Security features in NoSQL systems with a focus on graph-based NoSQL systems

Matildha Sjöstedt, Adam Andersson
matsj696@student.liu.se, adaan690@student.liu.se
TDDD17 - Information Security
Linköping University
Linköping, Sweden

***Abstract*—NoSQL DBMSs have become a more common way to store and organize data. Although, the interest in security features in these systems have not been that great and is not to well documented outside the documentation of each system. The focus in this study have been on finding security features in property graph systems and organizing these. The study ended in categorizing twelve different system in six security categories were only a few had features in all six categories. After the research the study chose two systems that seemed the most interesting to take a closer look at.**

The conclusion of the study is that there are security features in property graph systems but there are also flaws in the systems, so further development in security features could be made.

***Index Terms*—NoSQL, graph-based NoSQL, property-graphs, database security**

I. Introduction

Security features in DBMSs (Database Management Systems) is a topic that has been studied for many years. During the recent years it has become more common to use NoSQL (Not Only SQL) DBMSs as a way to create and manage databases with an alternative structure, for example graph-based. A reason for the rising popularity is that NoSQL DBMSs offer increased performance, flexibility and agility compared to systems for traditional, relational databases – such as SQL-systems. The issue here though is that control measures to provide security in these NoSQL systems are not that well catalogued and this could in turn be a problem if insecure database systems are being used to handle sensitive data.

The expected result of this project is a report that catalogues various security features and control measures regarding access control, inference control, flow control and data encryption in different types of NoSQL DBMSs, mainly focusing on Property Graph Systems. Hopefully we will find that there is some kind of security in the systems we investigate and that this could help others when choosing an effective but also secure database to use when handling sensitive data.

II. Background

This section presents the method of our study and relevant theory and definitions needed to understand the content of our report.

A. Method

To get an overview of how well-equipped different graph-based NoSQL systems are in terms of security, we will start by studying the documentation for 12 different systems that offer property graph databases. These systems are:

- Neo4J
- JanusGraph
- TigerGraph
- Oracle Spatial and Graph
- AgensGraph
- Sparksee
- Apache S2graph
- Dgraph

- Amazon Neptune
- Microsoft Cosmos
- ArangoDB
- OrientDB

While reading the documentation we will add the security features we find in a comparison table, together with a classification of the feature – such as access control or encryption – and a short description if needed. This way we will be able to see what types of features are more and less common as well as get an idea of which systems offer the most security features.

The next step is to compile the above result in some orderly way. Depending on the result this may for example be done by adding up features for each system and for each category or by creating a table where we check if a certain system has *any* features belonging to the different categories.

Finally we will select a few (two or three) of these systems to look into more closely. This means reading the documentation more carefully, possibly looking up the techniques/protocols used with the mentioned security features and also see if there have been other studies made that point out for example security flaws in these systems.

As part of the result we will also present any security features or adaptations that was made specifically for graph-based databases. This may be complemented by information from other sources, such as academic reports.

B. Theory

To know what kind of systems that are being investigated in this report we first need to give a little background about NoSQL and property graph systems. In comparison to SQL, NoSQL is different in a couple of ways:

- SQL databases are called relational databases and NoSQL databases are called non-relational databases
- NoSQL has dynamic schema so documents can be created without having a defined structure.
- NoSQL are horizontally scalable so you can handle more traffic by adding more servers to your database in comparison to SQL that is vertically scalable which means that you can increase the load by increasing hardware.
- SQL databases are table based whereas NoSQL databases are either key-value pairs, document-based, graph databases or wide column stores.
- SQL databases guarantee ACID properties and NoSQL (mostly) guarantees the BASE properties.

The above criteria are for all kinds of NoSQL databases [1], but this report have focused on property graph systems and what security aspects they possess. So what is a property graph? A property graph system is a system where the data is organized as nodes, relationships and properties. The nodes are the entities in the graph and can hold any number of attributes called properties. Nodes can be tagged with labels representing their different roles in your domain. Relationships in property graphs provide directed named connections between two nodes, for example HAS_CEO is a relationship between the nodes Company and Employee in figure 1 on the next page. Just like nodes, relationships can have properties. [2]

Now that a little theory about NoSQL and property graphs have been given, you might ask yourself why property graphs are good and why their security aspects should be investigated?

Graph databases have three key advantages over SQL databases:

- Graph databases improve performance by several orders of magnitude for intensive data relationship handling.
- In property graphs it is easy adding to existing graph structure without endangering current functionality.
- Development today is very agile and test-driven so using graph databases align perfectly with this way of working with development.

Below is a list of definitions in this report:

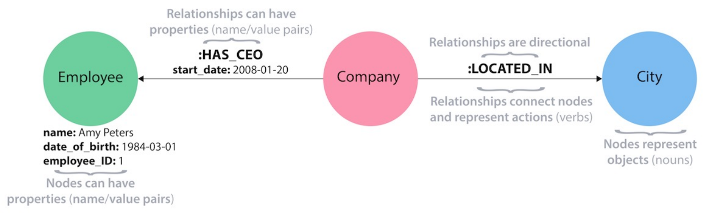

Relationships can have properties (name/value pairs)
:HAS_CEO
start_date: 2008-01-20
Employee
Company
Relationships are directional
:LOCATED_IN
Relationships connect nodes and represent actions (verbs)
City
Nodes represent objects (nouns)
name: Amy Peters
date_of_birth: 1984-03-01
employee_ID: 1
Nodes can have properties (name/value pairs)

Fig. 1. Example of a property graph [2]

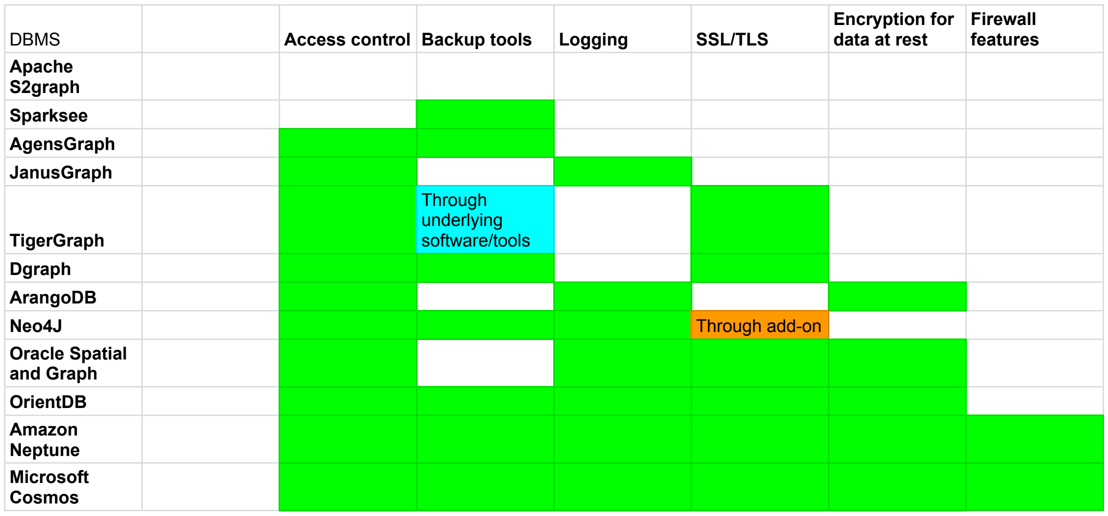
Definition	Explanation
ACID	Short for Atomicity, Consistency, Isolation and Durability
AES	Advanced Encryption Standard. A block cipher using symmetric keys. [3]
CSRF	Stands for Cross-Site Request Forgery. A type of attack which tricks/-forces/causes the victim to execute an action or submit a request unwillingly. [4]
DES	Data Encryption Standard. A block cipher using symmetric keys. DES is an older than AES, uses shorter keys and are not considered secure anymore. [5]
SSL	Secure Socket Layer. Cryptographic protocol used to secure network communications.
TLS	Transport Layer Security. SSL and TLS are often referred to synonymously, but TLS is a more updated protocol.
BASE	A model that stand for Basic Availability, Soft State and Eventual consistency.

C. *NoSQL security concerns in related work*

With the increased usage of NoSQL systems and consequently the increased number of available systems, it seems probable that the security aspect may have struggled to keep up with the development. In a report concerning the security of so called sharded NoSQL databases (sharded basically means that a large database is separated into several smaller ones) it is pointed out that the design of NoSQL-systems initially had little focus on security. They also describe the security aspect as being ”one of the most difficult challenges” for this type of systems. [6]

In an article published in 2018 by the Technical University of Sofia, a good overview of some common security issues in NoSQL systems is given. The first issue they bring up is regarding access control, which is often limited or completely missing. They also comment on bad habits of using default configurations and credentials. Some other common issues are no support of TLS/SSL (see section II-B) and sometimes no encryption for stored data. Bad query attacks – known as SQL-injection for traditional relational databases – is a well-known type of attack that also can be used against NoSQL systems. The article points out that the fact that these systems often make use of JSON or XML, may lead the execution of malicious code injected by an attacker. When it comes to NoSQL systems it is also common with self-made, custom API:s that lay on top of the database. This can enable you to add more security features yourself, but can also leave the system vulnerable to injection attacks and so called cross-site request forgery attacks (CSRF) (see section II-B). [7]

Case studies and active attacks to real NoSQL databases were made in a study against two well-known NoSQL systems, Neo4J and OrientDB. This highlighted several issues, some of them related to configurations, both bad default configurations that come with the system as well as bad use of configurations in real systems. Their study also found weaknesses regarding protection against brute-force attacks and bad query attacks. [8]



DBMS		Access control	Backup tools	Logging	SSL/TLS	Encryption for data at rest	Firewall features
Apache S2graph							
Sparksee			🟩				
AgensGraph		🟩	🟩				
JanusGraph		🟩		🟩			
TigerGraph		🟩	Through underlying software/tools		🟩		
Dgraph		🟩	🟩		🟩		
ArangoDB		🟩		🟩		🟩	
Neo4J		🟩	🟩	🟩	Through add-on		
Oracle Spatial and Graph		🟩		🟩	🟩	🟩	
OrientDB		🟩	🟩	🟩	🟩	🟩	
Amazon Neptune		🟩	🟩	🟩	🟩	🟩	🟩
Microsoft Cosmos		🟩	🟩	🟩	🟩	🟩	🟩

Fig. 2. Overview of supported security features

III. Result

In this section we present the findings of our study.

A. *Comparison of DBMSs that offer property graph databases*

After inspecting relevant documentation for all NoSQL graph DBMSs mentioned in section II-A we found slightly under 100 mentions of security features across the different systems. We could later sort them into 12 different categories. These categories include broad categories such as access control, firewall, encryption for stored data, connection encryption, fault tolerance, consistency, back-up and logging. But we also have a few more narrow categories like key management, certificate management and incident handling – for these we only recorded one or two features. At first we used the category authorization, but later decided to merge it with access control. For many of these categories we failed to take equal notice of features for all systems, making it unfair to directly compare the systems with regard to these categories. But despite this and even though the degree of delimitation for what we recorded as a ”feature” turned out to vary somewhat, it still presented us with some useful results. We got a good idea of what type of features are the most common and which systems seem to tackle most security aspects and which offer less security features or none at all.

In figure 2 we have compiled the initial result into a simple overview which shows which systems have *any* features from the six most relevant categories identified. These categories were the ones for which we studied the documentation and recorded features most equally across the systems – as discussed above. The systems in the rows of the chart are sorted based on how many categories they offer features in. The categories in the columns are sorted based on how well represented they are. We can see that we have an almost linear distribution in how many categories are represented, i.e. we see one system that is represented in no security categories and some that are represented in one, two, three etc. and finally a couple that is represented in all or all but one.

Looking at the categories, features offering some form of access control seem to be the most common among the graph systems we have looked at. This is followed by back-up and logging. The logging category includes any feature that logs for example actions or transactions made to the database. From what we could find in our inspection of the documentation only half of the systems (7/12 counting Neo4j) provide secure connections through SSL or TLS (see list of definitions in II-B). Less than half offer encryption for data at rest (data stored at disk which is not currently in transit or under operation). We could only find that two of the systems provide some kind of firewall features. Oracle Database – which offers Spatial and Graph – has an extension package ”Audit Vault and Database Firewall”, but it was difficult to find whether this works together with their graph system.

B. A closer inspection of some systems

After looking at our initial results and comparisons we have decided to look more closely at two systems: Neo4j and OrientDB. We chose these systems because they are both among the most popular graph DBMSs (according to db-engines current ranking in May 2019 [9]). Neo4j currently sits at the top of the rankings, but does – according to our findings – not offer any encryption for data at rest, firewall features and only secure connections through an add-on. Neo4j and OrientDB were also the only two systems we found that mentioned access control features clearly adapted to graph databases.

Neo4j: Neo4j is one of the bigger actors in graph databases and has customers like Microsoft, Cisco and Linkedin. The fact that they are so well known they have a good set of documentation. They claim that Neo4j, that is built from the ground up to be a graph database, is designed for storage, optimizing fast management and the traversal of nodes and relationships.

Also, join operation performance will, with the number of relationships, degrade exponentially. But in Neo4j the performance is linear since the corresponding action is performed as navigation from one node to another.[10]

The documentation of Neo4j is very extensive since it at first glance have a chapter about security which brings up things like securing extensions and SSL framework. Other than this chapter you can find chapters like Backup, that instructs on how to perform and restore backups, or Authentication and Authorization that covers role management and access control for subgraphs or property-levels.

When looking more closely at the system we find that there are two types of role management available. The first type is the normal role management where you can assign a specific role to a user and that role gives you specific privileges. These kind of roles, that are predefined in Neo4j, are reader, editor, publisher, architect and admin. The differences are the following:

- Reader: Read-only access to the data graph.
- Editor: Read/write access to the data graph. Write access are limited to creating and changing existing data in the graph.
- Publisher: Read/write access to the data graph.
- Architect: Read/write access to the data graph and set/delete access to indexes along with any other future schema constructs.
- Admin: Read/write access to the data graph and set/delete access to indexes along with any other future schema constructs. An admin also has the possibility to view/terminate queries.

So for example if you have a user that only should be able to read and write to existing property keys, node labels and relationship types you should make that user an editor. The commands to create a user and make that an editor are the following:

```
CALL dbms.security.createUser(JohnDoe, password, requirePasswordChange)
CALL dbms.security.addRoleToUser(Editor, JohnDoe
```

It is also possible for an admin to create custom roles, these are solely for executing certain custom developed procedures. These roles come in handy when you want to use the other type of role management, subgraph access control. The subgraph access control is a way to restrict a users access to specified portions of the graph. This is a very good security feature in property graphs since you can restrict a users access to certain portions of the graph depending on the labels on nodes or relationships with certain types.

An example of when you could use subpgraph access control is when you have a company and you want the accounting department to have restricted access. Then you could create a custom role called ”accounting”:
CALL dbms.security.createRole(’accounting’)
Then you assign the role to a user e.g. ”stephenmoneymaker”:
CALL dbms.security.addRoleToUser(’accounting’, ’stephenmoneymaker’)
Lastly you choose what procedures the role ”accounting” should have access to and then all users that are assigned that role gets the specified procedures:
dbms.security.procedures.roles=apoc.load.json.* :accounting
The above command gives all users with the role accounting the ability to execute procedures in the apoc.load.json namespace.

Another security aspect that Neo4j supports is sandboxing. Sandboxing means that you run your program in a separate and controlled environment, a ”sandbox”. It is a copy of the regular environment but anything that happens in the sandbox cannot affect anything on the outside. This way you can analyze when the program runs and see if there is any odd behaviour that could potentially injure the rest of your system. Neo4j uses sandboxing to ensure that procedures do not use insecure APIs since it is possible to access not publicly supported APIs when writing custom code.

Neo4j also uses white listing if you only want to use specific procedures from bigger libraries. By white listing specific procedures you will only load these ones into your system. For example if you want to use all methods in apoc.coll and no other extensions from the apoc library you could write:
dbms.security.procedures.whitelist=apoc.coll.*
This way only the methods in apoc.coll will be loaded and available to your system.

Lastly, Neo4j brings up a security checklist in the end of their Security chapter. This checklist brings up things a user of Neo4j can do to ensure an appropriate level of security for the users application. This checklist both include security aspects regarding Neo4j e.g ”Ensure the correct file permissions on the Neo4j files.” and security aspects that doesn’t only concern Neo4j e.g. ”Use subnets and firewalls.”

OrientDB: In their own documentation, OrientDB themselves claims to be giving more attention to security than any other NoSQL system, and in our comparative overview it did indeed end up second only to the systems provided by the two large companies Amazon and Microsoft. OrientDB makes use of what they refer to as ”roles”, which defines a users permissions, with regard to specific resources and so called CRUD-operations (Create, Read, Update, Delete). The permissions can be adjusted for resources at the ”schema-level” – such as classes, queries and configurations. [11] Classes in OrientDB can be thought of as classes in the context of object-oriented programming. In the documentation classes are described as being comparable to tables in relational databases, whereas the properties of the classes represent columns. In the graph-database both edges and vertices are classes, which can be inherited from in the creation of new classes. [12] Permissions can in that way be set for individual types of edges and vertices. For an existing record – edge or vertex in a graph database – permissions can also be set for individual users. [11]

Say for example that we have two nodes with an edge connecting them, the nodes are of classes student respectively course and the edge represents a student registered on a course. If Alice is a student she can be set to only have permissions to change the properties of the student node representing herself, while Bob who is a course administrator can be allowed to change all course-typed nodes and the registered-relation edges connecting to them.

The above example can be realized with the following commands to the OrientDB console:

```
1) INSERT INTO ORole SET name =
   'student_user', mode = 0
2) INSERT INTO ORole SET name =
   'course_admin', mode = 0
3) CREATE CLASS student_node
   EXTENDS V, ORestricted
4) CREATE CLASS course EXTENDS V
5) CREATE CLASS registered_on
   EXTENDS E
```

On line 1 and 2 we create two roles, `student_user` and `course_admin`. The zero after mode means that the rules are applied to the role as ”white-listing”, that is the role has no permissions from the start. On lines 3-5 we create the two node classes and the edge class. The `student_node` class extends the `Orestricted` class which allows permissions to be set for individual records of `student_node`. V and E stands for ”vertex” respectively ”edge”.

```
6) UPDATE ORole PUT rules =
   "database.class.course", 15
   WHERE name = "course_admin"
7) UPDATE ORole PUT rules =
   "database.class.registered_on",
   15 WHERE name = "course_admin"
8) UPDATE ORole PUT rules =
   "database.class.student_node",
   2 WHERE name = "course_admin"
9) UPDATE #a:b ADD _allowUpdate #c:d
```

The above commands set the permissions. Lines 6-7 gives the role `course_admin` full permissions – which is indicated by 15 (binary vector 1111) – to the classes `course` and `registered_on`. For `student_node`, `course_admin` only gets read permission, indicated by the number 2. The last line gives Alice, who has user id #c:d, permission to update the record with id #a:b, which is the student class node representing herself.

In OrientDB there also exists a ”special permission”, referred to as ”bypassRestriced” in the database configurations, which is said to bypass security restrictions – the admin user has this by default. It is not clearly expressed in the documentation exactly what restrictions can be bypassed, but as no limitations are specified it might be assumed that the ”special permissions” allows for any access and modification of the database. [11]

The documentation section for the OrientDB server security contains a couple of prompts about what the user should do to ensure security. For example to protect the server within a private network, change default passwords and to restrain access to configuration files. The server section also describes how to restore the admin user and even reset the root password. [13] A recently added feature to OrientDB is symmetric key authentication. They seem to support AES (see section II-B) with 128-bit keys, based on the given example in the documentation. As far as we could find the documentation does not explain or present any support on how to exchange these keys. [14] The connections – through HTTP and OrientDB's binary protocol – to the OrientDB server can be secured by TLS (see section II-B). This does not however seem to be enabled by default. The connection encryption feature is referred to as SSL in the configurations and the documentation, but they emphasize that TLS is what is actually used. The system uses the Java Keytool to manage certificates. [15] Java Keytool can be used to generate public and private keys and import/export certificates from the Java KeyStore. [16]

OrientDB supports encryption for stored data. They offer two algorithms – AES and DES (see section II-B). The encryption feature does not yet work through access on ”remote protocol”. Individual clusters – groups of records – can be encrypted separately. As with the OrientDB server, the documentation for stored data encryption does not mention any support or features concerning the management or generation of encryption keys – other than that the key is *not* stored in the database and if the user loses the key they will lose the encrypted content. [17]

IV. Analysis

In this section we discuss and analyze the results and the method that has been used in the study. There will also be further comparisons between our findings and the security concerns brought up in the background section.

A. *Result evaluation*

Out ff the twelve DBMSs with graph-database support which we investigated, ten of them provide some sort of access control. Before the study of these particular systems, but after some initial research on the topic of NoSQL security, we did not expect such a high number. In the report [7] (as mentioned in section II-C), it is described that access control is often limited or missing. Considering this, it should be noted that our comparative chart does not provide any qualitative comparisons. An inadequate access control scheme that can easily be bypassed or made useless with for example exposure of passwords on unencrypted channels, might be more dangerous to the user than a system with no access control at all. The risk being that a less security-aware user might fall under a false sense of security and not being concerned about adding sensitive data into the database.

In contrast to the support of access control, we see that secure connections and data at rest encryption are a lot less frequently supported. These are also categories of features mentioned in [7]. As mentioned above insecure connections can lead to a lot of problems, like man-in-the-middle attacks which might result in unauthorized access to data or modifications of for example queries or operations made by a user which are transmitted over an insecure channel. Encryption for data at rest can sometimes be one of the last lines of defense against hackers obtaining sensitive information. If such a feature is missing, more effort will likely have to be made in different security areas, to reducing the risk of a hacker gaining access to plain data from the database. Physical thefts of storage disks are another aspect to consider when this type of encryption is not used.

The two systems which we decided to look more closely on – Neo4J and OrientDB – both showed some types of security flaws when investigated in [8] (see section II-C). For OrientDB one of the issues mentioned was problems with default configurations. In the closer inspection of this system we did indeed find that OrientDB for example seems to not have TLS (see section II-B) enabled by default and there exists default users with default passwords that also need to be changed before one can safely use the database. Another issue which was mentioned in [8] was vulnerabilities to bad-query attacks. Not in the documentation for any system could we find any concrete mentions of countermeasures against such attacks. A flaw Neo4J is that they do not offer encryption for data at rest. Although in the checklist to ensure appropriate level of security that they provide in their chapter called Security they are mentioning that you should protect data-at-rest. In the checklist they say that you should use volume encryption (e.g. Bitlocker) and manage your access to database dumps and backups. Lastly they mention that you should ensure correct file permissions on your Neo4J files. This is the information they give you to protect data-at-rest, so they know what has to be done but has not implemented their own software to do so.

If we look at the different kind of security features the DBMSs provided, there were sometimes features that were specific for one or only a few of the DBMSs. Each of these features were then classified in one of the categories mentioned in the section III to give a view on what security categories each DBMS covers. Considering that a DBMS might have some of these exclusive features while they may lacked some features supported in most other DBMSs made it hard to make any analysis on which DBMS that is the most secure. One example of a specific feature is Sandboxing in the DBMS Neo4j, a feature that allows you to test procedures so they do not use insecure APIs. This feature has not been found in any of the other DBMSs but is a very useful security feature for Neo4j.

B. Method evaluation

The biggest concern we experienced with the method of our study was that we failed to investigate available documentation and record features in an equal manner for all systems. This resulted in several categories of features which we could not compare directly, seeing as for some systems we had not had these categories in mind when reading through the documentation. In the end time prevented us from going through the documentation again for more detailed recording of features. An obvious improvement for a similar study would thus be to clearly define a set of categories beforehand and also set up some definition on what can be considered a single feature. With more equal and fair recording for each systems more comparisons might have been possibly. For further investigation considerations should also be taken to the qualitative properties of each security feature.

V. Conclusions

In this section we summarize the most important findings in our result together with some further reflections from our part.

The findings of our study shows that the amount of provided security for the twelve investigated graph-based DBMSs (see section II-A) varies a lot. We have a few systems that provides features for all or close to all the categories covered in figure 2, whereas some provide none or only a few. We can see that the systems with features in all categories are systems developed by large companies with a well-established IT-infrastructure in general. The fact that these companies have more resources to provide DBMSs with more security features than for example free open-source projects or newer, less-established companies are quite natural. This also means that there exists options for users with varying needs, both with regard to the amount of security needed and what needs and abilities they have to develop a custom client or API for the database to cover up for the lack of native security features.

We could only find that two of the systems we looked at provided access control adapted to graph databases, these were Neo4J and OrientDB. Here the a user's permissions and access can be configured on the subgraph level – smaller parts of the graph or for single nodes or edges. For OrientDB nodes and edges can also be further divided into different classes, resulting in a very flexible way of setting permissions for each user. But there also seems to be some cons and possible vulnerabilities for these to systems, as presented by [8] in section II-C and discussed in IV-A.

Seeing that the study started with finding security features in the documentation of the DBMSs and then categorizing these into six different categories gives us no result in what DBMS that is the most secure other than if they have covered the six categories that we made our priority. This means that a database that only covers three of the six categories could have more security features than one that covers all of the categories. Although finding the most secure property graph system was not the aim of this study – but rather trying to find any security features and then categorizing them – this could be an interesting subject for a follow-up study.

References

[1] ”Difference between SQL and NoSQL”, GeeksforGeeks, https://www.geeksforgeeks.org/difference-between-sql-and-nosql/ accessed: 2019-05-06.

[2] ”What is a Graph Database?”, Neo4J, https://neo4j.com/developer/graph-database/, accessed: 2019-05-06.

[3] ”Advanced Encryption Standard (AES)”, SearchSecurity, updated: March 2017, https://searchsecurity.techtarget.com/definition/Advanced-Encryption-Standard, accessed: 2019-05-09.

[4] ”Cross-Site Request Forgery (CSRF)”, OWASP, updated: 2018, https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF), accessed: 2019-05-10.

[5] ”WHAT ARE THE DIFFERENCES BETWEEN DES AND AES ENCRYPTION?”, Townsend security, 2014, https://info.townsendsecurity.com/bid/72450/what-are-the-differences-between-des-and-aes-encryption, accessed: 2019-05-14.

[6] ”Security of Sharded NoSQL Databases: A Comparative Analysis”, Anam Zahid et. al., 2014, Conference on Information Assurance and Cyber Security (CIACS).

[7] ”Overcoming the security issues of NoSQL databases”, Tony Karavasilev and Elena Somova, 2018, Technical University - Sofia, Plovdiv branch.

[8] ”(In) Security in Graph Databases Analysis and Data Leaks”, Miguel Hernández Boza and Alfonso Muñoz Muñoz, 2017, Proceedings of the 14th International Joint Conference on e-Business and Telecommunications.

[9] ”DB-Engines Ranking of Graph DBMS”, Db-engines, https://db-engines.com/en/ranking/graph+dbms, accessed: 2019-05-06.

[10] ”The Neo4j Operations Manual v3.5”, p.2, Neo4j, https://neo4j.com/docs/pdf/neo4j-operations-manual-3.5.pdf, accessed: 2019-05-06.

[11] ”Database Security”, OrientDB documentation for ver. 2.2.x, https://orientdb.com/docs/2.2.x/Database-Security.html, accessed: 2019-05-06.

[12] ”Classes”, OrientDB documentation for ver. 2.2.x, http://orientdb.com/docs/last/Tutorial-Classes.html, accessed: 2019-05-14.

[13] ”Server Security”, OrientDB documentation for ver. 2.2.x, https://orientdb.com/docs/2.2.x/Server-Security.html, accessed: 2019-05-06.

[14] ”Symmetric Key Authentication”, OrientDB documentation for ver. 2.2.x, https://orientdb.com/docs/2.2.x/Security-Symmetric-Key-Authentication.html, accessed: 2019-05-06.

[15] ”SSL”, OrientDB documentation for ver. 2.2.x, https://orientdb.com/docs/last/Using-SSL-with-OrientDB.html, accessed: 2019-05-14.

[16] ”Java Keytool”, Jenkov.com - Tech and Media Labs, http://tutorials.jenkov.com/java-cryptography/keytool.html, updated: 2018-01-18, accessed: 2019-05-14.

[17] ”Database Encryption”, OrientDB documentation for ver. 2.2.x, https://orientdb.com/docs/last/Database-Encryption.html, accessed: 2019-05-14.