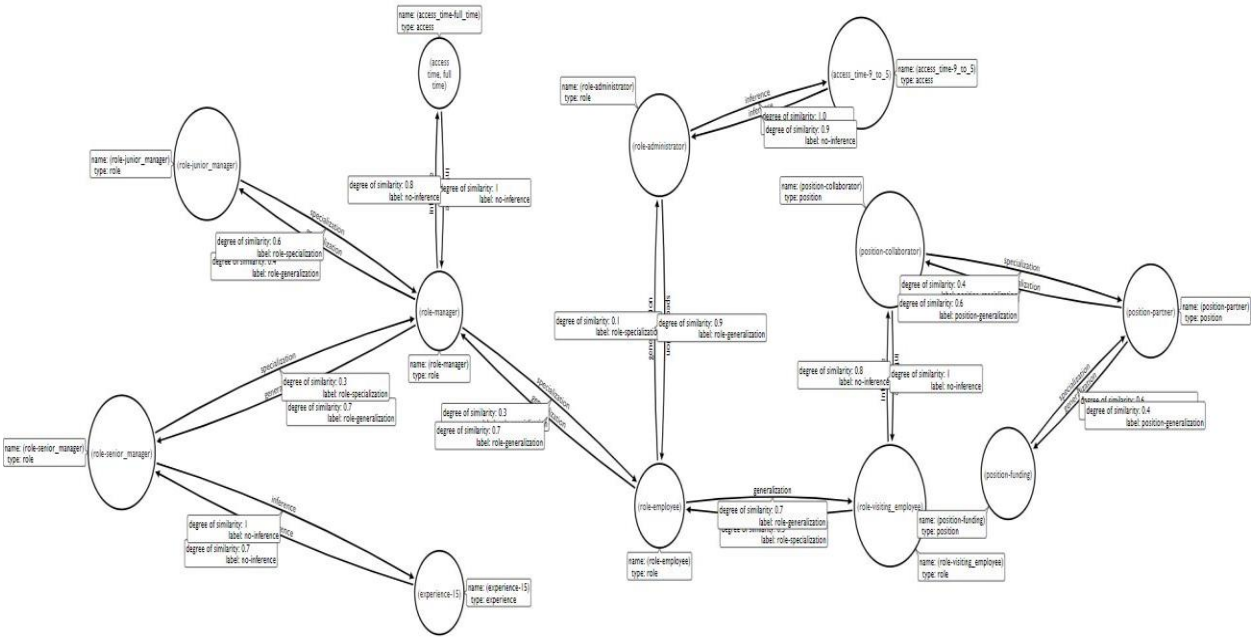


Attribute Graph:

The Attribute graph is a cyclic graph consisting of (N, S, $\overline{S}$, I, λ) where N stands for nodes, S stands for specialization edges, $\overline{S}$ stands for generalization edges, I stands for Inference edges and λ refers to the labels. In this paper, the attribute graph was built for accessing data in a corporate environment. The attribute graph shown below has color notations with respect to the attributes that each node belongs to.



The UML version of the graph shows the properties associated with each nodes and relationships present in the graph. The UML version of the graph is given below. From the graph, we can see the properties associated with each nodes and relations. All the nodes have “name” and “type” properties while the relations have “degree of similarity” and “label” properties. The “name” property provides the name of the node in the Neo4j database and it also provides a reference. The “type” property specifies the attribute to which each node belongs to. The “degree of similarity” property deals with the computation of semantic closeness between the attributes, to make a decision when the given information is incomplete. The “label” property is a way to reference each relation. This UML model is converted into a cypher code which is then fed to the Neo4j Database.



Python code:

The Python code is used to find the valid path between the given source and target nodes. The code is divided into two segments based on the constraints imposed on finding valid path. First Constraint is that, If the path considered contains the nodes of the path with the degree of similarity, then that path is illegal. Secondly, if the path contains both specialization and generalization edges in the same hierarchy/attribute then that path is illegal.

Pseudocode for Constraint-1:

Function Constraint-1(*AG*: *Attribute Graph, s: source node, d: destination node*):

Output: path*, σ(s,d) //* Attribute graph with valid paths, similarity of s with d

Let $AG = (N, S, \overline{S}, I, \lambda)$

Let path = *AG(s,d)*

Let max-path = *AG(max(σ(s,d)))*

IF *N* != *s* and *d* **do** // excluding the source and the target nodes

IF path(*N*) == max-path(*N*) **then** // comparing the nodes

del path

Pseudocode for Constraint-2:

Function Constraint-2(*AG*: *Attribute Graph, s: source node, d: destination node*):

Output: path, *σ(s,d)* // Attribute graph with valid paths, similarity of s with d

Let *AG* = (*N*, *S*, $\overline{S}$, *I*, *λ*)

Let path = *AG(s,d)*

Hierarchy = attribute(*λ*)

For Hierarchy in path **do** // If similar hierarchy exists in the path

IF path(S) == path($\overline{S}$) // If specialization and generalization edges exists.

del path

User Interface:

Built an HTML page to provide user interface by querying source and target nodes and printing the output of the code in the HTML page. The page contains Neo4j Background and has two blank spaces for entering the source and target nodes. These source and target are fed to the code with the help of flask's built-in method called "request.form" which takes in the source and target nodes in the form of key-value pair. The "render template" method of flask connects the HTML page to the code. The output is stored in a dictionary and is dumped into the HTML file using the "jsonify" method of flask. HTML file is given below.

RESULT:

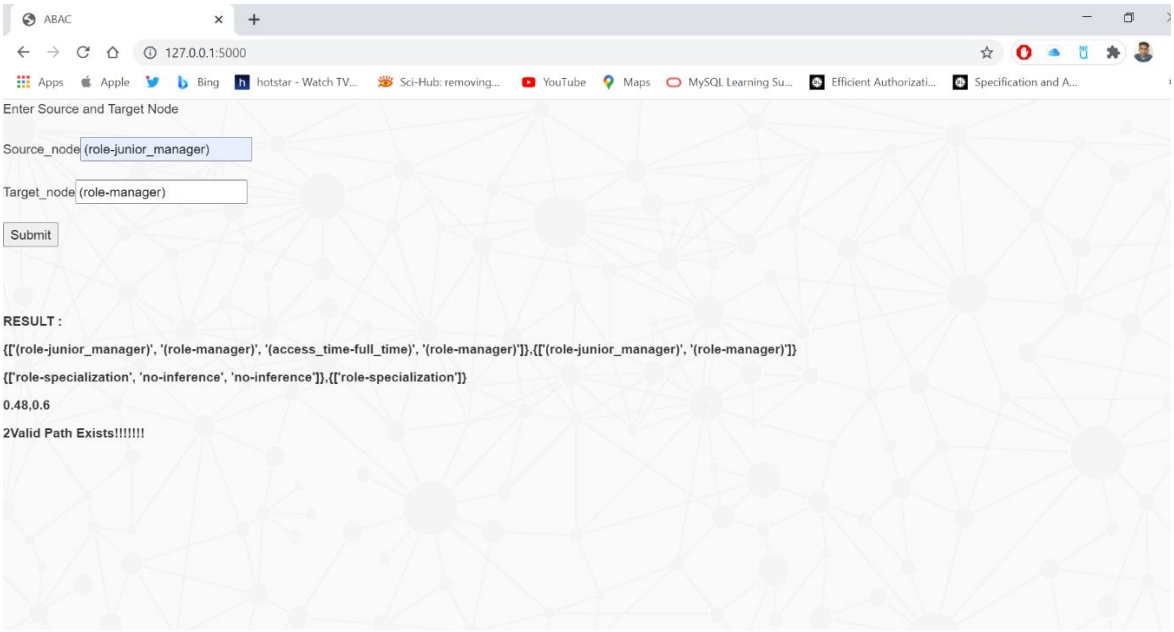
The virtual environment is setup to launch the python code to the HTML file and the set Flask_app command is used for flask to serve the python code. The flask_env default setting is production phase. But we initialize it to development phase to enable debugging the code and also allowing the hosting every time a change is made to the python code.

```
(neo4j-movies) C:\Users\sridh\CS533-project>set FLASK_APP=attribute_graph.py

(neo4j-movies) C:\Users\sridh\CS533-project>set FLASK_ENV=development

(neo4j-movies) C:\Users\sridh\CS533-project>flask run
 * Serving Flask app "attribute_graph.py" (lazy loading)
 * Environment: development
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 267-380-901
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

We can see that HTML file is hosted on the same IP address displayed in the command prompt. The Result displayed consists of Nodes in the valid path, relations in the valid path displayed in the form of hierarchy-type basis, degree of similarity of the valid path and the number of valid paths between the source and the target node provided by the User.



Thus, the Attribute Graph is constructed, and the valid path is found along with the degree of similarity using Neo4j Database.