Functional Elements Specific to Mongodb Databases

Dragoş NICOLAU
National Institute for Research and Development in Informatics – ICI Bucharest
dragos.nicolau@ici.ro

Abstract: The present paper aims to present some considerations on the functional features of MongoDB databases, belonging to the NoSQL typology (increasingly used especially by Web applications such as Youtube, Amazon, Google and Facebook - forced both to store an impressive volume of data and to provide, on the basis of these data, quick answers to competing queries, with an increasing level of rhythmicity). The article's contribution consists in presenting the conceptual equivalences between the classical bases (SQL) and the Mongo bases, as well as in explaining a range of commands used in the dialogue between an application and the MongoDB Server. Additionally, a brief , concrete case study on using MongoDB in a cybersecurity successful solution is here-in integrated.
Keywords: MongoDB, NoSQL Databases, Relational Databases.

INTRODUCTION

Organizations (companies, institutions, agencies, etc.) electronically collect considerable amounts of data for various purposes: product marketing, providing press information in text or multimedia format, providing software [6], presentation of financial, legal or administrative content, the provision of software platforms for telemedicine [4] or the realization of forecasts and analyzes for work strategies, etc. Usually, this data is stored in relational databases. We specify that the relational (classical) basis is a fixed collection of tables; the table is a fixed collection of fields, next to each field remembering a list – in theory, no matter how long - of data of exactly the same type; connections can be established between the tables, either in the form of additional fields - which in a data table store unique identifiers from another data table - or in the form of link tables; the field represents an informational category, ie a segment (identifiable by name) of information of a certain type, well determined. Lately, however, many developers have begun to implement and propose non-relational databases called NoSQL („Not Only SQL"). With the development and spread of Web applications (distinguished by a pronounced interactive character), there has been a need to store an amount of information with an increasing

volume: users are no longer satisfied with the role of querying databases, but they also want to produce content (by sending texts or files of any kind).

The Web universe has forced a rethinking of the way data is stored and processed, so we started looking for specific solutions to build software mechanisms that provide increased speed in analyzing a growing number of records [3]. Databases serving Web applications do not face complex queries (they are generally derived from a multitude of inner associations - equivalent to the intersection of data collections and outer - equivalent to the reunion of data collections), but rather simple yet demanding queries in terms of volume and rhythm. NoSQL databases successfully meet the above-mentioned volume and speed requirements.

Faced with the increase in traffic and storage volume, relational databases are no longer efficient in terms of speed, especially for applications that manage colossal volumes of data. Among the first applications (with impressive data turnover) that raised the issue of adaptability to high effort are Facebook, Google and Youtube. According to estimated studies, the volume of data stored in electronic format is growing rapidly from year to year, in 2018 reaching the level of almost 15 000. exabytes [2] (1 EB = 1018 bytes = 1 billion bytes).

The purpose of the article is to present the conceptual equivalences between classical databases (SQL) and Mongo databases, as well as to explain a range of commands used in the dialogue between an application and the MongoDB Server. In addition, a concrete case study on using MongoDB in a cybersecurity successful solution is included as a brief overview.

GENERAL ASPECTS REGARDING MONGODB DATABASES

MongoDB is a NoSQL database system. The main advantage offered by the non-relational bases is that they allow extremely effective queries, due to the fact that the data (no matter how complex its organization) is stored in a format with „elastic" structure (without scheme and without connection tables; we specify that the scheme means - stricto sensu - the organization of information in tables by dividing it into fields with proper name and fixed type of data). Currently, there are NoSQL databases developed by many companies (such as Amazon and Google) whose Web applications process huge amounts of data [7].

The main functional features of MongoDB databases are:

- Lack of scheme.
- Anatomical simplicity and elasticity, which allows for the execution of quick queries.
- Ability to add / edit / delete dynamically new attributes (fields) to existing records.
- Ability to share load on multiple servers (i.e. distribution of computing „effort").
- Ability to replicate data across multiple servers.
- Simultaneous access is possibly problematic with respect to the one performed within the relational model (i.e. no action on a MongoDB database benefits from any guarantee of exclusivity, unlike actions executed inside classical SQL transactions).

From the developers' perspective, MongoDB's basics are open-source and enjoy generous collections of API functions for dialogue with applications written in various languages. The MongoDB system enjoys very good compatibility with Cloud technology, which is based on virtualization.

The main arguments for the development and use of MongoDB databases are:

- Avoiding morphological complexity by storing data according to the model of structures in programming languages, not according to the relational model. Most applications operate with massive data structures but of relatively low complexity.
- Performing effectively but under certain deployment rules (as stated 2 paragraphs bellow), at the expense of slight suppression of reliability, as is the case with social networking Web sites.

- Programming applications that interact with databases is comfortable.

Responding in due time under the circumstance of massive data traffic stress is possible when conforming to certain deployment rules, one of each being sharding and scaling out, which incurres additional costs. Actually, there are studies that show SQL server configurations, such as Postgresql and Cassandra, outperforming MongoDB in both read and write at the same cost. MongoDB simply requires more servers. So, performance per-se may be an argument for adoption only under the requirement of extra investment in supplementary physical instances.

ABOUT STRUCTURING INFORMATION IN MONGODB DATABASES

Relational databases (classic SQL) mirrored with MongoDB Bases

Principally, MongoDB databases represent a new way of depositing information, with neither schema nor connection tables, the records being JavaScript tree structures.

Storing a huge amount of data (another important feature of NoSQL databases) rests on the „horizontal" distribution of computational effort. „Horizontal" distribution means running several **identical instances** on **different servers** simultaneously. It is useful in situations requiring very high traffic and meeting a significant number of simultaneous requests. Among other notable NoSQL systems used today, Hbase, CouchDB, GTM [9] must be cited. Data storage can be easily organized in complex (tree) models, which emphasizes the increased flexibility offered by the NoSQL typology.

The principle of data storage: classic SQL vs. MongoDB

Each database in MongoDB comprises a set of **collections** (a **collection** is the equivalent of **a table** in SQL databases. Each collection stores **documents**, which are the equivalent of **records** in tables. Whereas a line (record) organizes data storage in a row of columns, a document stores data in a JSON (JavaScript Object Notation) structure. The following is an document sample (classic SQL line equivalent) consisting of some **fields** (**column** equivalents) that store user data:

Information **passed** to the **insert command** of the MongoDB server

```
{
"username: "Ion Georgescu"
"email": "ion.georgescu@abc.com",
"age": 27,
"city": "Constanta"
}
```

Information **returned** by the **read command** of the MongoDB server (the insert mentioned on the left side has already been executed)

```
{
"_id":
ObjectId("5146bb52d8524270060001f3"),
"username": "Ion Georgescu"
"email": "ion.georgescu@abc.com",
"age": 27,
"city": "Constanta"
}
```

This above mentioned document is the equivalent of a single line in classic SQL (i.e. it is the equivalent of a record). A collection contains several such documents, as a classic table contains several records broken down by columns; in each document of a collection there is a unique identification field, a 24-byte field that serves as the primary key for every document. The _id identifier is generated and inserted automatically (by the management system) if it is not present in the structure ready to be inserted („id": ObjectId („5146bb52d8524270060001f3")). Otherwise, it is kept as such (a strongly discouraged

option), obviously if it is not a duplicate. This field is automatically generated (by the MongoDB server) when creating the document (i.e. when inserting) and it is used to identify uniquely each document. The equivalences will be exemplified by a **classic SQL table : MongoDB collection** pair. As shown in the figure below, each row in the SQL table turns into a document (blue frame) and each column into a field. (fig.1)

In short, the counterpart of a **table** is a **collection** (it contai ns the list of documents), and the counterpart of a **line** (**record**) is a **document.**

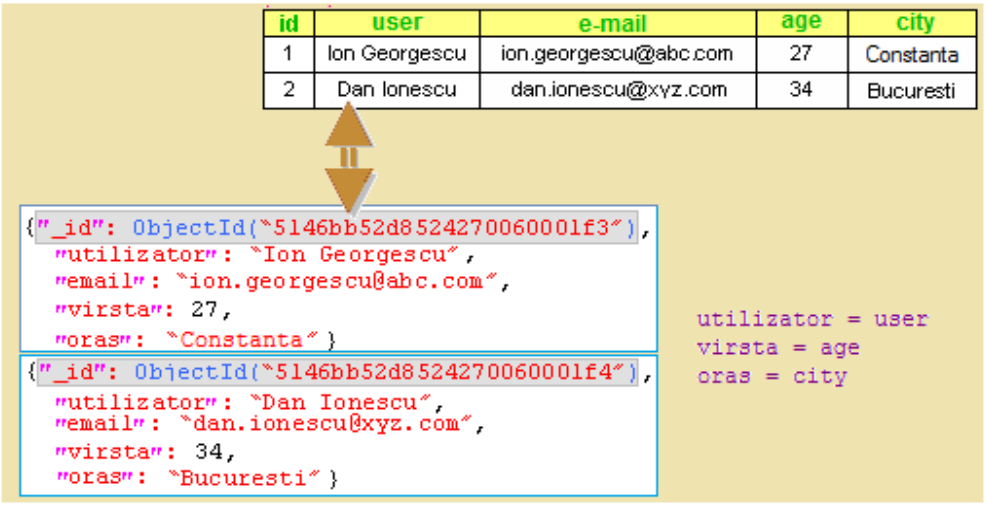

Fig. 1:*Classic SQL table vs MongoDB structure (a collection of documents; each document contains fields)*

An example of a dynamic scheme typical of the MongoDB typology

A remarkable feature of MongoDB basics is that that different documents in a collection may have different "schemas". As such, in MongoDB it is possible for a document to have 5 fields and another "sibling" document to have 7 fields, within the same **collection** (**table**). These fields can be easily added, modified or removed at any time. Moreover, there are no constraints on the data types of the fields, therefore in one instance a field can have data of int type and in the next instance to contain an array. These characteristics are fundamentally different from those of classical SQL where tables, columns, data types and relationships are predefined, rigid. Obviously, this new functionality (the dynamic "scheme") permits generating any variety of document at run-time.

Fig. 2 presents two documents (records) from the same collection (table), with different "schemas":

```
{ "_id":
ObjectId("5146bb52d8524270060001f4"),
"adresa": "Bd. Magheru, sector 1",
"virsta": 34,
"oras": "Bucuresti",
"studii": "Filologie",
"email": "dan.ionescu@xyz.com",
"nume": "Dan Ionescu", }
```

```
{ "_id":
ObjectId("5146bb52d8524270060001f3"),
"virsta": 27,
"oras": "Constanta",
"email": "ion.georgescu@abc.com",
"sex": "B",
"profesie": "inginer",
"nume": "Ion Georgescu", }
```

Fig. 2:*Two MongoDB documents with different "schemas" (anatomies)*

The ID values were chosen as illustrative for the idea of generating documents, the correct value being a **unique** string consisting of 24 hexadecimal digits (i.e. 0: F) **randomly generated**. The first document contains the fields „address (adresa)" and „education(studii)" that are not present in the second document, while the second document contains the fields „sex" and „profession(profesie)" that do not exist in the first document. If one were to use the classic SQL design, he (she) would have had 4 extra columns for all these fields, filled with empty values (or NULL) accordingly, thus taking up unnecessary space. This dynamic schema model is the reason for which NoSQL databases are very „flexible" in design. Various complex schemas (hierarchical, with tree structures etc.) that would require a number of SQL tables, can be efficiently designed using the anatomic versatility of a document. A typical example would be storing user posts (coupled with their specific assessment and comments) or other information that requires tree structuring. A classic SQL implementation for the same information context would require separate tables for storing this data assembly, whereas the MongoDB version can store all this information in a single document.

An example of making connections: classic SQL vs. MongoDB

A remarkable feature of MongoDB basics is that that different documents in a collection may have different "schemas".

Connections (relations) in classical SQL are made using primary and foreign keys, therefore queries use either links (*join*) or value matching tests for „link" fields. (It should be noted that in MongoDB all these complications disappear, because documents can be encapsulated and correlated.) For example, if it is necessary to store user information and contact information for those users, the classic SQL solution would require 2 tables - that is, {Users} and {Phones}, with primary keys (*Primary Key* = PK) called id, as seen in the figure below (fig. 3). The {Phones} table would mandatorily contain a contact_id column that represents the foreign key (*foreign key*=FK) respectively coinciding in value with the id field (column) in the {Users} table. The id and contact_id columns materialize the connection (in this example, the connection is of type **1: 1**) between pieces of information.

id	user	e-mail	age	city
1	Ion Georgescu	ion.georgescu@abc.com	27	Constanta
2	Dan Ionescu	dan.ionescu@xyz.com	34	Bucuresti

USERS

(PK) (FK)

PHONES

id	contact_id	home	mobile	office
101	1	211.11.11	0727-aabbcc	
102	2		0788-xxyyzz	356.22.33

***Fig. 3:** Two SQL (classic) tables connected by "Primary Key to Foreign Key"*

By contrast, MongoDB achieves the connection by inserting a reference, as in the example below. This manner of connecting is the most widespread, but by no means unique.

The MongoDB approach illustrated bellow will use two collections, { Users } and {Phones}, both with their own unique _id fields. In the {Phones} document there is a contact_id field connected with _id from the { Users } document; the contact field illustrates to which user the current phone list corresponds (fig. 4). In MongoDB, the corresponding relationships and operations must be performed manually (i.e. in the code development phase), as neither constraints nor rules of external keys do apply here. Correctly establishing the value match between the contact_id field in the document containing the phones and the _id field of the associated user - is the responsibility of the developer. For example, if a value entered for contact_id (in the {Phones} document) is not retrieved in the {Users} collection, MongoDB will never return any error informing that a connection was made to something that does not exist (unlike classic SQL, which would issue an "invalid foreign key" constraint error).

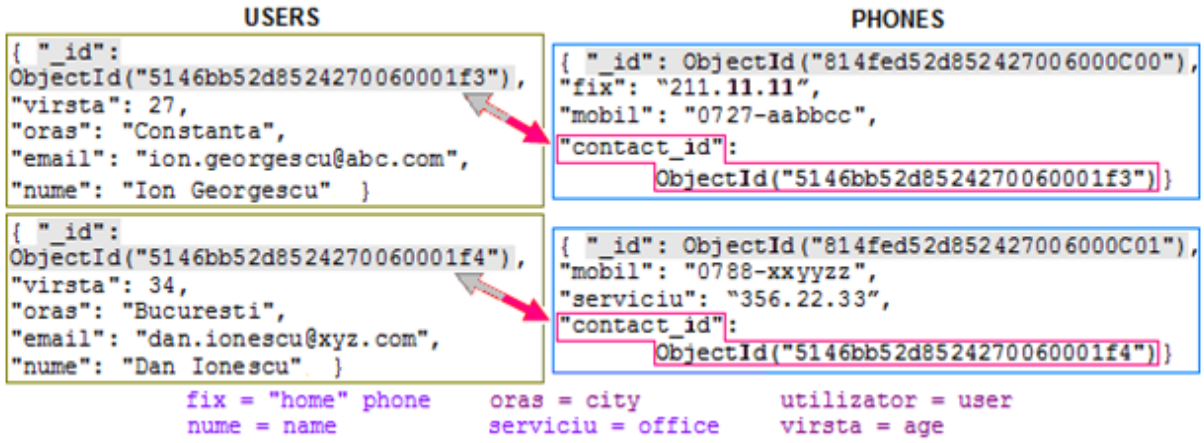

Fig. 4:*Two MongoDB collections, connected by inserting a reference, without keys (neither primary nor external)*

Another way to create MongoDB connections between categories of information is encapsulating entire documents: within the {Users} type document there is a peculiar field that contains all contact data; therefore, a whole complete information is injected on the spot [8]. In a similar manner, large, complex documents, as well as hierarchical data can be embedded to make connections between entities. The "injection" mode chosen to incarnate the connection - i.e. either the connection done by inserting a reference, or the connection done by encapsulating an entire document - depends on the specific scenario. If it is estimated that the data to be embedded will occupy large volumes of memory over time, a connection accomplished by inserting a reference is preferable.

The synthesized parallelism of the concepts: classic SQL vs. MongoDB

Fig. 5, below, represents the synthesis of what was previously presented regarding the parallelism between the concepts encountered in classical SQL and those used in MongoDB.

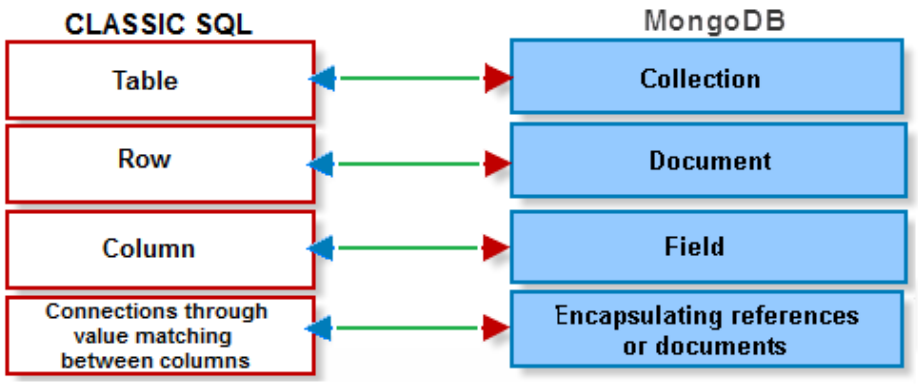

Fig. 5:*The synthesized parallelism of the concepts: classic SQL vs. MongoDB*

CLASSIC SQL COMMANDS VS. MONGODB COMMANDS

MongoDB is an open-source NoSQL database management system (written in C ++) designed to work with unstructured (in the meaning of classical database world) data, organizing them in block-tree format. MongoDB ensures advanced performance, high availability and very good adaptation to increased computing effort [1].

MongoDB is equipped by default with a console client application, namely the **bin / mongo.exe** executable, representing the interactive shell (client executable, Console type, which analyzes and sends commands for interpretation to the Windows Mongo Service - embodied by bin / **mongod. exe**) also written in C / C ++ and able to harvest the returned results. The shell is useful for test checks and administrative functions (literally, „Shell" means „bunch of encapsulated commands"). Each database is physically personified by two files (**MyBase.0** and **MyBase.ns**) located in a dedicated folder, configurable after installation by editing an initialization file.

It is worth noting that all of the MongoDB syntaxes below (containing table names, function names, field names, parameters, and arguments) are NOT „per se" calls, but simple JavaScript texts that are transmitted to the MongoDB server, to be interpreted - just as an established SQL command is also a simple text carrying an SQL syntax, conveyed to the relational database server, to be interpreted. The MongoDB server is the Windows Mongo Service - embodied by **bin / mongod.exe.**

Create

In MongoDB it is not necessary at all to create a collection structure explicitly (as it is when creating classic SQL tables using a CREATE TABLE query). The individual anatomy is created automatically for every document when inserted in the collection. However, an empty collection can be created using the createCollection() command.

SQL:

CREATE TABLE `postari` (`id` int (11) NOT NULL AUTO_INCREMENT,

`text` varchar (500) NOT NULL, `user` varchar (20) NOT NULL, `access` varchar (10) NOT NULL, `assessment` int (11) NOT NULL, PRIMARY KEY (` id`));

MongoDB:

db.createCollection(„postari")

Insert

In MongoDB to insert a document we use the insert() method which takes as input an object with key value pairs.

The inserted document will contain the self-generated field _id. A value of 24 bytes

for _id can also be used explicitly, but this is a soundly not-recommended practice.

SQL:

INSERT INTO `postari` (`id`, `text`, `user`, `access`,

`assessment`) VALUES (NULL, ‚Something posted ...', ‚Ionescu', ‚public', ‚0');

MongoDB:

db.postari.insert({user: „Ionescu", text: „Something posted ...", access: „public", assessment: 0})

There is no Alter Table command to change the anatomy of the document. As documents are dynamic, their „scheme" (anatomy) can be modified at any time, at execution.

Read

MongoDB uses the find() method which is equivalent to the SELECT command in SQL. The following MongoDB command effortlessly reads all the documents in a given collection:

SQL: SELECT * FROM `postari`;

MongoDB: db.postari.find()

The next query performs a conditional search for documents by the value „Ionescu" encountered for the **user** field. All criteria for finding documents must be placed in the first {} and separated by commas.

SQL: SELECT * FROM `postari` WHERE `user` = ‚Ionescu';

MongoDB: db.postari.find ({user: „Ionescu"})

The following queries return certain columns, text, and assessment as specified in the second set of {}.

SQL: SELECT `text`, `assessment` FROM `postari`;

MongoDB: db.postari.find ({}, {text: 1, assessment: 1})

Note that by default MongoDB returns the _id field with each search statement. If we do not want to have this field in the result set, the _id key with a value of 0 must be specified in the list of columns to be returned. The value 0 of the key indicates that we want to exclude this field from the resulting set.

MongoDB: db.postari.find ({}, {text: 1, assessment: 1, _id: 0})

The following query returns certain fields based on the value „Ionescu" encountered for the user field:

SQL: SELECT `text`, `assessment` FROM `postari` WHERE `user` = ‚Ionescu'

MongoDB: db.postari.find ({user: „Ionescu"}, {text: 1, assessment: 1})

Below is another criterion to return posts having the type of access = public. The criteria specified using commas represent the logical AND / OR condition. Therefore, this statement will search for documents that have user = Ionescu and/or access = public:

SQL: SELECT `text`, `assessment` FROM `postari` WHERE `user` = ‚Ionescu' AND `access` = ‚public'

MongoDB: db.postari.find ({user: „Ionescu", access: „public"}, {text: 1, assessment: 1})

SQL: SELECT `text`, `assessment` FROM `postari` WHERE `user` = ‚Ionescu' OR `access` = ‚public'

MongoDB: db.postari.find({$or: [{user: „Ionescu"}, {access: „public"}]}, {text: 1, assessment: 1})

The sort() method is used to sort the results in ascending order by assessment (indicated by 1):

SQL: SELECT * FROM `postari` WHERE `user` = ‚Ionescu' order by assessment ASC

MongoDB: db.postari.find({user: „Ionescu"}). sort ({assessment: 1})

To sort the results in descending order, specify the value -1 for the field:

SQL: SELECT * FROM `postari` WHERE `user` = ‚Ionescu' order by assessment DESC

MongoDB: db.postari.find({user: „Ionescu"}). sort ({assessment: -1})

To limit the number of documents to be returned, use the limit() method, specifying the number of documents:

SQL: SELECT * FROM `postari` LIMIT 10

MongoDB: db.postari.find().limit(10)

In the same way in which offset is used in SQL to skip a certain number of results, in MongoDB the skip() function is used. The query below returns 10 posts skipping the top 5:

SQL: SELECT * FROM `mail` LIMIT 10 OFFSET 5

MongoDB: db.postari.find().limit(10).skip(5)

Update

The first argument in the update() method specifies the document selection criteria. The second argument specifies the current update operation to be performed. For example, below are selected all documents with username = Ionescu to whom is set access = private:

SQL: UPDATE `postari` SET access = „private" WHERE user = ‚Ionescu'

MongoDB:

db.postari.update({user: „Ionescu"}, {$set: {access: „private"}}, {multi: true})

The difference from classic SQL is that MongoDB executes update() only once, on the first returned document. To update all documents of interest, a third argument must be set, specifying **multi** as **true**, thus indicating the intention to update supplementary documents.

Remove

SQL: DELETE FROM `postari` WHERE user = ‚Ionescu'

MongoDB: db.postari.remove ({user: „Ionescu"})

Indexing

By default, MongoDB is designed to „consider" the **_id** field as the default index for each **collection** (table). To give other fields the default index prerogative, the ensureIndex() method is used, specifying the fields and sort order by 1 or -1:

SQL: CREATE INDEX index ON `postari` (user, DESC assessment)

MongoDB: db.postari.ensureIndex({user: 1, ratings: -1})

MongoDB: db.postari.getIndexes() is the command that provides (returns) the list of indexes of a collection.

MONGODB AND CYBERSECURITY

From here onwards, a concrete case in which MongoDB is considered an available solution for serving applications involved cybersecurity, is presented.

McAfee analyzes cyberthreats from all angles, identifying threat relationships, such as malware used in network intrusions, websites hosting malware, botnet associations, and more. Threat information is extremely time sensitive: knowing about a three week old threat is useless.

In order to provide up to date, comprehensive threat information, McAfee needs to process quickly terabytes of different data types (such as IP address or domain) into meaningful relationships: e.g. is a certain web site trustworthy or not? or what other sites have been interacting with it? etc. Also, the success of a cloud-based system depends on a bidirectional data flow: GTI (Gateway Transaction Interface) gathers data from millions of client sensors and provides real-time intelligence back to these end products, at a rate of 100 billion queries per month.

McAfee was unable to address these needs and scale out effectively to millions of records with their previously existing solutions. For example, the HBase / Hadoop setup made it difficult to run convoluted, complex queries, and, on the other hand, experienced bugs with the Java garbage collector running out of memory. Another issue was with sharding and syncing; Lucene was able to index in interesting ways, but required too much customization. McAfee compensated for all the rebuilding and redeploying of Katta shards with "the usual scripting duct tape," but what they really needed was a solution that could seamlessly handle the sharding and updating on its own.

As the company was spending more time building solutions in-house rather than focusing on threat research [10], an effective database engine was needed in order to permit the developers to concentrate on : finding interesting bits in the data, figure out who's being harmful on the web at any given moment, and report that up the chain for whomever wants to use it.

McAfee selected MongoDB, which has excellent documentation and a developer

community that is increasing. MongoDB enables McAfee to develop quickly on a platform that can mount, delivering time to market advantages. Writing proof of concept applications has become comfortable to do in MongoDB. Plus, the ability to change document schema on the fly boosts productivity.

Auto-sharding makes it rather effortless to add more servers at any time to the effect of copping with Gateway Transaction Interface increasing data needs, as a two-fold increase in data over the last two year period (2018-2020) was noticed, a trend that is expected to continue. With the capacity to store more data, McAfee gains more visibility into threats and is able to perform more interesting data analysis. GTI receives queries of its data as JSON objects, which it can pass with minimal transformation into MongoDB. This greatly simplifies query workflow, and MongoDB's rather good speed and indexing capability obviates the need for a separate search engine solution such as Lucene / Katta. MongoDB is sometimes fast – queries on the user-facing McAfee. com site, for example, are now completed in ~150ms, down from 500ms.

CONCLUSIONS

This paper focuses on the presentation of the main functional features of MongoDB databases, belonging to the NoSQL typology.

The main feature of MongoDB databases is that they allow data to be stored in an unstructured way (NOT in the form of a classic table-rigid scheme, but in a tree form - the scheme being non-existent, which makes it possible to host dynamically any conceivable configuration, provided that the JSON syntax is observed), which considerably increases the efficiency of accessing them. Data can be stored in any possible combination (texts, integers, files, collections of entities, etc. or any imaginable grouping of them). The fundamentals of MongoDB are lack of schema, anatomical simplicity - which allows the execution of quick queries, the ability to add / edit / delete dynamically new attributes (fields) to existing records, the ability to share the load on multiple servers (i.e. distribution „ computing effort), the ability to replicate data on multiple servers and possible problems with simultaneous access (no guarantee of exclusivity for read/write operations, unlike the assurance given by the transactions of the classic relational model). In terms of software development, the basics of MongoDB are open-source and enjoy generous collections of API functions for dialogue with applications written in various languages. For concrete and simple examples, the equivalence between classic SQL syntax and NoSQL syntax of JavaScript origin is also illustrated.

From **another cyber-security perspective**, it should be emphasized that MongoDB databases are exposed to the danger of NoSQL injections, just as classical databases are exposed to the homologous menace, namely the well-established SQL injections. We remind that an injection is the process of sending (injecting) a command instead of "honest", innocuous pieces of information (data) with the purpose of changing radically the functioning of a command: the newly resulted command is intended to cause damages to the database or to obtain unauthorized access to stored data. Although the recent generations of MongoDB-server shell is planned to reject NoSQL injections, the responsibility for immunizing against such attempts still rests with he (she) who develops the application that communicates with the MongoDB service (server).

The article's contribution consists in presenting the conceptual equivalences between classical bases (SQL) and Mongo bases, as well as in explaining an illustrative range of commands used in the dialogue between an application and the MongoDB Server; also, a concrete case study on using MongoDB in a cybersecurity successful solution has been inserted as a brief overview.

AKNOWLEDGEMENTS

This article is the result of the software developing activity that the author has performed within the Project "***RO-SmartAgeing; 2021 phases***".

REFERENCE LIST

*** - "C# Driver" - 2018; https://docs.mongodb.com/getting-started

*** - "Data storage supply and demand worldwide, from 2009 to 2020 (in exabytes)"; https://www.statista.com/statistics/751749/worldwide-data-storage-capacity-and-demand – 2018

*** - "NoSQL Databases" - 2018; http://nosql-database.org/

Alexandru, A., Coardoş, D., Nicolau, D. - "A Model For Future Internet Of Things-Based Remote Monitoring Of Chronic Diseases "; Proceedings of the IE 2018 International Conference - Iaşi, 2018

Andersson, Erik & Berggren, Zacharias - "A Comparison Between MongoDB and MySQL Document Store Considering Performance"; http://www.diva-portal.org/smash/get/diva2:1161166/FULLTEXT01.pdf - 2017

Băjenaru, L., Marinescu, I. A., Tomescu, M., Savu, D. - "Biblioteca Naţională de Programe: O nouă abordare în managementul produselor software" - RRIA, vol. 27, nr. 4, decembrie 2017; https://rria.ici.ro/rria-vol-27-nr-4-2017

Nicolau, .D - "Consideraţii asupra bazelor de date NoSQL" - RRIA 2018

Tomescu, M., Savu, D., Marinescu, I.A. (2017). „Servicii Cloud de versionare cu suport baze de date NoSQL". Revista Romînă de Informatică şi Automatică (RRIA), Editura ICI,, vol. 27, nr. 3, septembrie 2017

Shalom, N. - "The Common Principles Behind The NoSQL Alternatives", Dec 2009 http://natishalom.typepad.com/nati_shaloms_blog/2009/12/the-common-principlesbehind-the-nosql-alternatives.html

Widner, Wess - "McAfee is Improving Global Cybersecurity With MongoDB" - 2020, https://www.mongodb.com/customers/mcafee