Management of sensitive data on NoSQL databases

Carlos M. García-Ruiz[1], Alejandro Oliver[1], Jesús Peral[1], Juan Trujillo[1], Carlos Blanco[2], Eduardo Fernández-Medina[3]

[1] Lucentia Research Group, Alicante University, San Vicente del Raspeig – Alicante, Spain.
[2] University of Cantabria. Spain
[3] Alarcos Research Group. University of Castilla La Mancha. Spain
{cmgr4,aor14}@alu.ua.es, {jperal,jtrujillo}@dlsi.ua.es
cblanco@ucan.es, eduardo.fernandez@uclm.es

Abstract. Nowadays, NoSQL databases are winning popularity through all the e-commerce related sites and it is becoming a tendency to migrate from a relational model to others without schema. NoSQL databases are more flexible since they allow an easier adaptation of the database structure and the capability of refactoring the schema on the fly for any project without having to stop the database service and evolving the database schema. Finally, the query optimization is designed to manage large amounts of data, whereas on relational databases it is more focused on operations rather than on data. In this paper, we intend to make a demonstration on how security can be implemented on the database layer focusing on the permissions of users, this could be applied to databases where some fields can contain sensitive data that certain users should not have access to. We also test the capability of trimming determined fields of a database to have a more secure environment. For this implementation, we have chosen MongoDB for its schema less property and the lack of internal mechanisms to automatically apply Access Control Rules. In this paper, the user restrictions are implemented using a Role-Based Access Control model on MongoDB. This means that a user can be granted one or more roles previously defined, where each role would have authorizations defined to make operations in specific databases, collections or documents.

1 Introduction

MongoDB [1] is an open-source multiplatform NoSQL database system. It is a document-oriented database with no predefined schema that implies every registry can have a different structure, e.g. a structure called Address could contain a plain-text field referring to the address and in some instances, it could or could not have a number.

This technology pretends to increase horizontal scalability of the data layer with a simple development complexity and the possibility of storing enormous amounts of data. When the necessity of scaling is given through several machines, NoSQL results very impressive on performance, and MongoDB is one of the NoSQL databases with highest performance rates. The BSON/JSON document model of MongoDB intends to be easy to develop and manage, and it offers high performance with the inner aggrupation of relevant data [1-3].

Every instance or combination of data is named by Document, and documents can be grouped into Collections (equivalent to tables in relational databases) with no predefined schema. Some fields can be indexed in documents for higher performance.

The main hierarchy of MongoDB is given by the previously presented elements. The main difference between MongoDB and relational database tools is that the concepts of tables, rows and columns do not exist, instead documents are used with different structures. An aggrupation of fields would form a document and in case this document is grouped with other documents it would result in a collection. The databases will be composed by collections. Finally, in a real scenario, a server can host many databases.

As mentioned before, MongoDB does not have a standard schema of data, but this does not mean that the data would be inconsistent. In fact, structured documents are very common in MongoDB, so it will not be difficult to manage all the data. For data transfer of documents in MongoDB, the BSON (Binary JavaScript Object Notation) is used. It consists of a binary representation of data structures, designed to be lighter and more efficient than JSON (JavaScript Object Notation). MongoDB has a series of tools that allow a very intuitive interaction with the databases, between the most popular are:

- Mongod: MongoDB database server.
- Mongo: Client of MongoDB that allows direct interaction with the databases.
- Mongofiles: tool for working with files directly over the databases.

Regarding security aspects and accessing sensitive data, MongoDB (and other schema less NoSQL databases) presents the serious problem that it is difficult to automatically or semi-automatically control the access to sensitive data due to the fact that several different structures of documents can coexist in the same database.

Therefore, in this paper, we propose the basis for semi-automatically implementing security rules on MongoDB (and other NoSQL databases in the future). In concrete, we have defined a security approach focused on NoSQL document databases. Our approach allows us the specification of both structural and security aspects related to document databases. Therefore, new collections of documents can easily be added and the new security and access-control rules can be more easily defined. In order to check the applicability of our proposal, we use it in to generate the required code to implement the defined security rules on MongoDB.

The rest of the paper is structured as follows. Section 2 summarizes the most relevant concepts and challenges on NoSQL databases security. Section 3 presents our proposal to establish security constraints on: collection, field and field content. Subsequently, Section 4 shows a case study with a dataset about patient clinical reports. Finally, the main contributions and our directions for future work are explained in Section 5.

2 Related Work

2.1 NoSQL databases security

NoSQL databases can be classified in four different categories depending on the type of model selected to store the data:

- Key/Value: data is stored and accessible by a unique key that references a value, e.g. DynamoDB, Riak, Redis. “Amazon” and “Best Buy”, among others, use this kind of implementation.
- Columns: similar to the key/value model, but the key consists of a combination of column, row and a trace of time used to reference groups of columns (called families). This model is the most related to relational databases due to its similar implementation, e.g. Cassandra, BigTable, Hadoop/HBase. Companies like “Twitter” or “Adobe” use this model of database.
- Documents: Data is stored in documents that encapsulate all the information following a standard format (XML, YAML, JSON, etc.). Documents have field names performing as key values and the content of the field is considered the value associated to its key. This is a more complex implementation of the key/value model, e.g. MongoDB, CouchDB. As case uses of this technology we can find “Netflix”.
- Graphs: This applies the graph theory expanding between multiple computers. This model is appropriated for structures of data that have a pattern similar to a graph. For example, transportation networks, maps, etc. E.g. Neo4J, GraphBase.

With regard to the incorporation of security policies in these NoSQL databases (used in Big Data technologies) several work have been defined. However, they usually do not consider security at the modelling stages [4-6].

Other contributions present a complete secure development of information systems. Although they do not focus specifically on NoSQL databases and their specific security problems, they present interesting ideas: (1) Secure TROPOS [7] is an extension that includes security in the TROPOS methodology for software development based on the intentional goals of agents. (2) Mokum [8], which is an active object-oriented knowledge-based system for modelling, allows the specification of security and integrity constraints. (3) UMLsec [9] uses formal semantics in order to evaluate security specifications; it defines the specification of confidentiality, the integrity requirements and the accessing control policies. (4) The application of the model-driven approach to include security properties in high-level system models and the automatic generation of secure systems are carried out in MDS (Model-driven Security) [10].

2.2 Security challenges in NoSQL databases

Given the wide variety of NoSQL databases, it is almost compulsory to consider the generic weaknesses of these data models, and for each particular case apply the convenient strategies. Compared to relational databases we can find the following security measures to consider:

- **Authentication:** The main weakness of NoSQL databases is authentication because they usually come with default credentials or with no required authentication at all (e.g. Redis). In many cases the databases are supposed to be running in trusted environments, due to this supposition many implementations are in serious danger.
- **Data integrity:** NoSQL's philosophy is based on high availability and performance. Because of this, data integrity is usually penalized. Therefore, it is necessary the usage of external mechanisms in order to ensure integrity.
- **Confidentiality and storage encryption:** Generally, data storage is placed in plain text except certain cases like Cassandra and its technology "Transparent Data Encryption". In most of the cases is inevitable to delegate the data encryption to the application layer or to the file system used.
- **Data auditing:** Most of the NoSQL database technologies lack their own auditing system, therefore they are exposed to attacks due to the incapability of monitoring concrete events over registries.
- **Data exchange security:** Encrypted communications and the use of the SSL protocol is usual in relational databases. However, in NoSQL systems it is generally disabled by default and it is optional (Cassandra), or a complex configuration is required (MongoDB).
- **Classic vulnerabilities in databases: even more injection:** Finally, the most exploded aspect of NoSQL databases is the command injection, requests are executed through an API formatted according to a convention, usually JSON or XML. At this point, an incorrect verification of the parameters can lead to the execution of undesired commands. The possibilities of injection and its risks when using an API are even stronger when using a procedural programming language, whereas in relational databases the risk is high because of the SQL language, but not as dangerous as in NoSQL databases.

Concluding, NoSQL is growing higher and higher in popularity between innovative technologies and is inevitable to expect an increase of resources in security is going to be used in all kind of production environments. Furthermore, after analyzing the previous work, it is necessary to define a complete secure approach focused on NoSQL databases.

3 Our proposal

One of the contributions of this proposal is the establishment of the security privileges needed to access each field of the data set. It is carried out by using Natural Language Processing (NLP) and lexical ontological resources. We have used the lexical resource WordNet[1]. By analyzing the values of each field we establish tree kinds of security constraints.

[1] http://wordnetweb.princeton.edu/perl/webwn (visited on February, 2017).

0. Security collection. All users are not allow to access all site collections, a given level of security is required in order to limit the collections users can access.

1. Security constraints. There are fields in which all the information is sensitive at the same security level, that is, it does not depend on their specific values.

For instance, the information of the address field is sensitive and a certain security level (for instance, SL = 1) could be required for queries. This level is the same for all the values of this field, that is, there are not instances of address more sensitive than others.

2. Fine-grain security constraints. Nevertheless, there are special cases in which it is necessary to define fine-grain security constraints to establish higher security privileges for certain values of the field. For instance, to query a field representing diseases could require a specific security level in general (for instance, SL = 1), but certain values that represent terminal diseases could require a higher security level (for instance, SL = 2). In this way, a user with security level of 1 could only see diseases not related to terminal diseases.

Once the security constraints have been established, the designer models the data set according to our security approach. In this paper we have defined an approach focused on a kind of NoSQL databases, document databases. It allows the specification of both structural and security aspects related with document databases. It permits modeling structural aspects such as Databases (as Packages), Collections (as Classes) and Fields (as Properties).

The security configuration of the system which we want to model is defined by using three points of view: a hierarchical structure of Security Roles; a list of Security Levels with the clearance levels of the users; and a set of horizontal Security Compartments or groups. We can define security rules associated with structural elements. Each rule indicates the actions that certain subjects can carry out over certain objects. Furthermore, we can define fine-grain security rules which affect specific fields of a collection. This kind of rules allows us to establish different security privileges when the values of a field satisfy a condition.

4 Case study

4.1 Description

The dataset used for the demonstration is a custom adaptation from the UCI Machine Learning Repository that represents the patient clinical reports of 130 hospitals between the years 1999 and 2008 [11].

4.2 Data-set definition

With MongoDB started as a service, the creation of the test database “Hospital” is very simple, we just use the command “use Hospital”. For the example, we will work with the “Patient” and “Admission” collections. In Fig. 1 we can observe the document validation restriction implemented in the “Patient” collection.

```
db.createCollection("Patient",
{ validator: { $and:
[
        {$or: [
        {name: {$type: "string"}},
        {name: {$exists: false}}
        ]},
        {$or: [
        {race: {$type: "string"}},
        {race: {$exists: false}}
        ]},
        {$or: [
        {gender: {$type: "string"}},
        {gender: {$exists: false}}
        ]},
        {$or: [
        {age: {$type: "number"}},
        {age: {$exists: false}}
        ]},
        {$or: [
        {address: {$type: "string"}},
        {address: {$exists: false}}
        ]},
]}});
```

Fig. 1. Patient collection structure.

In both collections, pre-loaded data can be found. The documents contain data of different patients with their admissions. Fig. 2 shows the insertion of the test documents in the “Patient” collection, while in the “Admission” collection it has been used a reference technique insertion.

```
db.Patient.insert([{name: "John", race: "Caucasian", gender: "Male", age: 34, address:
"Main Street 1"},
                {name: "Mila", race: "African", gender: "Female", age: 21, address:
"Main Street 2"},
                {name: "Charles", race: "Caucasian", gender: "Male", age: 98, address:
"Main Street 23"}]);
```

Fig. 2. Example documents insertion into the Patient collection

4.3 Restriction model implementation

Once the database and its collections are created, we will restrict collections user can access and set the restriction policies to the fields. Thanks to MongoDB native views system, we are able to set restrictions to control which kind of information a user can access, and also thanks to the aggregation pipelines, we can determine if a field contains sensitive data and trim every result of a concrete field.

In order to create the view we will use the following syntax: “db.createView(viewName, collectionName, pipeline, options)”. In Fig. 3 we observe the creation of a view with a security level 2, where the field “medical_specialty” is restricted if the data on this field is “Oncology” as we considered it restricted data for this level of access to the database.

```
db.createView("Admission2", "Admission",
              [{$project: {"patient_id": 1, "_id": 0, "date": 1, "type": 1,
"source": 1,"time_in_hospital": 1,
"medical_specialty": {$cond: {if: {$eq: ["$medical_specialty","oncology"]},then:
"",else: "$medical_specialty"}},"treatment.medicament": 1, "treatment.dose": 1 }}]);
```

Fig. 3. View creation for level 2 restriction access.

The next step is to implement the creation of the role that will have access to this View. For this we will use the command: “db.createRole(roleName, privileges, otherRoles)”. In Fig. 4 it is shown that the role will only be applied to security level 2. The “Collection: Admission2” is the view defined with restrictions and as we can see we treat it as if it was a collection of the database.

```
db.createRole({
        role: "2",
        privileges: [
        { resource: { db: "Hospital", collection: "Patient"},
        actions: ["find","update","insert"]},
        { resource: { db: "Hospital", collection: "Admission2"},
        actions: ["find"]}],
        roles: []
});
```

Fig. 4. Role creation for access level security 2.

Finally, the users are created, this way the environment will be entirely prepared for testing. In Fig. 5 we use the “createUser” command to define a user that would be capable of applying the security role 2 over the “Hospital” database.

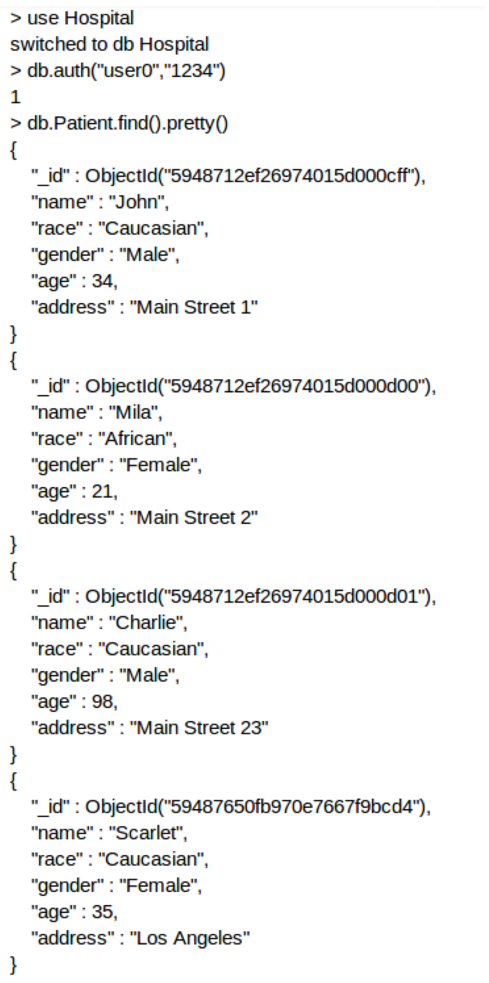
```
db.createUser({
   user: "user2",
     pwd: "1234",
  roles: [ { role: "2", db: "Hospital" }]
});
```

Fig. 5. User creation with level 2 security.

In order to manage the access of users to the database and collections it is mandatory to include in the configuration file “mongod.conf”, inside “#security” the following sentence: “security.authorization : enabled”. The file is located in “/etc/mongod.conf”, accessible via terminal.

4.4 Environment restrictions demonstration

In this section, we will prove that the restrictions defined on the database are completely functional. The first step is to login as a user, with the previously defined user, and since we activated the authentication protocol on the service mongod, now it is compulsory to log in as a user to access the databases and collections.



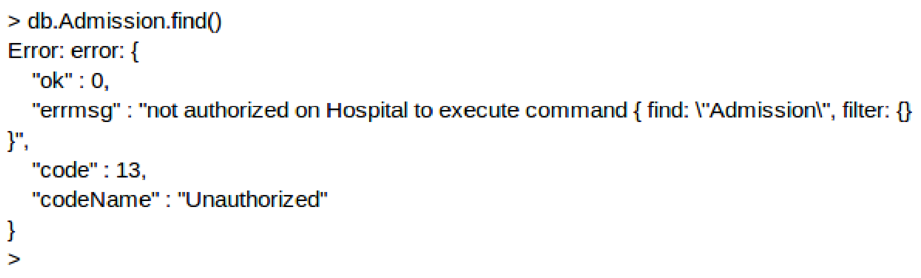
```
> use Hospital
switched to db Hospital
> db.auth("user0","1234")
1
> db.Patient.find().pretty()
{
   "_id" : ObjectId("5948712ef26974015d000cff"),
   "name" : "John",
   "race" : "Caucasian",
   "gender" : "Male",
   "age" : 34,
   "address" : "Main Street 1"
}
{
   "_id" : ObjectId("5948712ef26974015d000d00"),
   "name" : "Mila",
   "race" : "African",
   "gender" : "Female",
   "age" : 21,
   "address" : "Main Street 2"
}
{
   "_id" : ObjectId("5948712ef26974015d000d01"),
   "name" : "Charlie",
   "race" : "Caucasian",
   "gender" : "Male",
   "age" : 98,
   "address" : "Main Street 23"
}
{
   "_id" : ObjectId("59487650fb970e7667f9bcd4"),
   "name" : "Scarlet",
   "race" : "Caucasian",
   "gender" : "Female",
   "age" : 35,
   "address" : "Los Angeles"
}
```

Fig. 6. Level 0, security collection.

For the demonstration, we created users with restriction levels 1, 2 and 3.

1. **Level 0, security collection**: At level 0 a user has access to certain collections, in this case to the "Patient" collection (Fig. 6).

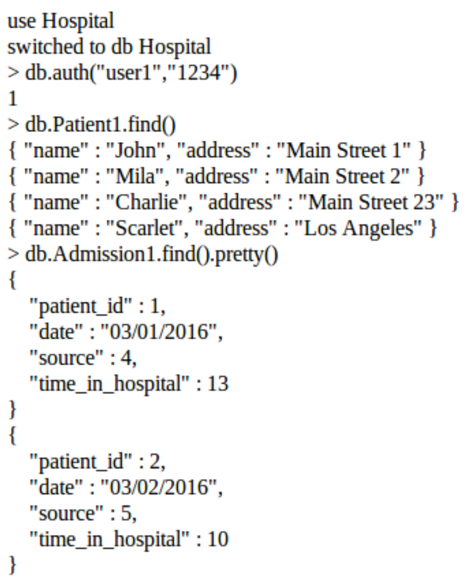
In case someone tries to access to the collection "Admission" we would receive a message on the screen which informs that required permissions to access to the already stated collection are not owned (Fig. 7).



```
> db.Admission.find()
Error: error: {
   "ok" : 0,
   "errmsg" : "not authorized on Hospital to execute command { find: \"Admission\", filter: {}
}",
   "code" : 13,
   "codeName" : "Unauthorized"
}
>
```

Fig. 7. Level 0, security collection(2).

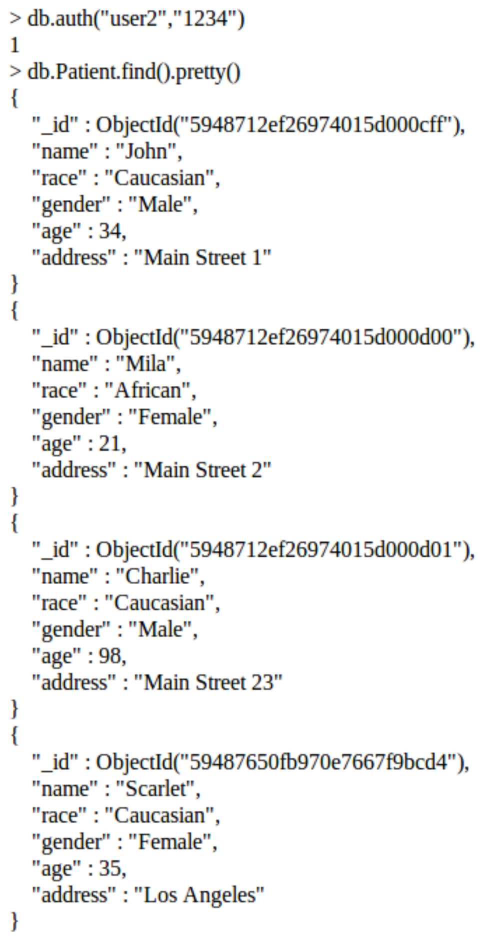
2. **Level 1, security constraints**: The level 1 can only access the defined views of "Patient" and "Admission" since it is the lowest permission level allowed for the database. Fig. 8 shows how the system will not authorize to enter any database if the user does not have permission and the view will only show the fields to what the user has permission.



```
use Hospital
switched to db Hospital
> db.auth("user1","1234")
1
> db.Patient1.find()
{ "name" : "John", "address" : "Main Street 1" }
{ "name" : "Mila", "address" : "Main Street 2" }
{ "name" : "Charlie", "address" : "Main Street 23" }
{ "name" : "Scarlet", "address" : "Los Angeles" }
> db.Admission1.find().pretty()
{
   "patient_id" : 1,
   "date" : "03/01/2016",
   "source" : 4,
   "time_in_hospital" : 13
}
{
   "patient_id" : 2,
   "date" : "03/02/2016",
   "source" : 5,
   "time_in_hospital" : 10
}
```

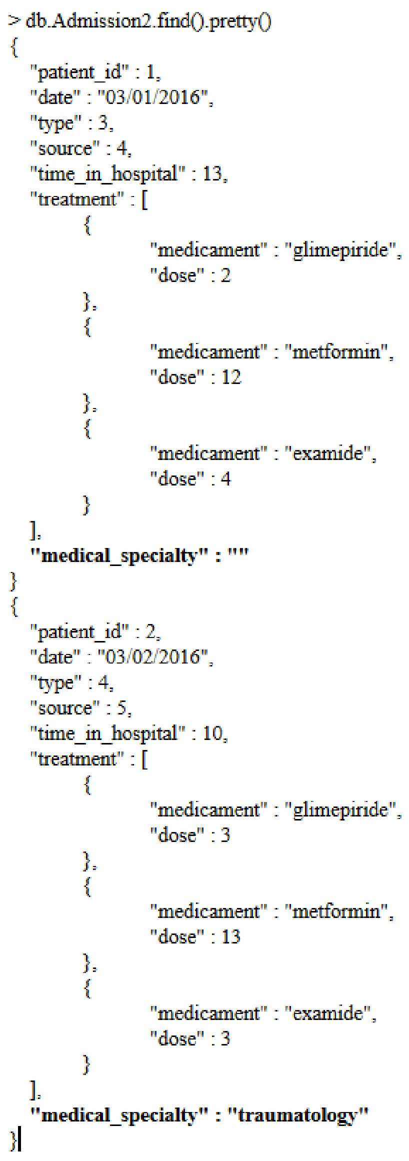
Fig. 8. Level 1, security constraints.

3. **Level 2, fine-grain security constraints**: The level 2 user is defined to be able to access the collection “Patient”, but with restrictions on the “Admission” collection. Fig. 9 shows the user has total access to “Patient” and it does not show specific fields on “Admission”. Furthermore, on the “Patient” collection with this level of security, if the data in the field “medical_specialty” of “Admission” is “oncology” the data will be treated as sensitive and it will not be shown in any result of queries made by this user (Fig. 10).



```
> db.auth("user2","1234")
1
> db.Patient.find().pretty()
{
   "_id" : ObjectId("5948712ef26974015d000cff"),
   "name" : "John",
   "race" : "Caucasian",
   "gender" : "Male",
   "age" : 34,
   "address" : "Main Street 1"
}
{
   "_id" : ObjectId("5948712ef26974015d000d00"),
   "name" : "Mila",
   "race" : "African",
   "gender" : "Female",
   "age" : 21,
   "address" : "Main Street 2"
}
{
   "_id" : ObjectId("5948712ef26974015d000d01"),
   "name" : "Charlie",
   "race" : "Caucasian",
   "gender" : "Male",
   "age" : 98,
   "address" : "Main Street 23"
}
{
   "_id" : ObjectId("59487650fb970e7667f9bcd4"),
   "name" : "Scarlet",
   "race" : "Caucasian",
   "gender" : "Female",
   "age" : 35,
   "address" : "Los Angeles"
}
```

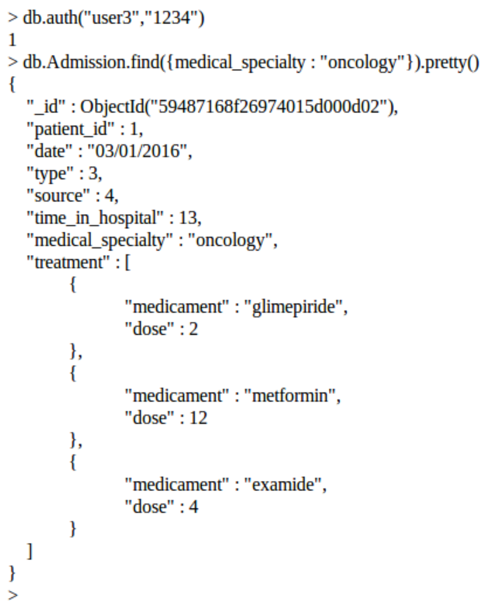
Fig. 9. Level 2, fine-grain security constraints.



```
> db.Admission2.find().pretty()
{
   "patient_id" : 1,
   "date" : "03/01/2016",
   "type" : 3,
   "source" : 4,
   "time_in_hospital" : 13,
   "treatment" : [
           {
                   "medicament" : "glimepiride",
                   "dose" : 2
           },
           {
                   "medicament" : "metformin",
                   "dose" : 12
           },
           {
                   "medicament" : "examide",
                   "dose" : 4
           }
   ],
   "medical_specialty" : ""
}
{
   "patient_id" : 2,
   "date" : "03/02/2016",
   "type" : 4,
   "source" : 5,
   "time_in_hospital" : 10,
   "treatment" : [
           {
                   "medicament" : "glimepiride",
                   "dose" : 3
           },
           {
                   "medicament" : "metformin",
                   "dose" : 13
           },
           {
                   "medicament" : "examide",
                   "dose" : 3
           }
   ],
   "medical_specialty" : "traumatology"
}
```

Fig. 10. Level 2, fine-grain security constraints (2).

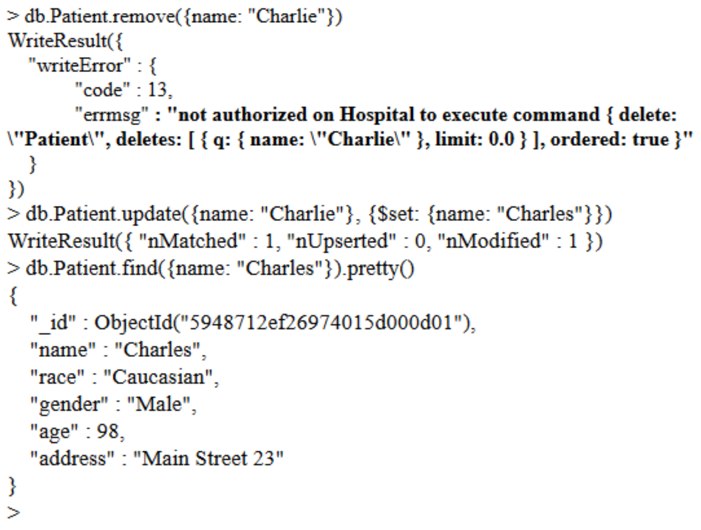
4. **Level 3**: Finally, the user with access level 3 will be able to see all the fields and all the data. As shown in Fig. 11, the result query displays all the available fields and data in both "Patient" and "Admission" collections.



```
> db.auth("user3","1234")
1
> db.Admission.find({medical_specialty : "oncology"}).pretty()
{
    "_id" : ObjectId("59487168f26974015d000d02"),
    "patient_id" : 1,
    "date" : "03/01/2016",
    "type" : 3,
    "source" : 4,
    "time_in_hospital" : 13,
    "medical_specialty" : "oncology",
    "treatment" : [
            {
                    "medicament" : "glimepiride",
                    "dose" : 2
            },
            {
                    "medicament" : "metformin",
                    "dose" : 12
            },
            {
                    "medicament" : "examide",
                    "dose" : 4
            }
    ]
}
>
```

Fig. 11. Level 3 restriction demonstration.

Despite the level 3 is the highest security level, for the sake of investigation it has a restriction on the CRUD (Create, Read, Update, and Delete) operations over the database, it will not be able to remove any data from the collections, this way we can demonstrate that the restrictions concerning commands can also be defined. Fig. 12 shows an attempt to execute an unauthorized command from this user level of security running into an error.



```
> db.Patient.remove({name: "Charlie"})
WriteResult({
    "writeError" : {
            "code" : 13,
            "errmsg" : "not authorized on Hospital to execute command { delete:
\"Patient\", deletes: [ { q: { name: \"Charlie\" }, limit: 0.0 } ], ordered: true }"
    }
})
> db.Patient.update({name: "Charlie"}, {$set: {name: "Charles"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.Patient.find({name: "Charles"}).pretty()
{
    "_id" : ObjectId("5948712ef26974015d000d01"),
    "name" : "Charles",
    "race" : "Caucasian",
    "gender" : "Male",
    "age" : 98,
    "address" : "Main Street 23"
}
>
```

Fig. 12. Level 3 unauthorized execution example.

5 Conclusion

In this paper, we have made a demonstration on how security can be implemented on the database layer focusing on the permissions of users. We have applied our proposal to a real case study where some database fields can contain sensitive data that certain users should not have access to. The results of implementing restriction security on a clinical database can be concluded into a success.

The first reason to consider it an achievement is that we were able to implement user security to the database level, when the majority of production implementations use higher layers such as the application layer to implement the user security restrictions.

The second one is that the usage of Views is interesting since they are natively restricted to make read-only operations. This could be very useful when we want to give access to data to our clients and for example, in this case study, using views, a patient that tried to access his data could only be able to see the data concerning the patient, with no additional data of the hospital management or the doctor's management.

Due to the success of this experiment, our intention is to make a wider look into NoSQL security issues and try to implement the described security model in a more complex environment with MongoDB, and once we observe that it is capable of managing the security restrictions, our target would be to emulate the security environment in other NoSQL database technologies based in different structures, such as graph databases.

6 Acknowledgements

This work is part of the Final Degree Project in the Computer Science Degree, presented by Carlos M. García-Ruiz, Alejandro Oliver, and Jorge Espinosa in June 2017, and adviced by Jesús Peral, Juan Trujillo, Eduardo Fernández, and Carlos Blanco. This work is partially funded by the funded projects SEQUOIA-UA (TIN2015-63502-C3-3-R) and the SEQUOIA-UCLM (TIN2015-63502-C3-1-R) from the MINECO.

7 References

1. https://docs.mongodb.com/manual/core/views/#create-view
2. https://blog.pandorafms.org/nosql-vs-sql-key-differences/
3. https://docs.mongodb.com/manual/reference/method/db.createRole/
4. N. Kshetri. Big data's impact on privacy, security and consumer welfare. Telecommunications Policy, 38(11):1134-1145, 2014.
5. K. Michael and K. Miller. Big data: New opportunities and new challenges [guest editors' introduction]. Computer, 46(6):22-24, 2013.
6. R. Toshniwal, K.G. Dastidar, and A. Nath. Big data security issues and challenges. International Journal of Innovative Research in Advanced Engineering (IJIRAE), 2(2):15-20, 2015.
7. L. Compagna, P.E. Khoury, A. Krausová, F. Massacci, and N. Zannone. How to integrate legal requirements into a requirements engineering methodology for the development of security and privacy patterns. Artificial Intelligence and Law, 17(1):1-30, 2009.

8. R.P. van de Riet. Twenty-five years of mokum: For 25 years of data and knowledge engineering: Correctness by design in relation to mde and correct protocols in cyberspace. Data & Knowledge Engineering, 67(2):293-329, 2008.
9. J. Jurjens and H. Schmidt. Umlsec4uml2 - adopting umlsec to support uml2. Technical report, Technical Reports in Computer Science. Technische Universitat Dortmund, http://hdl.handle.net/2003/27602, 2011.
10. D. Basin, J. Doser, and T. Lodderstedt. Model driven security: from uml models to access control infrastructures. ACM Transactions on Software Engineering and Methodology, 15(1):39-91, 2006.
11. A. Frank and A. Asuncion. UCI machine learning repository [http://archive. ics. uci. edu/ml]. Irvine, ca: University of California. School of Information and Computer Science, 213, 2010.