Multilevel Security for Graph Databases

Martin Christensson
Linköping University
Linköping, Sweden
march230@student.liu.se

Oscar Järpehult
Linköping University
Linköping, Sweden
oscja033@student.liu.se

***Abstract*—With the ever increasing size and complexity of data sets, alternatives to storing data in traditional tabular SQL databases has been on the rise. One popular way of doing this is using a graph database. However, implementing previously prevalent methods of managing access control for tabular databases poses challenges of implementation to graph databases. In this report we present an implementation of multilevel security in a graph database using Neo4j. The exact implementation are generated with Cypher queries and different example queries are presented. Furthermore, we discuss the difficulties of our implementation of multilevel security and the possible uses of the implementation.**

***Index Terms*—graph database, Cypher, multilevel security, Neo4j**

I. Introduction

In every IT system that is built today there is some form of secret information or data that should be accessible while still remaining hidden for other users. Traditional relational databases which have been the ruling way of storing data for decades have solved this by implementing multilevel security where each entry in the database is assigned a security class that determines who can access it and edit it.

With the ever increasing size and complexity of modern data sets, the time it takes for an SQL query to search and traverse a modern database is not in line with the speed of other application processes. These processes will be slowed down waiting for the results of these queries. To combat this problem, new methods of storing data have been researched called NoSQL. One of these methods is the graph database which excels when there are a lot of relations between entries in the database.

With graph databases still being a relatively new concept there is no well defined standard for implementing multilevel security.

This report aims to come up with an idea for how multilevel security can be implemented in a graph database. This will be done by providing specification for how this should be implemented as well as providing an analysis of this idea.

This report will introduce the reader to the relevant topics in section II. In section III we describe the implementation. Section IV and V outline our results as well as a general discussion of the aforementioned results. In section VI we discuss related work and in section VII we present conclusion derived from this study and the presented related work.

II. Background

In order to understand the contents of this report this section will provide a short introduction to three topics. The first subsection explains what a graph database is and what separates it from a traditional relational database. The second subsection explains what multilevel security is as well as how it is implemented in relational databases. The final subsection present the environments used for the implementation.

A. Graph Databases

A graph database consist of nodes which are connected with edges called relationships. A node can contain any number of key-value pairs which is called properties. Every node may also be tagged with one or more labels which is used to determine what role the node have in the graph network. A relationship connect two nodes with a directed edge and a name describing the type of connection between the two nodes. Similar to nodes, relationships can also have multiple properties. [1]

In Fig. 1, an arbitrary graph database is displayed. The circles in the figure is the nodes, with the associated text boxes representing the properties of those nodes and the arrows representing the relationships between those nodes. In this specific example, a person has the property "name" which in this case is *"Peter"*. We can also see that "Peter" has two inbound relations "owned by" which he does to nodes *"Document"*, called *"Doc1"* and *"Doc2"*. These Documents both contain a property *"Content"* which is *"some text"*.

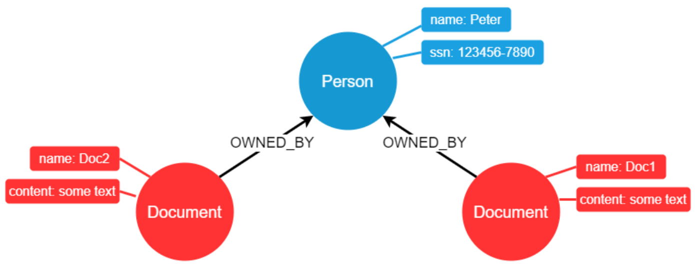

Fig. 1. Example of Graph Database.

B. Multilevel Security

Protecting confidential or sensitive data which is stored in a database is paramount in many instances. To manage access control, a set of multilevel security classes can be implemented to allow the classification of data and of users on a system of hierarchical security levels. This will allow a distinct separation of the rights a certain user might have to access data and what those rights entail. This is referred to as clearance. A user has clearance for data if it has a clearance level equal to or higher than the security level of the requested data. To maintain confidentiality in the system, identification for all users is enforced. This ensures accountability in the system by linking a user to a physical person or an application. [5]

The users specific access to data can be separated to permissions to describe the specific access they have to data. This allows the access control to be fine-grained in stating not only the access of data by a user, but which actions can be taken by that user for the object. These actions are typically Create, Read, Write (Update) and Delete but can range from any defined actions desirable. In this report, only Read and Write access privileges will be utilized. The main aims of these finer granular hierarchical security levels is not only to ensure the prevention of unauthorized individuals from accessing data, but also preventing individuals from declassing data or information. By restricting the action to write, information is able to be read but not changed, altered or declassified by a user. This preserves the integrity and confidentiality of information. [2]

Fig. 2 shows an example implementation of multilevel security. For this example a "no read up, no write down"-model is implemented. This allows for data to only be read which has a lower security classification and to write to a higher classification. Data with security classifications equal to that of the User in this case can both read and write to data. This examples illustrates how both information and users can be assigned classifications and based on this, restricted to access to the system.

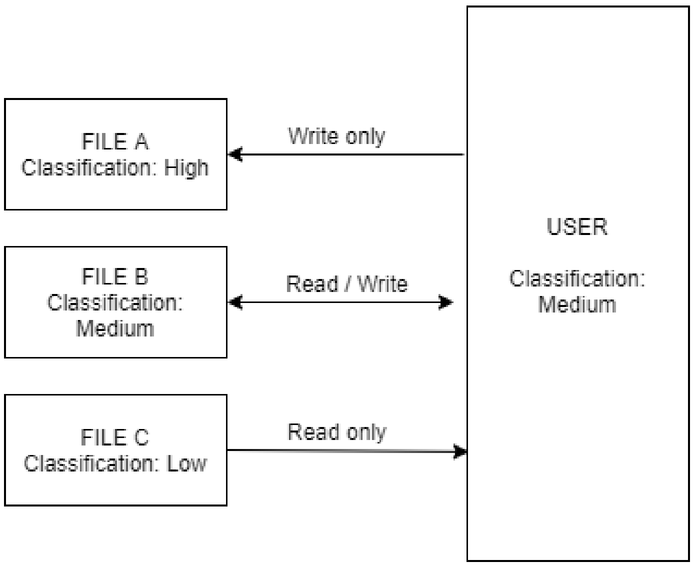

Fig. 2. Example of Multilevel security.

C. Neo4j & Cypher

Neo4j is a graph database management system (GDBMS) developed by Neo4j, Inc and made publicly available since 2007. It is the most popular GDBMS and is the 21st most popular database management system (DBMS) overall according to DB-Engines 2020 ranking. [3] Neo4j uses its own query language called Cypher. The language uses a SQL-like syntax and is also open source. [4]

III. Method

This section shows how multilevel security for a graph database in Neo4j was implemented. A small graph database example without any security classification will be created to begin with and then expanded upon.

A. Setup Neo4j Graph Database

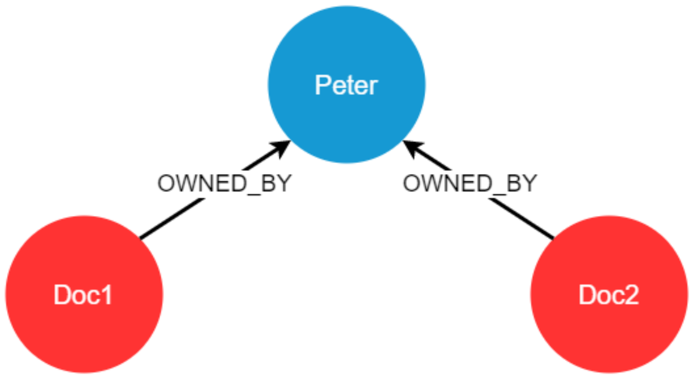

Fig. 3. Initial Graph Database.

The initial graph is shown in Fig. 3. The graph contains two types of entities which is labeled either "Person", here labelled "Peter", or "Document". There is also a type of relation "OWNED_BY" which shows who own a document. In this case we have a simple database that has two documents owned by the person Peter.

Code to generate graph in Fig. 3:

```
CREATE
(p:Person {name: 'Peter', ssn:
    '123456-7890'}),
(d1:Document {name: 'doc1', content: 'some
    text'}), (d2:Document {name: 'doc2',
    content: 'some text'}),
(d1)-[:OWNED_BY]->(p), (d2)-[:OWNED_BY]->(p)
```

This database can now be queried for all documents with the following code:

```
MATCH (n:Document) return n
```

B. Setup Security Levels

The current database has no way of granting or denying access to single entities based on user. If someone has access to the database that person will be allowed read and edit

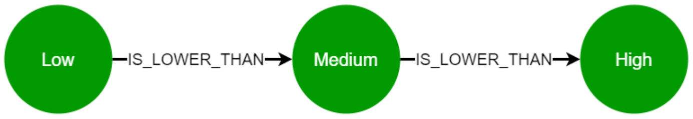

Fig. 4. Security Classes Graph.

everything in the database. This will change by setting up security levels.

Fig. 4 shows a graph that contains three different security levels: Low, Medium and High. They are chained together with the relation type "IS_LOWER_THAN" in order to determine which security levels is above or below the others. By using these relations we will be able to give a higher security level the same privileges as the lower ones.

Code to generate graph in Fig. 4.

```
CREATE (low:SecurityLevel {name: 'low'}),
    (med:SecurityLevel {name: 'medium'}),
    (high:SecurityLevel {name: 'high'}),
(low)-[:IS_LOWER_THAN]->(med),
    (med)-[:IS_LOWER_THAN]->(high)
```

C. Set Security Level of Nodes

In order to set security levels of nodes in the database the graphs in Fig. 3 & Fig. 4 will be combined by adding relations between the security levels and the other nodes.

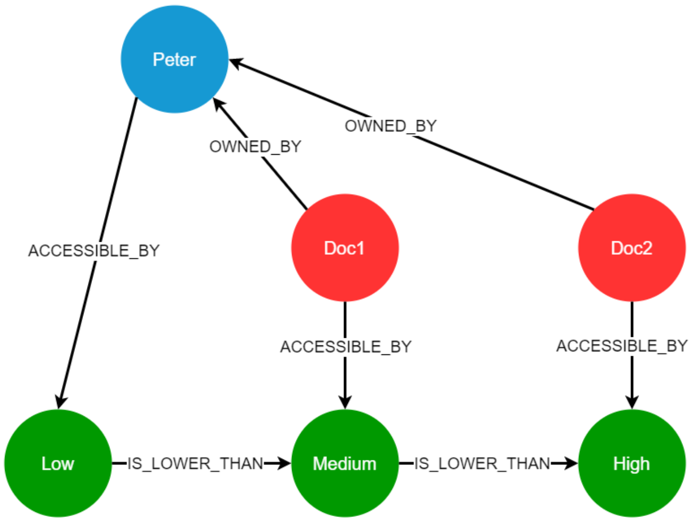

Fig. 5. Security Level on Nodes.

Fig. 5 shows the two subgraphs merged with the new relation type "ACCESSIBLE_BY" which connects a node to one of the three security levels. A user with the low security level will only be able to access the node labeled "Person" while the other two security levels will grant access to one document each as well as inherit all content with a lower security level.

Code to add the new relations:

```
MATCH (p:Person {name: 'Peter'}),
(low:SecurityLevel {name: 'low'}),
    (med:SecurityLevel {name: 'medium'}),
    (high:SecurityLevel {name: 'high'}),
(d1:Document {name: 'doc1'}),
(d2:Document {name: 'doc2'})
CREATE (p)-[:ACCESSIBLE_BY]->(low),
    (d1)-[:ACCESSIBLE_BY]->(med),
    (d2)-[:ACCESSIBLE_BY]->(high)
```

Instead of accessing a node directly we can now take the security level into account to get all nodes we should be able to access with a given security level by using the following syntax:

```
(Normal MATCH query) -[:ACCESSIBLE_BY]->
    (objsec:SecurityLevel)
    -[:IS_LOWER_THAN*0..2]->
    (subsec:SecurityLevel {name: 'low' or
    'medium' or 'high'}) (normal RETURN
    syntax)
```

For example we can access all nodes with a security level of medium or lower with the code:

```
MATCH (n) -[:ACCESSIBLE_BY]->
    (objsec:SecurityLevel)
    -[:IS_LOWER_THAN*0..2]->
    (subsec:SecurityLevel {name: 'medium'})
    return n
```

D. Separate Read and Write Access

In the current model we do not differentiate on read and write access. This can be done by replacing the "ACCESSIBLE_BY" relationship with "READABLE´_BY" and "WRITABLE_BY".

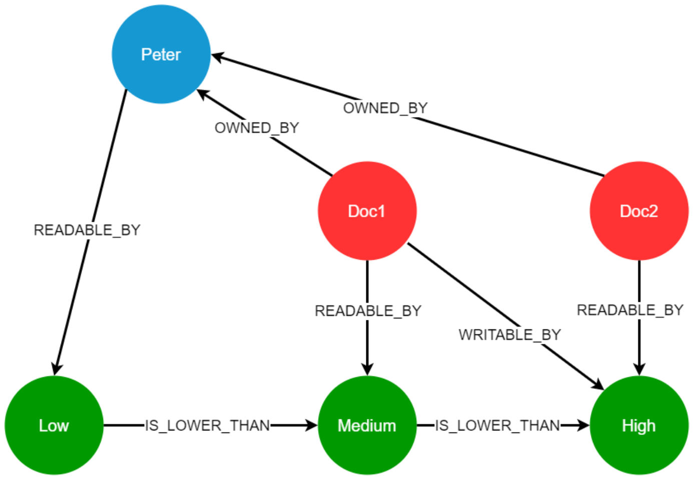

Fig. 6. Write & Read Access

In Fig. 6 the "READABLE_BY" relation has the same connections as "ACCESSIBLE_BY" had in the previous model and a new relation with the type "WRITABLE_BY" has been added.

Code to remove "ACCESSIBLE_BY":

```
MATCH ()-[a:ACCESSIBLE_BY]->()
DELETE a
```

Code to add read and write relations:

```
MATCH (p:Person {name: 'Peter'}),
(low:SecurityLevel {name: 'low'}),
    (med:SecurityLevel {name: 'medium'}),
    (high:SecurityLevel {name: 'high'}),
(d1:Document {name: 'doc1'}),
(d2:Document {name: 'doc2'})
CREATE (p)-[:READABLE_BY]->(low),
    (d1)-[:READABLE_BY]->(med),
    (d2)-[:READABLE_BY]->(high),
(d1)-[:WRITABLE_BY]->(high)
```

Nodes will now be matched with the same syntax stated in [C] by replacing "ACCESSED_BY" with "READABLE_BY". In order to edit a node we will use the following syntax:

```
(Normal MATCH query) -[:WRITABLE_BY]->
    (objsec:SecurityLevel)
    -[:IS_LOWER_THAN*0..2]->
    (subsec:SecurityLevel {name: 'low' or
    'medium' or 'high'})
(normal SET syntax)
```

For example a user with the high security level can edit the content of Doc1 with the following code:

```
MATCH (d:Document {name: 'doc1'})
    -[:WRITABLE_BY]-> (objsec:SecurityLevel)
    -[:IS_LOWER_THAN*0..2]->
    (subsec:SecurityLevel {name: 'high'})
SET d.content = 'new text'
```

E. Security Level on Single Attributes

What if we want to set a security levels for a single attribute? A node may have properties that needs to have separate security levels and that is currently not possible since the whole node is assigned a security level. We will solve this by separating some of the properties into new attribute nodes.

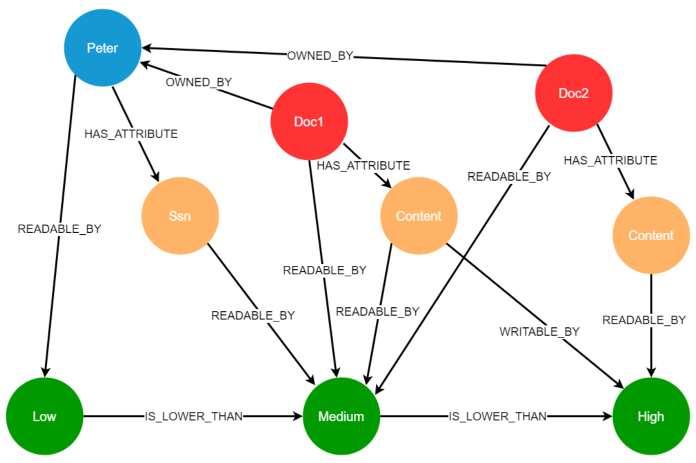

Fig. 7. Security Level on Attributes

Fig. 7 shows the modification to the graph. There are three new attribute nodes connected to their parent node with the relationship "HAS_ATTRIBUTE". Ssn can now only be read with a medium security level or higher and Doc2 will now be visible for medium security level while the content of it is only being visible to high security level.

Code to generate graph in fig. 7 from scratch:

```
CREATE
(p:Person {name: 'Peter'}), (a1:Attribute
    {ssn: '123456-7890'}),
(d1:Document {name: 'doc1'}), (a2:Attribute
    {content: 'some text'}),
(d2:Document {name: 'doc2'}), (a3:Attribute
    {content: 'some text'}),
(low:SecurityLevel {name: 'low'}),
    (med:SecurityLevel {name: 'medium'}),
    (high:SecurityLevel {name: 'high'}),
(low)-[:IS_LOWER_THAN]->(med),
    (med)-[:IS_LOWER_THAN]->(high),
(d1)-[:OWNED_BY]->(p), (d2)-[:OWNED_BY]->(p),
(p)-[:HAS_ATTRIBUTE]->(a1),
    (d1)-[:HAS_ATTRIBUTE]->(a2),
    (d2)-[:HAS_ATTRIBUTE]->(a3),
(p)-[:READABLE_BY]->(low),
    (a1)-[:READABLE_BY]->(med),
    (d1)-[:READABLE_BY]->(med),
(a2)-[:READABLE_BY]->(med),
    (a2)-[:WRITABLE_BY]->(high),
    (d2)-[:READABLE_BY]->(med),
(a3)-[:READABLE_BY]->(high)
```

IV. Analysis

In this section we will test the database model and make sure the security levels works as intended. This will be done by executing queries using the syntax given in section III. We will test both read and write access.

A. Read Access

In order to make sure read access works as intended, a query that returns all nodes will be executed for each security level. This is done be running the following query three times with every security level inserted into the name field:

```
MATCH (n) -[:READABLE_BY]->
    (objsec:SecurityLevel)
    -[:IS_LOWER_THAN*0..2]->
    (subsec:SecurityLevel {name: 'low'})
    return n
```

Fig. 8 shows the results when returning all nodes for a given security level. Nodes with a lower security level is also visible for the higher security levels. This indicates that the security levels inherit nodes with a lower security level correctly. When comparing fig. 8 to Fig. 7 it can also be confirmed that a node is only visible if it has a relation of type "READABLE_BY" to a security level equal to or lower than the security level used in the query.

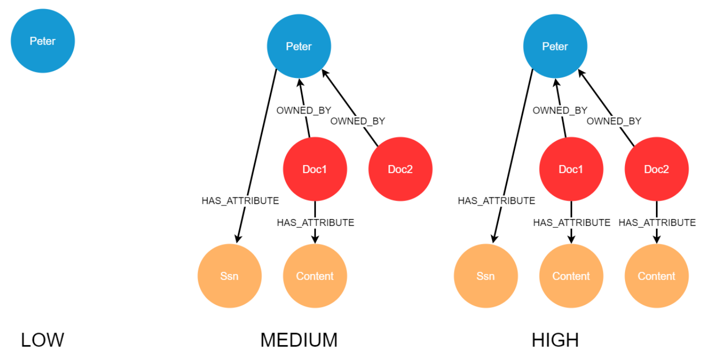

Fig. 8. Readable Nodes Depending On Security Level

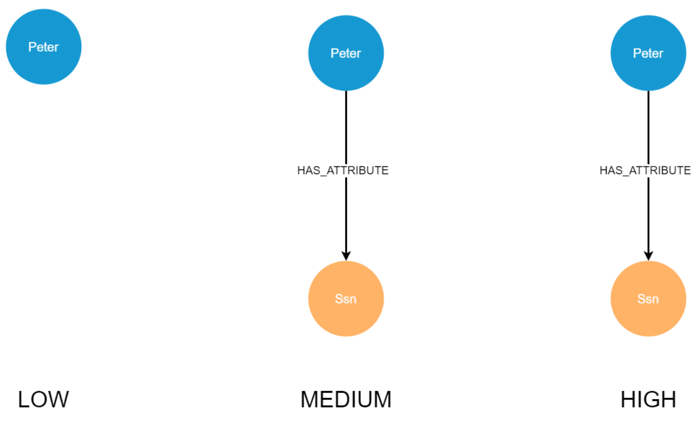

Fig. 9. Readable Attributes for Persons Depending On Security Level

B. Write Access

Determining if a user have permission to edit a node is done similar to read access with the difference of checking for the "WRITABLE_BY" relation instead. All editable nodes for every security level is returned by running the following query for each security level:

```
MATCH (n) -[:WRITABLE_BY]->
    (objsec:SecurityLevel)
    -[:IS_LOWER_THAN*0..2]->
    (subsec:SecurityLevel {name: 'low'})
    return n
```

When querying for writable nodes for low and medium security level, no nodes are returned. For high security level only the node with content for Doc1 is returned since this is the only node with a "WRITABLE_BY" relation.

C. Complex Queries

So far the implementation has only been tested for simple queries that only ask for one type of nodes but Cypher supports a lot more types of queries. If we want to match all persons along with their attribute nodes this can be done with the following query:

```
OPTIONAL MATCH (p:Person) -[:READABLE_BY]->
    (:SecurityLevel) -[:IS_LOWER_THAN*0..2]->
    (:SecurityLevel {name: 'medium'})
OPTIONAL MATCH (p) -[:HAS_ATTRIBUTE]->
    (a:Attribute)-[:READABLE_BY]->
    (:SecurityLevel) -[:IS_LOWER_THAN*0..2]->
    (:SecurityLevel {name: 'medium'})
return p, a
```

Since both Persons and Attributes have a security level we must check the "READABLE_BY" relation for both types of nodes, making the query a lot more complex.

Fig. 9 shows the results when returning all persons with their connected attributes based on security level. Everyone can see Peter while his ssn is only visible for medium and high security levels. Comparing this to Fig. 7 the query seems to work as intended.

V. Discussion

The method used in this report to create a graph database with multilevel security has been proven to work as intended in section IV. However, only basic query syntax have been used for testing this implementation and there may still be some query operations that wont work. We found that writing working queries becomes very complex as soon as more than one type of node should be included in the result since we need to check the security level for every single node. While it is still possible it is very easy to make a mistake when writing these queries which in the end may result in someone being able to exploit the database.

The current implementation assumes that users who wish to access the database do not have direct access to the database but instead accessing it through an additional application that enforces the access control policies. This means that someone must write queries for all use cases and make sure they work as intended before serving these queries to a user.

Another limitation of the proposed solution is the rapidly increasing size and complexity of the database. With the implementation of representing properties of nodes as nodes we inevitably add relations previously not represented as data. Even for a very minimal data set this is obvious, but for a larger data set, with multiple users having multiple properties attached and having finer grain access control, this could lead to a far more complex database. This size and complexity entails difficulties for both querying the database as well as the readability, but could also result in both a increase in required storage of the database and the performance of the database management system.

Our solution does only provide multilevel security for nodes and not for relationships. This means that it is not possible to hide the relationship between two nodes while still keeping both nodes visible. It would still be possible to create a workaround for this issue by adding replacing the relationship with a new node that is connected to both initial nodes. This new node can be assigned a security level which will make it possible to hide the connection of the initial nodes. This would further increase the size of the database.

VI. Related Work

Ahmadi Small [6] present work regarding the introduction of conditions predicates to an attribute based access control system in a graph database representation. Their research offers universal graph traversal algorithms for policy conditions. Crawford [7] offers a thorough introduction to granular security in graph databases. His work present a wide variety of possible implementations and discussions for introducing granular security on a graph database using Neo4j.

VII. Conclusion

The main goal with this report was to present a way to implement mandatory access control through multilevel security on a graph database. From the performed analysis of the implementation, we can conclude that we have successfully found a way to do this for the most common type of Cypher queries. From the presented result the implementation is suitable only for basic queries. The implementation lacks scalability for more complex data sets, and queries for these becomes increasingly difficult to construct. The implementation also bloats the size of the database, increasing the number of nodes and relations in the database which might have an effect on the performance of the implementation.

References

[1] Neo4j, What is a Graph Database? https://neo4j.com/developer/graph-database/ [Online; accessed 19-04-2020]. 2020.

[2] Ma, Kun Zhang, Weijuan & Tang, Zijie. (2014). Toward Fine-grained Data-level Access Control Model for Multi-tenant Applications. International Journal of Database Theory and Application. 7. 79-88. 10.14257/ijdta.2014.7.2.08.

[3] DB-Engines, DB-Engines Ranking https://db-engines.com/en/ranking [Online; accessed 19-04-2020]. 2020.

[4] Neo4j, Cypher Query Language https://neo4j.com/developer/cypher-query-language/ [Online; accessed 19-04-2020]. 2020.

[5] Red Hat Inc. Multi-Level Security (MLS). Red Hat Enterprises Linux Deployment Guide, 43.6. https://web.mit.edu/rhel-doc/5/RHEL-5-manual/DeploymentGuide-en-US/sec-mls-ov.html [Online; accessed 2020-04-16]. 2006.

[6] Hadi Ahmadi Derek Small. Graph Model Implementation of Attribute-Based Access Control Policies. Nulli-Identity managment. 2019.9.21.

[7] Brian Crawford. Granular security in a graph database. Naval postgraduate school. 2016.03.27.