NATIONAL AND KAPODESTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSC THESIS

Efficient Queries in MongoDB with Encrypted Fields

Andriani G. Triantafyllou
Eleni G. Mantzana

Supervisors: **Alexios Delis,** Professor NKUA
Panagiotis Liakos, PhD student NKUA

ATHENS

OCTOBER 2017

BSC THESIS

Efficient Queries in MongoDB with Encrypted Fields

Andriani G. Triantafyllou
S.N.: 1115201300179
Eleni G. Mantzana
S.N.: 1115201300091

SUPERVISORS: **Alexios Delis,** Professor NKUA
Panagiotis Liakos, PhD student NKUA

ΕΘΝΙΚΟ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ

Efficient Queries in MongoDB with Encrypted Fields

Ανδριανή Γ. Τριανταφύλλου
Ελένη Γ. Μαντζάνα

Επιβλέποντες: **Αλέξιος Δελής**, Καθηγητής ΕΚΠΑ
Παναγιώτης Λιάκος, Υποψήφιος Διδάκτορας ΕΚΠΑ

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2017

ΠΤΥΧΙΑΚΗ

Efficient Queries in MongoDB with Encrypted Fields

Ανδριανή Γ. Τριανταφύλλου
Α.Μ.: 1115201300179
Ελένη Γ. Μαντζάνα
Α.Μ.: 1115201300091

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Αλέξιος Δελής**, Καθηγητής ΕΚΠΑ
Παναγιώτης Λιάκος, Υποψήφιος Διδάκτορας ΕΚΠΑ

ΠΕΡΙΛΗΨΗ

Στην παρούσα πτυχιακή εργασία παρουσιάζουμε αποδοτικές τεχνικές εισαγωγής δεδομένων και εκτέλεσης ερωτημάτων (queries) σε μια μη σχεσιακή βάση δεδομένων (Non Relational Database), δίνοντας την επιλογή στον χρήστη να κρυπτογραφήσει κάποια από τα πεδία της εγγραφής που εισάγει. Ασχοληθήκαμε με τον Java Driver μιας μη σχεσιακής βάσης και πιο συγκεκριμένα της MongoDB, τροποποιώντας κάποιες ήδη υπάρχουσες συναρτήσεις του και ενισχύοντάς τον με δικές μας συναρτήσεις προκειμένου να πετύχουμε την κρυπτογράφηση (encryption) των δεδομένων. Με τις αλλαγές που πραγματοποιήσαμε, υποστηρίζεται πλέον η εισαγωγή εγγραφών στη βάση οι οποίες περιέχουν κρυπτογραφημένα πεδία (encrypted fields). Συγκεκριμένα, έχουν υλοποιηθεί δύο τρόποι κρυπτογράφησης: κρυπτογράφηση με χρήση SHA-256[1] και BCrypt[2] κρυπτογράφηση. Η SHA-256 (Secure Hash Algorithm μήκους 256 bits) βασίζεται σε πολλαπλούς "γύρους" κατακερματισμού (hashing). Η BCrypt επίσης βασίζεται σε hashing συνάρτηση, προσδίδοντας όμως μεγαλύτερη ασφάλεια λόγω της salt προσθήκης, ένα τυχαίο δεδομένο που χρησιμοποιείται κατά την παραγωγή της κρυπτογραφημένης εξόδου των δεδομένων. Για την κρυπτογράφηση των πεδίων με τις δύο παραπάνω μεθόδους έχουν αξιοποιηθεί οι βιβλιοθήκες DigestUtils[3] και BCryptPasswordEncoder[4] του Spring για την SHA-256 και την BCrypt αντίστοιχα. Σκοπός της παρούσας πτυχιακής, λοιπόν, αποτελεί η χρονική μελέτη των εισαγωγών και της εκτέλεσης ερωτημάτων πάνω στη NoSQL βάση, σε σύγκριση με τον απλό Java Driver που δεν χρησιμοποιεί κρυπτογράφηση.

Αρχικά, παρατίθεται και αναλύεται ο αλγόριθμος που χρησιμοποιήθηκε για την αποδοτική εισαγωγή των δεδομένων με χρήση της SHA-256 κρυπτογράφησης στα επιλεγόμενα πεδία μιας εισαγωγής. Βασικό στοιχείο της υλοποίησης είναι το ότι δίνεται η δυνατότητα καθορισμού από τον χρήστη των συγκεκριμένων πεδίων που επιθυμεί να εμφανίζονται στη βάση κρυπτογραφημένα. Επιπλέον, μελετάται ο αλγόριθμος που αναπτύχθηκε για την αποδοτική αναζήτηση στη βάση των εγγραφών οι οποίες περιέχουν κρυπτογραφημένα πεδία και αναλύεται ο τρόπος υλοποίησής του, που είχε ως αποτέλεσμα οι χρόνοι εισαγωγής και αναζήτησης με ταυτόχρονη ύπαρξη κρυπτογραφημένων πεδίων να ανταγωνίζονται αυτούς του ήδη υπάρχοντος Java Driver.

Στη συνέχεια, παρουσιάζεται ο αλγόριθμος για την εισαγωγή των δεδομένων με χρήση BCrypt κρυπτογράφησης στα επιλεγόμενα πεδία μιας εγγραφής. Δίνεται η δυνατότητα προσδιορισμού των συγκεκριμένων πεδίων, που θα είναι κρυπτογραφημένα. Παρακάτω, προβάλλεται ο αποδοτικότερος αλγόριθμος για την εφαρμογή ερωτημάτων πάνω στη βάση για αυτόν τον τρόπο κρυπτογράφησης και αναλύονται οι παράγοντες διαφοροποίησής του από τον προαναφερθέν.

Ακολούθως, παρατίθενται χρονικές μετρήσεις τόσο απλών, βασικών ερωτημάτων, αλλά και πιο πολύπλοκων ερωτημάτων όπως για παράδειγμα με χρήση ενσωματωμένων πεδίων (embedded fields). Γίνεται σύγκριση των αποτελεσμάτων τόσο μεταξύ των δύο παραπάνω τρόπων προσέγγισης σε ό,τι αφορά τους χρόνους εισαγωγής και αναζήτησης εγγραφών στη βάση, όσο και μεταξύ της υλοποίησης με κρυπτογραφημένα πεδία και του αρχικού, ευρέως διαδεδομένου, Mongo Driver που δεν υποστηρίζει επερωτήσεις σε κρυπτογραφημένα πεδία. Παράλληλα, γίνεται ανάλυση των trade-offs σε κάθε περίπτωση.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Κρυπτογραφημένη Εισαγωγή Πεδίων σε Mongo Βάση Δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: ασφάλεια, nosql βάσεις, mongodb, κρυπτογραφημένα πεδία, χρονικές μετρήσεις εισαγωγών και ερωτημάτων

ABSTRACT

In this thesis, we present efficient techniques for inserting data and running queries over a non-relational database, giving the user the option to encrypt certain fields, the ones they want, of the document they insert. We worked on the Java Driver of a non-relational database, more specifically the MongoDB, by modifying some of its existing functions and enhancing it with our own functions in order to achieve encryption of the data. With the changes we made to the Java Driver, our application now supports the insertion of documents containing encrypted fields and gives the user the ability to run queries even about the encrypted fields. In particular, two encryption modes have been implemented: encryption using SHA-256 and BCrypt encryption. The SHA-256 (256-bit Secure Hash Algorithm) is based on multiple “rounds” of hashing. BCrypt also relies on a hashing function, but considered to be a more secure algorithm due to the addition salt, a random data used in the production of the encrypted data output. In order to encrypt the fields with the two encryption types we mentioned before, we have utilized the libraries DigestUtils [3] and BcryptPasswordEncoder[4] from Spring for SHA-256 and BCrypt encryption respectively. The main purpose of this thesis is to study the efficiency of inserting data and running queries on a NoSQL database, with the data containing encrypted fields, compared to the simple Java Driver that does not support encryption.

First, the algorithm used in order to efficiently insert documents into the database using the SHA-256 encryption on the requested fields is quoted and analyzed. A key element of the implementation is that we provide the user with the ability to define the specific fields they want to appear on the database as encrypted fields. In addition to this, the algorithm developed in order to achieve efficient querying on the database for document fields that are encrypted, and its implementation is analyzed. The algorithm we developed resulted in having an insertion time with encryption and a querying process time (with the data being encrypted) that competes with the time the existing Java Driver needs to complete those processes.

Then, the algorithm used in order to support BCrypt encryption is presented and analyzed. Again, the user is able to specify the fields they want to encrypt with the BCrypt algorithm. Below, the most efficient algorithm for querying the base for this encryption mode is shown and the factors that make it different from the SHA-256 are analyzed.

Subsequently, time measurements of both simple, basic queries and more complicated queries, such as using embedded fields, are presented. The results are compared in two ways: firstly, there is the comparison between the two encryption methods, SHA-256 and BCrypt, and secondly, the comparison between our approach and the existing insert, find etc methods of Mongo Driver library. At the same time, trade-offs are analyzed in each case.

SUBJECT AREA: Insertion of Encrypted Fields in MongoDB

KEYWORDS: nosql database, mongodb, security, encrypted fields, insertion and query time measurements

ACKNOWLEDGEMENTS

We would like to thank our supervisors, Mr. Alexis Delis and Mr. Panagiotis Liakos for their excellent guidance and asistance in the preparation of this thesis.

CONTENTS

FIGURES LIST

TABLES LIST

INTRODUCTION

Nowadays, we live in a modern society in which Internet has become an integral part of our lives. People of different ages and different professional fields use it for both personal and professional reasons. One of the main reasons they use it is the potential to store data online. Storing data online offers them numerous advantages. Some of these advantages are the easy access on the data saved, the fact that the data are portable and accessible from everywhere, the potential for multiple access on them, the back-up usage of them, etc.

Usually, people have the tension to store sensitive information on the internet. So, most companies which provide such services are very concerned for the data safety and are trying their best in order to provide the users with secure applications We also have to take into consideration the fact that not only everyday people need to store their data online, but also a great amount of large companies are becoming digital day by day and choose to run their applications and their systems on cloud services. However, the malicious attacks, attacks against computer systems or database systems in order to crack them and gain access to the data they contain, have increased in a noticeable scale. It has become an urgent need for the world of informatics to develop secure methods of storing data online, wihtout them being vulnerable to malicious attacks.

The applications supplying us with online storing services, store the data of their users either in Relational or Non Relational databases. Technology, at the time, has made an impressive progress on what concerns the security threats in Relational Databases, providing the users with a wide variety of different secure storing and managing practices, as we are going to analyze later, especially compared to the options given for Non Relational Databases. This is the reason why we decided to focus on Non Relational Databases whose use in web applications is ever-increasing because of their advantages on managing big data. More specifically, we focused on the Non Relational Database called MongoDB. After carefully studying on it, we came to the conclusion that its driver does not support any form of encryption on the fields of the documents a database can contain, resulting to the absence of the option to run queries on documents containing encrypted fields so we decided to enrich the original Java Driver by supporting insertions with documents including encrypted fields and we gave the user the ability to perform queries on those documents.

An approach similar to ours has already been implemented by Xingbang Tian, Baohua Huang and Min Wu on 2014 in order to be presented to the 2014 IEEE Workshop[5]. This approach, titled “A Transparent Middleware for Encrypting Data in MongoDB”, offers field encryption by extending some of the existing interface. The algorithm is based on a flag that indicates whether the BSON document contains an encrypted field or not and the implementation does not offer querying process on the encrypted fields. The basic disadvantages of this approach are that having a flag that informs the user whether the document contains encrypted fields is considered a security vacuum and its existence slows down the search process and the fact that search on encrypted values is not supported.

Our approach, on the other hand, provides a solution to those problems by introducing some methods that enhance the MongoDB Java Driver with the operations of insert a document that contains an encrypted value, insert a batch of documents that contain an encrypted value and search queries with matching values. The implementation we are going to present you is running on a MongoDB and it allows the user to choose between two different encryption types, the SHA-256 and the BCrypt, which are extensively analyzed below, while trying to meet two basic requirements of a synchronous management system used to handle a large volume of sensitive data: the urgent requirement to ensure the data security, e.g. a password, against a wide variety of attacks and the demand of time-efficient insertion of large volume of data into databases.

1. STORING DATA

1.1 Relational Databases

A widespread model for managing efficiently a database system is the Relational Model [6]. The Relational Database Model (RM) was introduced by Edgar F. Codd in 1969. This model works as follows: all the data contained in a database are represented by tuples consisting of specific variables. A model consists of tables and every tuple is part of a table, with the tables being able to be correlated with each other. A tuple is inserted into the database using a SQL query, the INSERT e.g. INSERT INTO TABLE (column1, column2, column3, …, columnN) VALUES (value1, value2, value3, …, valueN);. A user can access the data directly by running a SELECT query on the database, e.g. SELECT column1, column2, ... FROM tableName WHERE column1 = 1;, with the software system being responsible for returning the correct results to the user.

The basic feature of the Relational Model is that the representation of the data has to be very specific, the information has to be consistent and represented in a logical way. In order to achieve that, we have s to place some limitations on the design of the model.

Thus, two very important issues arise, which are the basic problems of the SQL databases. The first one is about the flexibility and the scalability of the database and the second one is about the complexity of the data that can be inserted into the database. On what concerns the first issue, for an SQL database to quickly return the search results, all the contents of a table have to be on the same server, otherwise the tables are difficult to handle and the querying process slows down significantly. As far as the complexity of the data is concerned, as we have already mentioned, the data to be inserted into a Relational Database should somehow correspond to and fit into one of the tables that are already present on the model.

Given the fact that we are at a time where the size of the data and the information we want to be stored is enormous and its form is quite complex, one can easily understand that the use of a model that does not have the limitations of the Relational Model is now imperative, in order to get the results of e.g. a search as fast as possible [7, 8].

1.2 Non Relational Databases

Apart from the Relational Model, there is also the Non Relational Model, which is represented by the NoSQL[9] databases. The term NoSQL was firstly introduced by Carlo Strozzi in 1998 in order to describe a shell-baseed relational database management system he created. The term was reintroduced to the programming community in early 2009 by Johan Oskarsson, in an event he arranged, dedicated to ”open source distributed, non relational databases”. There are five different types of NoSQL databases:

1. **Key-Value Store,**
2. **Document-based Store,**
3. **Graph-based,**
4. **Column-based store and**
5. **Multi-model databases.**

On this paper we examine the MongoDB, which is an example of a Document-based Store database.

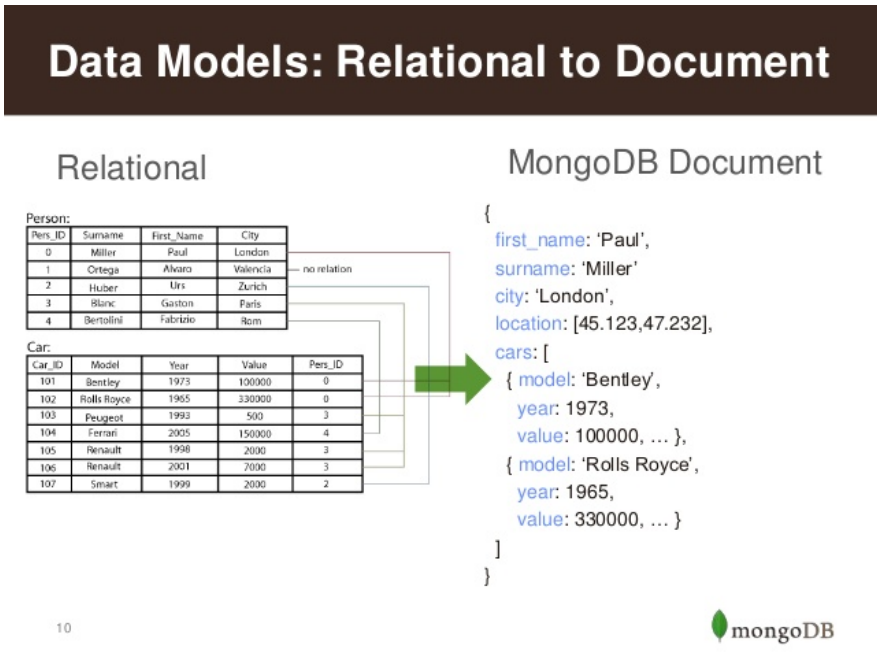
NoSQL databases allow us to store data that are not only represented by a tuple that is part of a table, like a SQL database, but data that are modeled in other ways too, like for example a JSON-like (JavaScript Object Notation) document. These systems are mainly used when we have to store a very large amount of data (like big data applications) and in real-time online applications. A real-time online application is an app that enables the users to upload and receive information the exact moment they are using it, without requiring a software check in order to upload or download the data. An example of such an application is Facebook and Twitter, where the user posts a status update or tweets something, and the content of their post/tweet is immediately available to their friends and followers.

Those databases came to give a solution to two main SQL databases problems:

1. **this of flexibility and scalability of the database**
2. **and this of the complexity of the data that can be inserted in the database.**

On what concerns the scalability, the fact that a NoSQL database does not consist of a certain model with certain tables, results to a more "flexible" database. That means that the database can be splitted on many servers, with the split not causing a problem to the insertion and querying process on what concerns the time needed to complete those two actions. This characteristic makes the expansion of a Non Relational Database much cheaper and less complex than the expansion of a Relational Database[10, 11].

A major problem for the SQL databases is that they do not allow embedded fields, which is a very efficient way of representing possible relations between the data, and that the data have to be represented in a certain way: a way that makes them fit into a table of the model. On the contrary, a NoSQL database, and especially a MongoDB which is a Document-based Store database can store data with the only "restriction" being that they are JSON-like documents. In those documents, the user is given the chance to store data of any type, with every kind of relations they want and without the restriction of a certain format.



Data Models: Relational to Document

Relational

Person:

Pers_ID	Surname	First_Name	City
0	Miller	Paul	London
1	Ortega	Alvaro	Valencia
2	Huber	Urs	Zurich
3	Blanc	Gaston	Paris
4	Bertolini	Fabrizio	Rom

no relation

Car:

Car_ID	Model	Year	Value	Pers_ID
101	Bentley	1973	100000	0
102	Rolls Royce	1965	330000	0
103	Peugeot	1993	500	3
104	Ferrari	2005	150000	4
105	Renault	1998	2000	3
106	Renault	2001	7000	3
107	Smart	1999	2000	2

MongoDB Document

```
{
 first_name: 'Paul',
 surname: 'Miller'
 city: 'London',
 location: [45.123,47.232],
 cars: [
  { model: 'Bentley',
    year: 1973,
    value: 100000, … },
  { model: 'Rolls Royce',
    year: 1965,
    value: 330000, … }
 ]
}
```

Figure 1: Migrating Relational Database to Document [23]

2. DATABASE SECURITY PROBLEMS

Today, everything runs on a database, from web applications to businesses. Databases are mostly used to store the information needed for the above to operate and most of the time the data they contain are sensitive and should not be accessed by everyone. A database can be stored locally, on one or multiple servers, at the cloud. The sensitivity of the information stored makes the databases a target to various malicious attacks such as DDoS attacks (Distributed Denial of Service attack), SQL injections for the Relational Databases or direct server attacks. This situation creates a major problem to every operation relying on databases: the administrator of the database has to continuously secure the private data from every possible attack.

2.1 Encryption as a solution

In general, apart from protecting the databases in a physical way by securing the servers with firewalls, the confidentiality of the data is protected by using various encryption types before storing the data. Precisely, before inserting the data into the database, a key is used in order to encrypt the sensitive value and instead of storing the original value, the encrypted one is stored. The encryption protects the data in a way that even if someone compromises the servers physically or manages to gain access to the data in another way, the only thing they will actually obtain is a huge amount of data that, without the key to decrypt them, is useless [12, 13].

2.1.1 Encryption in Relational Databases

All relational databases nowadays are using one or more of the following encryption approaches:

1. **Transparent/External database encryption,**

   also known as “data at rest”. In this method, the whole database is encrypted and it is mostly used to encrypt data that are stored on physical storage media such as hard disk drives.

2. **Column-level encryption,**

   where individual columns of a table are encrypted with a key. The main problem with this method is that the database’s performance decreases, as well as searching and indexing because of the time needed to decrypt each column.

3. **Field-level encryption,**

   which is mostly used when a user needs the data to be both encrypted and comparable without the need to decrypt them.

4. **Encrypting File System,**

   a method that applies not only to data that are part of a database system, but to data that are independent from the database. This type of encryption is used to encrypt databases that do not require frequent changes due to the performance issues it has.

5. **Symmetric and asymmetric database encryption,**

   the symmetric encryption applies a private key to data that is stored in a database. The data can be decrypted by applying the same secret key to the encrypted value. The main advantage of this method is the speed. Asymmetric encryption uses two keys, a public and a private one. The public key is the encryption key and it can be accessed by anyone whereas the private key is unique and can be accessed by a certain user and it is used to decrypt the data.

6. **Key management,**

   after the introduction of symmetric and asymmetric encryption, it became urgent to store all public and private keys to the database because even if a single key was lost, the data that had been encrypted with that key would be lost too. A Key Management System was the solution to that problem.

7. **Hashing,**

   where sensitive data, for example a password, is converted into a string of a fixed length by applying a hashing algorithm to it and then the string is stored into the database. A big advantage of hashing is that it is irreversible, in order to match 2 values, one has to apply the same hashing algorithm on both of them and compare the hashed results. To make hashing even stronger, methods like salting and pepper were introduced.

8. **Application-level encryption,**

   where the application itself is responsible for the encryption of the data.

2.1.2 Encryption in Non Relational Databases

On this paper we deal with a NoSQL database, and especially the MongoDB. Right now, MongoDB supports only two encryption modes:

1. **Transport Encryption,** where all of the MongoDB’s network traffic is encrypted using TLS/SSL (Transport Layer Security/Secure Sockets Layer). TLS/SSL are both cryptographic protocols that secure the communications over a computer network. They use symmetric cryptography, public-key cryptography and a message authentication code in order to secure the integrity of the data.
2. **Encryption at Rest,** where the MongoDB Enterprise uses AES256-CBC (Advanced Encryption Standard in Cipher Block Chaining mode) algorithm via OpenSSL. In this case, the data are encrypted with a symmetric key that is used in order to both encrypt and decrypt the data. Those symmetric keys are also stored into the database, while the master key is external to the server.

2.1.3 Our solution

After carefully examining the aspects of SQL and NoSQL databases on what concerns the security of the data, we reached the conclusion that even though both models support encryption in different ways, none of the models provides a default, build-in, efficient way of running queries on the database while supporting encryption. Given the fact that we cannot change the basic structure of the Relational Model, we decided to work on a Non Relational Database, the MongoDB and enhance the MongoDB Java Driver.

The MongoDB is a document database with great scalability and flexibility. We can store JSON-like documents, whose fields can vary. On what concerns the security, we have already mentioned the basic ready-to-use encryption methods this database provides, transport encryption and encryption at rest. When compared to a SQL database, MongoDB does not offer the variety of encryption types the SQL offers. So our idea is to improve the Java Driver by overriding some of its functions: insertOne, insertMany, find, and modify the Driver in order to provide encryption to the documents.

We decided to use two different algorithms, the SHA-256 and the BCrypt. Our approach is to enhance MongoDB with something like the Column-level encryption or the Field-level encryption that SQL offers, using hashing algorithms. When a user wants to insert a document into the database, they can choose which field of the JSON document they want to encrypt and they also have the ability to choose the encryption algorithm they want to use. In order to achieve that, we created an application in which the user gives the name of the field to encrypt and the encryption method.

Basically, we implemented some functions that work exactly like the original Java Driver functions, but in their body, we get the value of the field we want to encrypt, we apply the encryption algorithm and we write back to the document the encrypted string. In order to be able to decrypt the documents, we took advantage of the fact that a MongoDB can store documents with completely different fields. More specifically, we created another collection in which we store a document that contains information about the field we encrypt and the encryption type. When we want to execute a query, we open that document and apply the right decryption to the right field in order to get the result of the query. Our method does not only support documents with simple fields, but also embedded fields.

Below are some examples from the documents we store in our database:

1. **Simple document, inserted using the Java Driver:**

```
{
      “_id” : {
         “$oid” : “59e97b212271ed377050ceb5”
  },
    “name” : “Michael”,
    “e-mail” : “mike@bulls.com”
}
```

Figure 2: Json example

2. **Document with encrypted field, using SHA-256:**

```
{
  “_id” : {
        “$oid” : “59e97c872271ed1f98e33e35”
   },
   “name” : “Michael”,
   “e-mail” : “ba742f7f8ef43237d1a4cca486e0486836cc3177b122a29afa
      42f89e5c575617”
 }
```

Figure 3: Json example with SHA-256 Encrypted Field

3. **Document with encrypted field, using BCrypt:**

```
{
  “_id” : {
        “$oid” : “59e97b212271ed377050ceb5”
  },
  “name” : “Michael”,
  “e-mail” : “$2a$10$RXsso1W1HUhKhKWpDQhuaOOx8taPXW2eLP003
     NZYOCbtsm2aXs0Za”
 }
```

Figure 4: Json example with BCrypt Encrypted Field

4. **Document from the collection with fields and encryption types:**

```
{
    “_id”: {
        “$oid”: “59e9785c50304f276cc34798”
    },
    “field”: “e-mail”,
    “enc”: “random”
}
```

Figure 5: Json example of Encrypted Field

3. MONGODB PRELIMINARIES

As we mentioned before, there are plenty different types of NoSQL Databases. One of them, widely known and used by the programming community is MongoDB [14]. One reason that explains the widespread use of MongoDB is the fact that this NoSQL database is free and open-source, published under the GNU Affero General Public License. This Non Relational Database is a document-oriented database. A document-oriented database contains data that are stored in adjustable, JSON-like documents, that give us the capability of storing documents varying in fields' content and an easily modified data structure. This document model is quite intuitive for developers to learn and use, while still providing a schema-less design, high efficiency and automatic scaling qualities which are now urgent needs for the era of information and cannot be satisfied by the traditional RDBMS systems. Many drivers have been developed in quite a few languages such as Python, Ruby, Scala etc. but in this thesis we chose to analyze and enhance the Java driver.

3.1 MongoDb Common Terms

3.1.1 JSON-like Documents

Each record in MongoDB is a document that consists of pairs of fields and values. This document is similar to a JSON-Object, which means that the value of a field may consist of an entire other document or even an array of other documents. This is an advantage that cannot be applied to RDBMS systems, and is known as embedded fields. Embedded documents are used to store relationships between data by storing the related data in a single document. This operation allows applications to have access to the related data and modify them in fewer steps. We provide you with such an example below:

```
{
  “firstname”: “John”,
  “lastName”: “Smith”,
  “age”: 25,
  “address”: {
    “streetAddress”: “21 2nd Street, New York”,
    “state”: “NY”,
    “postalCode”: “10021-3100”
  },
  “phoneNumbers”: [
   {
      “type”: “home”,
      “number”: “212 555-1234”
    },
    {
      “type”: “mobile”,
      “number”: “123 456-7890”
    }
  ],
  “children”: [],
  “spouse”: null
}
```

Figure 6: Json Example of human being

3.1.2 MongoDB Collections

MongoDB’s documents are managed and stored in Collections. Collections are similar to tables in RDBMS systems. However, in Mongo’s system we are not obligated to store documents with identical data structure. Each document may vary from the others stored in the same collection. For example, the document below could be stored in the same collection as the one described above.

```
{
  “firstName”: “Andriani”,
  “lastName”: “Triantafyllou”,
  “age”: 22,
  “address”: {
    “streetAddress”: “21 2nd Street”,
    “city”: “New York”,
    “state”: “NY”,
    “postalCode”: “10021-3110”
  },
  “phoneNumbers”: [
    {
      “type”: “home”,
      “number”: “212 1234-1234”
    },
    {
      “type”: “mobile”,
      “number”: “123 456-7899”
  }
  ]
}
```

Figure 7: Json Example no.2 of human being

As you can see, some fields such as “children”, “spouse”, etc have been omitted. These examples represent the schema-less design, which was introduced by the NoSQL Databases and is one of their greater innovations. A lot of people misunderstand the term “schema-less”. Schema-less doesn’t indicate that you don’t have to design the schema of your application, analyze it and comprehend how it works. The term refers to the fact that the documents stored, are not obligated to have a specific data structure, they can differ. Below, we will present the main differences between RDBMS and MongoDB schema .

Table 1: Schema Design Differences between MongoDB and RDBMS

RDBMS	**MONGODB**
Each table row has the same structure, columns	Documents’ Structure can Differ
Joins, Transactions are Supported	Joins, Transactions are not Supported
SQL Queries	No SQL queries, javascript
Fixed Schema	Dynamic Schema
Vertical Scaling	Horizontal Scaling

3.1.3 The MongoDB _id Field

All the documents stored in MongoDB are mandatory to be composed of the _id field [15]. This field acts as the primary key of the record. It can be either defined by the user, or be automatically generated by the MongoDB Driver. So for example, in the above human beings document representations figures 6, 7, MongoDB will add a 24 digit unique identifier to each document of the collection.

3.2 Java Driver

The driver we are about to examine is the Java Driver of MongoDB [16]. It's quite simple and we will present the fundamental steps to connect to a MongoDB[17], insert some documents and run queries on the database.

3.2.1 Connect to MongoDB instance stored in MongoDB online

The connection steps are described below :

- In line 1 an object of class MongoCredential is created, which stores the user's sign in information.
- In line 3 an object of class MongoClient is created, in order to initialize the Client with the user information given before.
- Following in line 6 an object of class MongoDatabase is created, storing the database requested, in this case the database db.

```
1 MongoCredential credential = MongoCredential.createCredential(
     usrnm, db, pswd.toCharArray());
2
3 MongoClient mongoClient = new MongoClient(new ServerAddress(srvr)
     , Arrays.asList(credential));
4 // Access database named db, providing owner's username usrnm and
      owner's password pswd and Server's Address srvr
5
6 MongoDatabase database = mongoClient.getDatabase(db);
7 // Access database db in order to access the collections stored
     in it
```

Figure 8: The steps to connect in MongoDB through the Java Driver

3.2.2 Insert documents

1. Single Document Insertion
   - In line 1 a Document is created with the embedded fields name.first, name.last and the field e-mail. Their values are "Angelina", "Jolie", "angelinaJ@gmail.com" respectively.
   - In line 3 this document is inserted in the MongoCollection.

```
1 Document document = new Document(“name”, new Document(“first”, “
     Angelina”).append(“last”, “Jolie”)).append(“e-mail”, “
     angelinaJ@gmail.com”);
2
3 mongoCollection.insertOne(document);
```

Figure 9: Insertion of a single Document in MongoDB

2. Multiple Documents Insertion
    - In line 1 a Document is created with the embedded fields “name.first”, “name.last” and the field “e-mail”. Their values are Angelina, Jolie, angelinaJ@gmail.com respectively.
    - In line 3 a Document is created with the embedded fields “name.first”, “name.last” and the field “e-mail”. Their values are “Keira”, “Knightley”, “keiraK@gmail.com” respectively.
    - In line 5 these documents are both inserted in the MongoCollection as an array of Documents.

```
1 Document document = new Document(“name”, new Document(“first”, “
     Angelina”).append(“last”, “Jolie”)).append(“e-mail”, ”
     angelinaJ@gmail.com”);
2
3 Document keira = new Document(“name”, new Document(“first”, “
     Keira”).append(“last”, “Knightley”)).append(“e-mail”, “
     keiraK@gmail.com”);
4
5 mongoCollection.insertMany((Arrays.asList(angelina, keira)));
```

Figure 10: Insertion of multiple Documents in MongoDB

3.2.3 Query

- In line 1 all the Documents of the collection are found and returned through the FindIterable[18] class.
- In line 4 all the Documents of the collection that include the field “e-mail” with the value “angelinaJ@gmail.com” are found and returned through the FindIterable [18] class.
- In line 7 all the Documents of the collection featuring the embedded field “name.last” with the value “Knightley” are found and returned through the FindIterable [18] class.

```
FindIterable<Document> results = collection.find();
//Find all the documents Stored

FindIterable<Document> result = collection.find(new Document("e-
   mail", "angelinaJ@gmail.com"));
//Find all the documents stored with the pair {field = "e-mail",
   value = "angelinaJ@gmail.com"}

FindIterable<Document> result2 = collection.find(new Document("
   name.last", "Knightley"));
//Find all the documents stored with the pair {embeddedfield = "
   name.last", value = "Knightley"}
```

Figure 11: Find Queries on a MongoDB through the Java Driver.

4. DOCUMENT'S FIELD ENCRYPTION APPROACH

In this thesis, we decided to implement our approach of this issue, MongoDB's lack of handling encrypted fields. So, two algorithms supporting field encryption have been developed. The encryption types used are BCrypt and SHA-256 Encryption. At this section, we will be describing and analyzing them in detail, while justifying our algorithm choices.

4.1 Define Encrypted Fields

In our Approach we support two different encryption types. The user is free to choose the encryption type to be used. So, the user has the potential to define the record fields by their names and the encryption to be performed on them. At this point, they can either choose to perform the SHA-256 encryption type or the BCrypt encryption to all the specified fields of a collection. This approach minimizes the continuous queries about the encryption type to be performed to each field of the Document given. These fields are stored in another collection in the database, in order to be always accessible to the system and result to the right performance of the operations on the database collection.

In order to achieve time enhancement and be competitive to the original Mongo Driver and its operations, the field collection which stores the specified encrypted fields is stored in a cache memory between our application and the database. The data are managed in a user friendly way. These data are necessary through the database's operations and the queries giving us access to them may vary depending on quite a few factors such as the database's size, etc. So, this choice eliminates such queries on the database, saving a lot of time. The cache is widely used during both the insertion of a document and the various queries on the Database's Collection, ensuring the right handling of the data.

4.2 Supporting SHA-256 Encryption

4.2.1 About SHA-256

SHA-2 (Secure Hash Algorithm 2) is implemented by a set of cryptographic hash functions, developed by the United States National Security Agency (NSA) and published in 2001 [1]. SHA-256 is part of this set. A cryptographic function is an algorithm that transforms a digital data into an other form which cannot identify the original input data and differs significantly. Usually, this output data has a fixed length, in SHA-256 case that's 256 bit. The algorithm is based on shift amounts and additive constants performed on a number of rounds. It provides a secure way to store data because the hash is quickly generated from the original data, but the original data can not be easily discovered from the hash, so it is not easy to crack. In other words, the inverse process hash-to original is almost impossible.

4.2.2 Insert Document

The insertion of plain Documents with no Encrypted Fields is still supported from the Driver. The Driver presented is an enhanced, enriched edition of the original one, supporting most of the operations in encrypted fields too. The encryption can be applied to both plain fields and embedded fields. Below, we present the algorithms' pseudocode for these settings.

1. **Insert Document with Encrypted Fields**

   The document is modified before being inserted. The encryption function splits the fields, examining each one separately. If the field name is included in the Client's cache, storing the Encrypted Fields, then the field's value is replaced by the SHA-

256 output of the original field value.

- In line 1 is defined the function insertOneHash, that inserts a Document in the Collection after modifying it properly.
- In line 7 is defined the function encryptIfNeeded2, that encrypts a specific field of a Document if it is necessary.
- In line 19 is defined the function encryptDocument, which examines each pair Field,Value of the Document and encrypts it if needed.

Pseudocode 1: Insertion algorithm for a single Document of SHA-256's Approach

```
1  FUNCTION insertOneHash(Document document)
2  {
3    INITIALISE doc TO the RETURN value of CALL encryptDocument(
         document,EMPTY_STRING)
4    INSERT doc to the collection
5  }
6
7  FUNCTION encryptIfNeeded2(String fieldName,String value,Entry
       <Field,Value> entry)
8  {
9    IF fieldName has to be encrypted
10   {
11     IF  fieldName has to be encrypted in SHA-256 way
12     {
13       INITIALIZE encoded TO SHA-256 encrypted form of value
14       SET Value of entry TO encoded
15     }
16   }
17 }
18
19 FUNCTION Document encryptDocument(Document doc, String pathD)
20 {
21   FOR each pair{field,value} in doc
22   {
23     INITIALIZE path TO “”
24     INITIALIZE fieldName TO field
25     IF  pathD is not Empty
26       SET path TO pathD.fieldName
27     ELSE
28       ADD fieldName TO path
29     INITIALIZE fieldValue TO value
30     IF fieldValue is a  String
31     {
32       INITIALIZE stringValue TO the string format of
             fieldValue
33       CALL encryptIfNeeded2(path, stringValue, field)
34     }
35     ELSE_IF fieldValue is an Integer
36     {
37       INITIALIZE integerValue TO the integer format of
```

```
             fieldValue
38         INITIALIZE stringValue TO the string format of
             intgerValue
39         CALL encryptIfNeeded2(path, stringValue, field)
40       }
41       ELSE_IF fieldValue is a Float
42       {
43         INITIALIZE floatValue TO the float format of fieldValue
44         INITIALIZE stringValue TO the string format of
             floatValue
45         CALL encryptIfNeeded2(path,stringValue, field)
46       }
47     }
48     RETURN doc
49   }
```

2. **Insert Document with Embedded Encrypted Fields**
   In case of embedded fields the Document's structure differs. In that case, so, the function encryptDocument is modified.
   - In line 1 is defined the function encryptDocument, which examines each pair field,Value of the Document and encrypts it if needed.
   - The function has an extra case now in line 12. This condition is added because of the Embedded Fields. In that case, the Value is an other one Document, so the function recursively calls itself with parameter the Value.

Pseudocode 2: Insertion algorithm for multiple Documents of SHA-256's Approach

```
1    FUNCTION Document encryptDocument(Document doc, String pathD)

2    {
3      FOR each pair{field,value} in doc
4      {
5        INITIALIZE path TO “”
6        INITIALIZE fieldName TO field
7        IF  pathD is not Empty
8          SET path TO pathD.fieldName
9        ELSE
10         ADD fieldName TO path
11       INITIALIZE fieldValue TO value
12       IF fieldValue is a Document
13       {
14         INITIALIZE tempDoc TO the Document format of fieldValue
15         CALL encryptDocument(tempDoc, path);
16       }
17       ELSE_IF fieldValue is a  String
18       {
19         INITIALIZE stringValue TO the string format of
             fieldValue
20         CALL encryptIfNeeded2(path, stringValue, field)
21       }
22       ELSE_IF fieldValue is an Integer
```

```
    {
      INITIALIZE integerValue TO the integer format of fieldValue
      INITIALIZE stringValue TO the string format of intgerValue
      CALL encryptIfNeeded2(path, stringValue, field)
    }
    ELSE_IF fieldValue is a Float
    {
      INITIALIZE floatValue TO the float format of fieldValue
      INITIALIZE stringValue TO the string format of floatValue
      CALL encryptIfNeeded2(path,stringValue, field)
    }
  }
  RETURN doc
}
```

4.2.3 Queries

1. **Find Query With No Parameter**
    - In line 10 is called the function decryptDocument which sets all the encrypted Values in SECRET_VALUE, as we see in its definition below.
    - The decryptDocument is recursive, due to the embedded fields which may contain the Document.

Pseudocode 3: Find all, in pseudocode, of SHA-256's Approach

```
1  FUNCTION List find()
2  {
3    INITIALIZE docs TO the RETURN value of CALL MongoDB's original find() function
4    IF docs is Empty
5      RETURN null
6    ELSE
7    {
8      INITIALIZE documentsList TO a List of Documents
9      FOR each document in docs
10       ADD the RETURN value of CALL decryptDocument(document, EMPTY_STRING) TO documentsList
11     RETURN documentsList
12   }
13 }
14
15
16
17 FUNCTION Document decryptDocument(Document document, String pathD)
18 {
19   FOR each pair{field ,value} in document
```

```
  {
    INITIALIZE fieldName TO  field
    INITIALIZE path TO “”
    IF  pathD is not Empty
      SET path TO pathD.fieldName
    ELSE
      ADD fieldName TO path
    INITIALIZE fieldValue TO value
    IF fieldValue is a Document
    {
      INITIALIZE tempDoc TO the Document format of fieldValue
      CALL decryptDocument(tempDoc, path);
    }
    ELSE
    {
      IF path is an encrypted field
      SET value to SECRET_VALUE
    }
  }
  RETURN document;
}
```

2. **Find Query With Document Parameter**

    - In line 3 is called the function encryptDocument that returns the Encrypted format of the Document given. In SHA-256 encryption, the output of a specific data remains always the same.
    - In line 4 is called the original find(document) of MongoDB's document. We search for the encrypted form of the Document, as the Documents are not stored in their original format.
    - In line 11 is called the function decryptDocument which sets all the encrypted Values in SECRET_VALUE.

Pseudocode 4: Find document matching values, in pseudocode, of SHA-256's Approach

```
 1 FUNCTION List  find(Document document)
 2 {
 3   SET doc TO the return value of CALL encryptDocument(
        document,EMPTY_STRING)
 4   INITIALIZE docs TO the RETURN value of CALL MongoDB's
        original find(doc) function
 5   IF docs is Empty
 6     RETURN null
 7   ELSE
 8   {
 9     INITIALIZE documentsList TO a List of Documents
10     FOR each document in docs
11       ADD the RETURN value of CALL decryptDocument(document,
            EMPTY_STRING) TO documentsList
12     RETURN documentsList;
13   }
14 }
```

4.3 Supporting BCrypt Encryption

4.3.1 About BCrypt

The BCrypt algorithm is based on a hashing function designed by Niels Provos and David Mazières[1]. It is based on the Blowfish cipher, which is a symmetric-key block cipher. This algorithm is mostly used in order to hash passwords. The fact that is uses salt, makes the BCrypt algorithm invulnerable against rainbow table attacks. A rainbow table is a precomputed table that is used in order to reverse a cryptographic hash function and crack passwords, like the BCrypt. The way the BCrypt algorithm is implemented, makes it resistant to brute-force search attacks. All the BCrypt hashed strings have the prefix “$2a$” or “$2b$” or “$2y$”. The rest of the hash string is a 128-bit salt encoded as 22 characters and a 184-bit hash value encoded as 31 characters. The hashed string also contains the cost parameter, which indicates the number of key expansion rounds. For example, if we have the hash string:

$2a$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy

the $2a$ indicates this is a BCrypt hashed string, the salt is N9qo8uLOickgx2ZMRZoMye and the hash value is IjZAgcfl7p92ldGxad68LJZdL17lhWy. The value 10$ is the cost parameter, we have 210 key expansion rounds.

4.3.2 Insert Document

BCrypt encryption can be applied to both plain and embedded fields, like the SHA-256 encryption. The document is modified by changing the value of the field the user chose to encrypt and then inserted into the database.

1. **Insert Document with Encrypted Fields**
   In order to modify the document, we need to know the field that we have to encrypt. So, we check if the collection that contains the sets <FieldToEncrypt, EncryptionType> includes field name requested. When found, we replace the value of this field with the hashed string that occurred after applying the BCrypt encryption on the original value.

   - In line 1 the function insertOneRandomPass is defined. This function inserts the document into the Collection after encrypting the requested field.
   - In line 6 the function encryptIfNeeded2 is defined. This function checks if the collection of the fields to encrypt contains a specific field of the Document, encrypts the value and sets the encoded value to the field.
   - In line 15 the function encryptDocumentRandomPass is defined. This function checks if we are dealing with a plain or an embedded field in order to give the right path to the encryptIfNeeded2 function.

Pseudocode 5: Insertion of Document with plain fields, in pseudocode, of BCrypt’s Approach

```
1 FUNCTION insertOneRandomPass(Document document) {
2   INITIALISE doc TO the RETURN_VALUE of encryptDocument(
       document, EMPTY_STRING)
3   INSERT doc into the collection
4 }
```

```

FUNCTION encryptIfNeeded2(String fieldName,String value,Entry
    <Field,Value> entry) {
  IF fieldName has to be encrypted {
    IF  fieldName has to be encrypted with BCrypt {
      INITIALIZE encoded TO BCrypt encrypted form of value
      SET Value of entry TO encoded
    }
  }
}

FUNCTION Document encryptDocument(Document doc, String pathD)
    {
  FOR each pair{field,value} in doc {
    INITIALIZE path TO “”
    INITIALIZE fieldName TO field
    IF  pathD is not Empty
      SET path TO pathD.fieldName
    ELSE
      ADD fieldName TO path
    INITIALIZE fieldValue TO value
    IF fieldValue is a  String {
      INITIALIZE stringValue TO the string format of
         fieldValue
      CALL encryptIfNeeded2(path, stringValue, field)
    }
    ELSE_IF fieldValue is an Integer {
      INITIALIZE integerValue TO the integer format of
         fieldValue
      INITIALIZE stringValue TO the string format of
         intgerValue
      CALL encryptIfNeeded2(path, stringValue, field)
    }
    ELSE_IF fieldValue is a Float {
      INITIALIZE floatValue TO the float format of fieldValue
      INITIALIZE stringValue TO the string format of
         floatValue
      CALL encryptIfNeeded2(path,stringValue, field)
    }
  }
  RETURN doc
}
```

2. **Insert Document with Embedded Encrypted Fields**
   If the document the user wants to insert into the database contains embedded fields, we need to modify the function encryptDocument, in order to support embedded fields. The function is defined below, with some changes in the way we handle the path of the field that is about to be encrypted.

Pseudocode 6: Insertion of Document with embedded fields, in pseudocode, of BCrypt's Approach

```
 1 FUNCTION Document encryptDocument(Document doc, String pathD)
 2 {
 3   FOR each pair{field ,value} in doc
 4   {
 5     INITIALIZE path TO “”
 6     INITIALIZE fieldName TO field
 7     IF  pathD is not Empty
 8       SET path TO pathD.fieldName
 9     ELSE
10       ADD fieldName TO path
11     INITIALIZE fieldValue TO value
12     IF fieldValue is a Document
13     {
14       INITIALIZE tempDoc TO the Document format of fieldValue
15       CALL encryptDocument(tempDoc, path);
16     }
17     ELSE_IF fieldValue is a  String
18     {
19       INITIALIZE stringValue TO the string format of
            fieldValue
20       CALL encryptIfNeeded2(path, stringValue, field)
21     }
22     ELSE_IF fieldValue is an Integer
23     {
24       INITIALIZE integerValue TO the integer format of
            fieldValue
25       INITIALIZE stringValue TO the string format of
            intgerValue
26       CALL encryptIfNeeded2(path, stringValue, field)
27     }
28     ELSE_IF fieldValue is a Float
29     {
30       INITIALIZE floatValue TO the float format of fieldValue
31       INITIALIZE stringValue TO the string format of
            floatValue
32       CALL encryptIfNeeded2(path,stringValue, field)
33     }
34   }
35   RETURN doc
36 }
```

As we observe in line 15, we recursively check whether we have reached to a field that maybe is the one the user wants to encrypt.

4.3.3 Queries

1. **Find Query With No Parameter,**
   The Find Query With No Parameter is exactly the same as in the SHA-256 encryption approach, Pseudocode 3 .

2. **Find Query With Document Parameter,**
   The Find Query With Document Parameter differs from the SHA-256 encryption approach, due to the fact that a specific value doesn't always have the same BCrypt encryption, because of the salt value. So, in that case the algorithm has to be adjusted properly.

   - In line 5 is called the function parseDocument. The function parses the Document, finds the encrypted fields and stores them to the 3rd parameter. Also a new Document is constructed, which is composed of the fields that do not require encryption.
   - In line 6 the original find(document) function of the MongoDB' driver is called. In this way we minimize the loops, restricting the documents to modify.
   - In line 12, the condition refers to the case that the Document given in the Find query is not composed of encrypted fields.
   - In line 18, the condition refers to the case that the Document given in the Find query is composed at least of one encrypted field.
   - In line 22, the condition refers to the fact that the Documents compared are composed of the same pairs<Field, Value>.

Pseudocode 7: Find Documents matching values, in pseudocode, of BCrypt's Approach

```
1  FUNCTION List find(Document document)
2  {
3    INITIALIZE fields TO a Map with Pairs<Field,Value>
4    INITIALIZE temp TO a new Document
5    CALL parseDocument(document,temp,fields,EMPTY_STRING)
6    INITIALIZE docs TO the RETURN value of CALL MongoDB's
         original find(doc) function
7    IF docs is Empty
8      RETURN null
9    ELSE
10   {
11     INITIALIZE documentsList TO a list of Documents
12     IF fields is Empty
13     {
14       FOR each tempDocument in docs
15         ADD the RETURN value of CALL decryptDocument(
               tempDocument,EMPTY_STRING) TO documentsList
16       RETURN documentsList
17     }
18     ELSE
19     {
20       FOR each tempDocument in docs
21       {
22         IF tempDocument matches to Document
23           ADD the RETURN value of CALL decryptDocument(
                 tempDocument,EMPTY_STRING) TO documentsList
24       }
25       RETURN documentsList
26     }
```

```
    }
}
```

5. TIME MEASUREMENTS

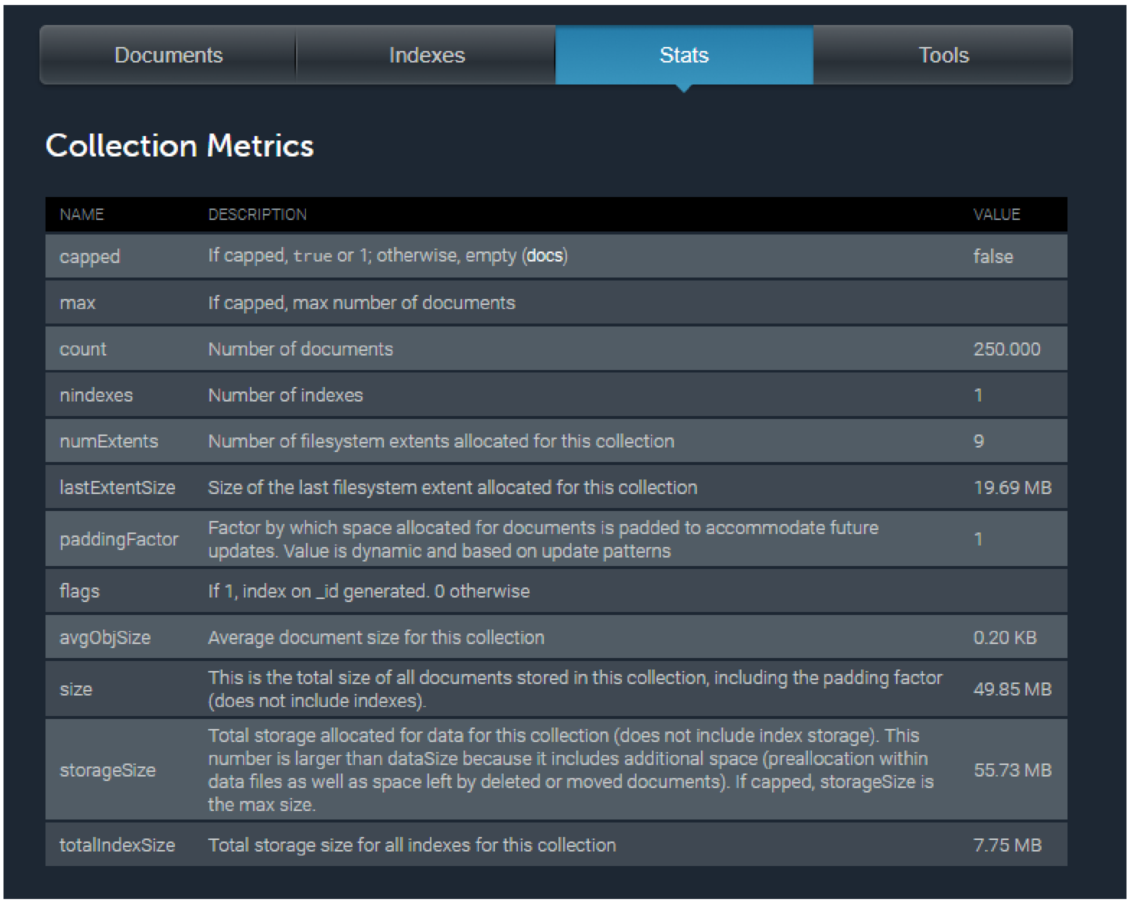
In this part, we are going to analyze the results of our implementation and compare our approach with the Java Driver's implementation.

5.1 About Test Environment

All the tests were run on the mLab[19]. mLab is a cloud database service that hosts MongoDB databases. The cloud providers that cooperate with mLab are Amazon, Google and Microsoft Azure. Due to the fact that we wanted to simulate realistic conditions, we chose a cloud database in order to test the limits of our implementation in two ways:

1. amount of time needed to encrypt all Documents
2. how latency affects the insertion time.

Specifically, we created and managed different amounts of Documents in order to examine each method's score. In our tests, we reached the number of 250.000 documents stored at a time, which size was 50MB. As one can see in figure 12 the mLab's environment is accurate and intuitive to comprehend. This was one of the reasons justifying our tests' environment choice. All the tests were run on the same Computer System and Internet Connection, in order to accomplish same circumstances to all of them.



Collection Metrics

NAME	DESCRIPTION	VALUE
capped	If capped, `true` or 1; otherwise, empty (docs)	false
max	If capped, max number of documents	
count	Number of documents	250.000
nindexes	Number of indexes	1
numExtents	Number of filesystem extents allocated for this collection	9
lastExtentSize	Size of the last filesystem extent allocated for this collection	19.69 MB
paddingFactor	Factor by which space allocated for documents is padded to accommodate future updates. Value is dynamic and based on update patterns	1
flags	If 1, index on _id generated. 0 otherwise	
avgObjSize	Average document size for this collection	0.20 KB
size	This is the total size of all documents stored in this collection, including the padding factor (does not include indexes).	49.85 MB
storageSize	Total storage allocated for data for this collection (does not include index storage). This number is larger than dataSize because it includes additional space (preallocation within data files as well as space left by deleted or moved documents). If capped, storageSize is the max size.	55.73 MB
totalIndexSize	Total storage size for all indexes for this collection	7.75 MB

Figure 12: mLab's Statistic Environment

For the visualization of the results, we used a free online program called “plot.ly” [20]. We have implemented two different chart types:

- Scatter Plot
- Line Plot

The Line Plot was used in order to instantly compare the implementations and the Scatter Plot for individual time measurements of each method.

5.2 Queries on Original Mongo Driver

Firstly, we run some test cases using the Original MongoDB Java Driver. These were used throughout our method’s evaluation as a comparison.

5.2.1 Insert Queries of Documents

Below, you can see the time measurements of the Insert Queries tests.

- In table 2 we present all the measurements thoroughly.
    1. The documents’ number varies from 500 to 250.000.
    2. Column 2 , of table 2, refers to the insertion of plain documents without embedded fields.
    3. Column 3, of table 2, refers to the insertion of documents with embedded fields.
- The figure 13 visualizes the measurements for the queries on plain documents.
- The figure 14 visualizes the measurements for queries on documents with embedded fields.

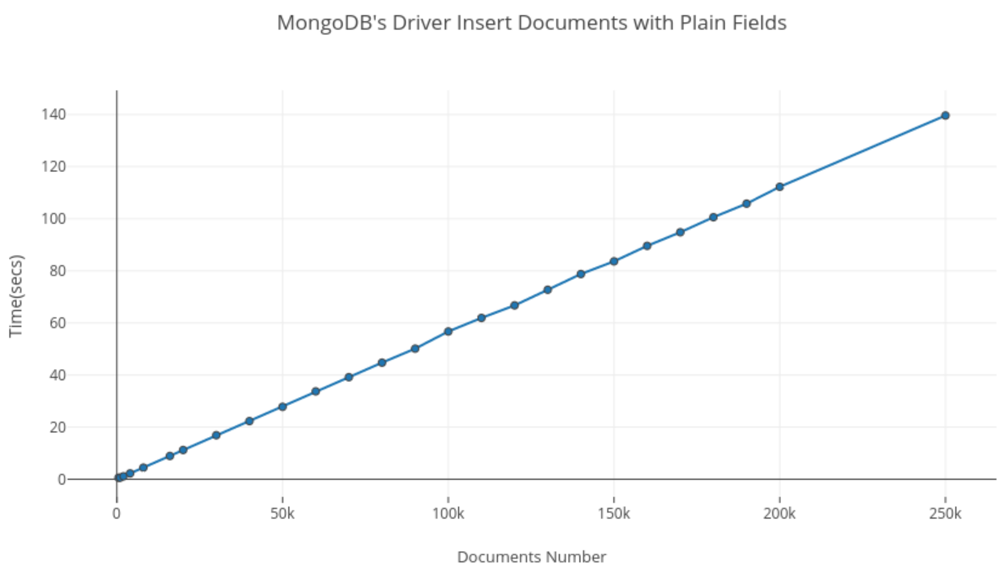


Figure 13: MongoDB’s Driver Insert Documents with Plain Fields

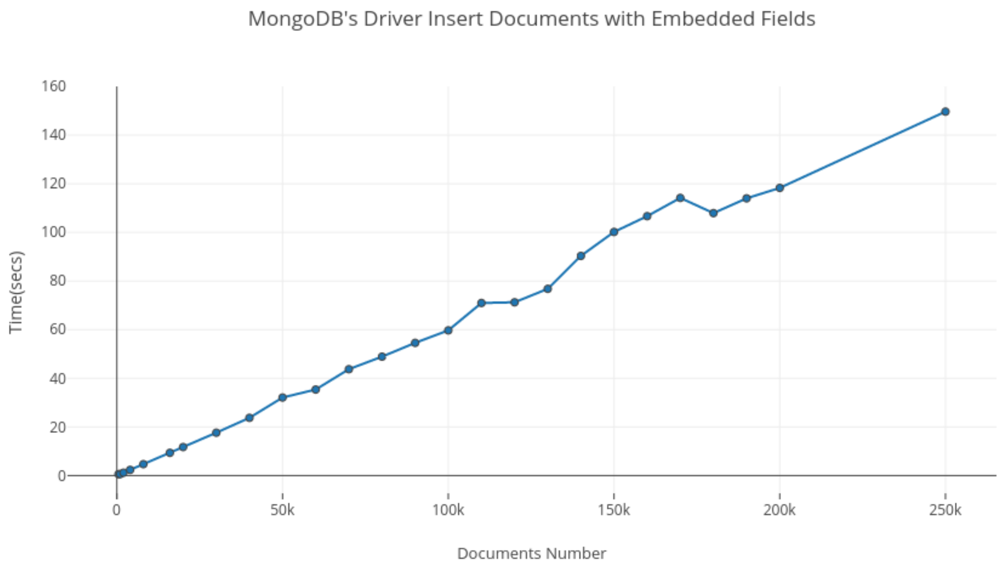


Figure 14: MongoDB's Driver Insert Documents with Embedded Fields

Observations:

- The duration of inserting documents with embedded fields exceeded that of plain documents, due to the bigger datasize.
- The plot of insertion of documents with plain fields exhibits a linear increase.
- The plot of insertion of documents with embedded fields exhibits a linear increase with some abnormalities, appearing on the diagram as rough edges.
- Those abnormalities occurred because of the network latency.
- In the small tests, for up to 30.000 documents, the insertion times are exactly the same.

Table 2: MongoDB's Insert Measurements

Insert Queries		
Number of Documents	Time (secs) for Documents with Plain Fields	Time (secs) for Documents with Embedded Fields
500	0.5730946	0.5816705
1000	0.5895274	0.5947747
2000	1.1122522	1.1722622
4000	2.238315	2.3365142
8000	4.446298	4.6910906
16000	8.927377	9.352967
20000	11.22777	11.720046
30000	16.896626	17.622036
40000	22.315216	23.741806
50000	27.78702	32.082764
60000	33.72613	35.375786
70000	39.170597	43.732872
80000	44.747826	48.89606
90000	50.125816	54.56426
100000	56.716427	59.703728
110000	61.912457	70.96251
120000	66.72994	71.24544
130000	72.721596	76.78131
140000	78.76392	90.32357
150000	83.63236	100.17131
160000	89.57758	106.662766
170000	94.80113	114.17041
180000	100.5821	107.920784
190000	105.77751	113.98962
200000	112.25198	118.2887
250000	139.61145	149.65392

As we can see from the above table:

- The average insertion time for a document without embedded fields is constant, around 0.000556 seconds, regrardless of the amount of documents we are inserting into the database .
- The insertion time for a document with embedded fields varies from 0.00054 to 0.00121 seconds.

5.2.2 Find All Queries of Documents

At this section are presented the time measurements of the FindAll Queries tests.

- In table 3 are presented all the measurements thoroughly.
    1. The documents' number varies from 500 to 250.000.
    2. Column 2, of table 3 refers to the find query of plain documents, without embedded fields.
    3. Column 3, of table 3 refers to the find query of documents with embedded fields.
- The figure 15 visualizes the measurements for queries on documents with plain fields.
- The figure 16 visualizes the measurements for queries on documents with embedded fields.

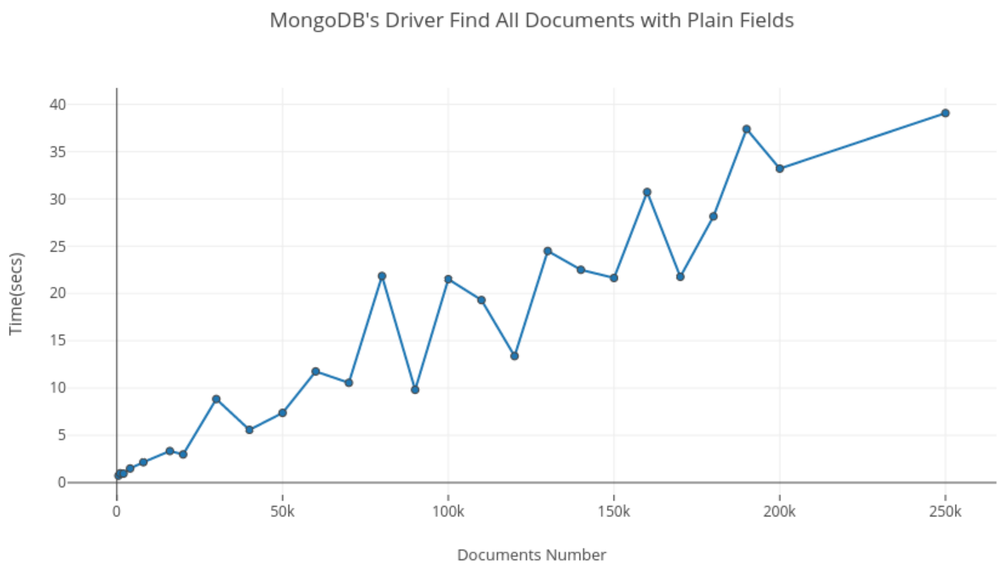


Figure 15: MongoDB's Driver Find All Documents with Plain Fields

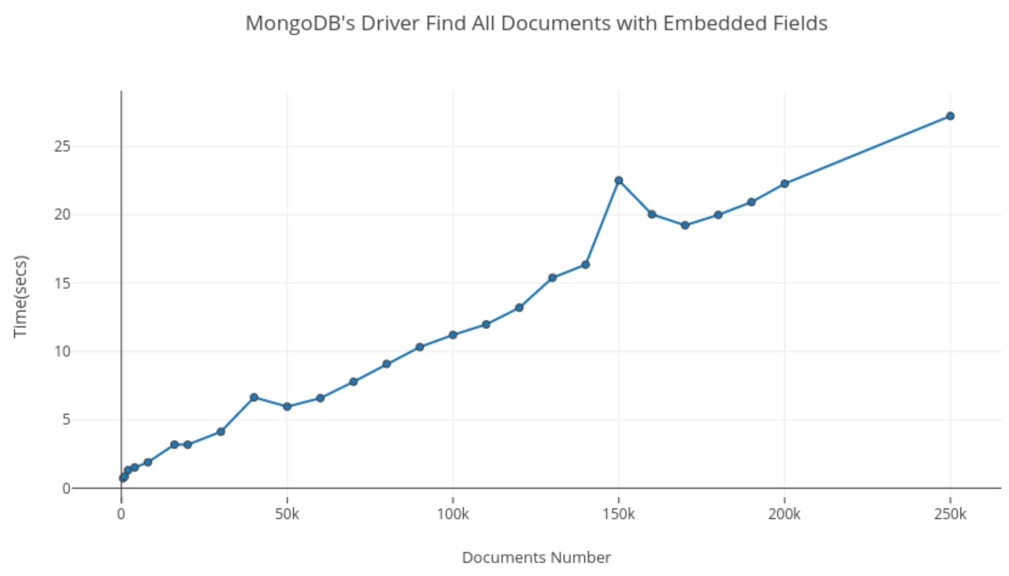


Figure 16: MongoDB's Driver Find All Documents with Embedded Fields

Observations:

- The plot of time needed to execute queries to documents without embedded fields exhibits a linear increase with some rough edges.
- These edges occur due to the network latency.
- The plot of time needed to execute queries to documents with embedded fields exhibits a linear increase, but with some rough edges.
- These edges occur due to the network latency.
- The duration of the find queries in documents with embedded fields does not always exceed that of the documents with plain fields. As we mentioned earlier, a document with embedded fields has a bigger size compared to a document with plain fields. This fact results to greater document insertion duration when we are dealing with documents with embedded fields. On the matter of queries though, the size of the document does not affect the querying process when it comes to small documents, like the ones we used in order to test our implementation. By observing the diagrams, we notice that sometimes the queries we run for documents without embedded fields took more time than the ones for documents with embedded fields, proving the previous statement. These plots also point out that the latency has a very important role in the time it takes to communicate with the database and process the data.

Table 3: MongoDB's Find All Documents Measurements

Find Queries		
Number of Documents	Time (secs) for Documents with Plain Fields	Time (secs) for Documents with Embedded Fields
500	0.73660636	0.7245615
1000	0.98596567	0.8598934
2000	0.9515232	1.3168386
4000	1.4927201	1.5118712
8000	2.159213	1.8968664
16000	3.3412073	3.1834989
20000	2.981193	3.1781032
30000	8.836872	4.1284575
40000	5.579202	6.6470017
50000	7.3707237	5.9651155
60000	11.757226	6.5864515
70000	10.55608	7.783245
80000	21.848125	9.090209
90000	9.81017	10.326397
100000	21.510502	11.210602
110000	19.305304	11.980717
120000	13.375557	13.204302
130000	24.490334	15.394913
140000	22.50495	16.354048
150000	21.636757	22.519304
160000	30.727268	20.033978
170000	21.761988	19.232533
180000	28.148558	19.99701
190000	37.385254	20.933147
200000	33.20038	22.276958
250000	39.06967	27.231556

- The query time for a document without embedded fields varies from 0.000147 to 0.000372 seconds.
- The query time for a document with embedded fields varies from 0.000113 to 0.000375 seconds.

As we can see, the latency can cause a slowdown to the querying process with the slowest query being three times slower than the fastest one. This happens because in order to execute queries, in our implementation, we used the FindIterable[18] function, which gets the documents one by one and runs the query on them. This process results to continuous connections to the database and disconnections, because our database is hosted by mLab, a cloud service. Those connections and disconnections are the main reason why we observe those abnormalities on the plots.

5.3 Queries on SHA-256 Encryption

5.3.1 Insert Queries of Documents with Encrypted Fields

Below are presented the time measurements of the Insert Queries tests of our SHA-256 Approach, Pseudocode 1.

- In table 5 are presented all the measurements thoroughly.
    1. The documents' number varies from 500 to 250.000.
    2. Column 2, of table 5, refers to the insert query of plain documents, without embedded fields.
    3. Column 3, of table 5, refers to the insert query of documents with embedded fields.
- The figures 17 ,18 visualize the measurements for the queries on plain documents and documents with embedded fields respectively.

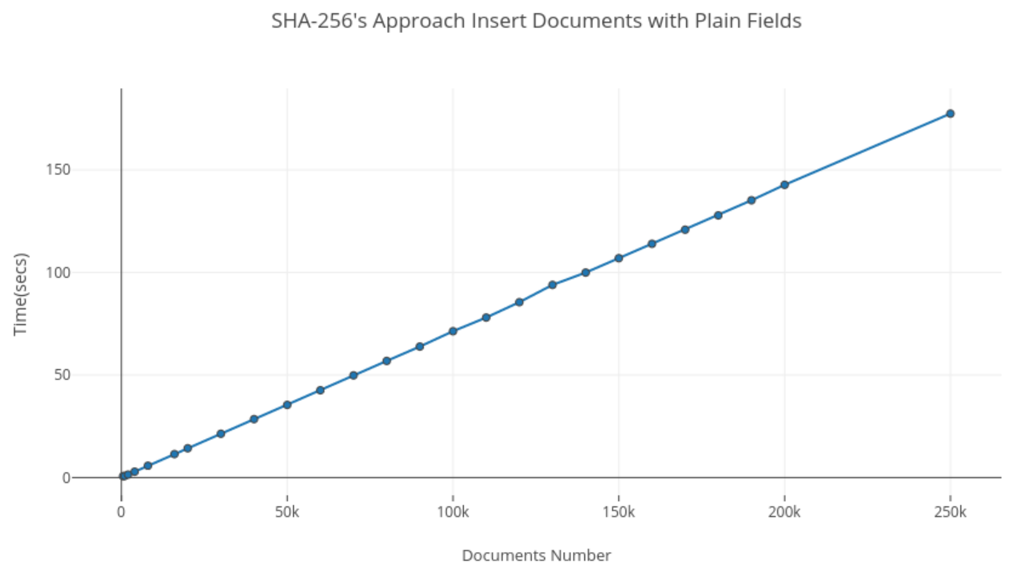


Figure 17: SHA-256's Approach Insert Documents with Plain Fields

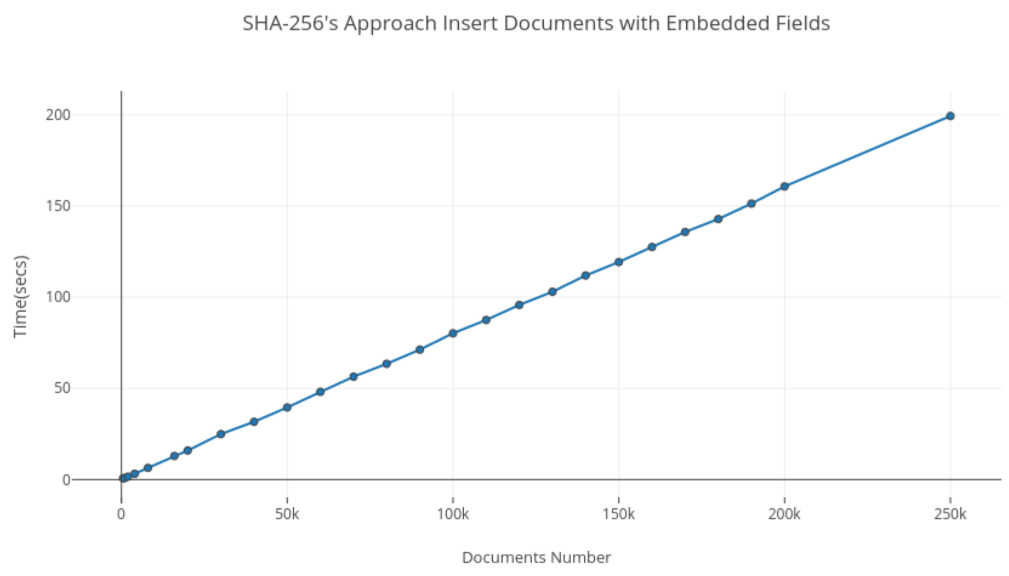


Figure 18: SHA-256's Approach Insert Documents with Embedded Fields

Observations:

- The duration of inserting documents with embedded fields exceeded once again that of plain documents, due to the bigger datasize.
- Both plots of insertion of documents with plain fields and with embedded fields exhibit a linear increase.
- In the small tests, for up to 50.000 documents, the insertion times are almost the same.
- At this point, we need to point out a situation we dealt with during the experiment period. MongoDB's Java Driver consists of two insert functions.

  1. insertOne function[21] : Inserts a single document into a collection.
  2. insertMany function[22] : Inserts multiple documents into a collection.

  The two of them should be exploited properly. At the start of test measurements, we exploited insertOne function, inserting the documents one by one. This choice resulted in achieving same time scores in SHA-256's Approach and MongoDB's Driver. However, we noticed that the SHA-256's time scores increased in an extreme way when we increased the datasize. Consequently, we exploited insertMany function and the result was remarkable. In table 4 we present a sample of the time decrease.

Table 4: The insertOne function compared to insertMany

Insert Queries				
Number of Documents	Time (secs) of insertOne for MongoDB's Driver	Time (secs) of insertMany for MongoDB's Driver	Time (secs) of insertOne for SHA-256's Approach	Time (secs) of insertMany for SHA-256's Approach
1000	148	0,59	152	0,77
2000	317	1,11	317	1,43

Table 5: SHA-256's Approach Insert Measurements

Insert Queries		
Number of Documents	Time (secs) for Documents with Plain Fields	Time (secs) for Documents with Embedded Fields
500	0.686565	0.73590267
1000	0.767477	0.819133
2000	1.4297721	1.6111319
4000	2.8627377	3.1608565
8000	5.767674	6.3819146
16000	11.440591	12.951159
20000	14.283775	15.95993
30000	21.358059	24.928192
40000	28.489304	31.699194
50000	35.416405	39.543854
60000	42.56149	48.123077
70000	49.812492	56.435375
80000	56.865116	63.46296
90000	63.86848	71.23979
100000	71.353096	80.192635
110000	78.01235	87.49852
120000	85.50201	95.75311
130000	93.93346	102.99113
140000	99.94496	111.88866
150000	106.95854	119.26789
160000	113.97095	127.529175
170000	120.8392	135.75302
180000	127.82629	142.8367
190000	135.18794	151.31757
200000	142.71826	160.72505
250000	177.43808	199.31969

- The query time for a document without embedded fields varies from 0.000709 to 0.00136 seconds.

- The query time for a document with embedded fields varies from from 0.000797 to 0.00146 seconds.

SHA-256's Approach VS MongoDB's Java Driver:

At this point we visualized the comparison between the MongoDB's Original Java Driver and the SHA-256's Approach we implemented.

1. The figure 19 refers to the Insert Queries of Plain Documents.
   - Both lines exhibit a linear increase.
   - The MongoDB's method is quicker, the line is always above in the plot graph.
   - However, their distance is quite small considering the datasize methods are dealing with.
2. The figure 20 refers to the Insert Queries of Documents with Embedded Fields.
   - Plot lines exhibit a linear increase in insertion of Documents with embedded fields, too.
   - The MongoDB's method is slightly quicker, as you could see the line is always above in the plot graph.
   - However, their distance is quite small considering the datasize methods are managing.
   - This distance is caused by time needed to complete the encryption of the list of Documents that are about to be inserted. As it was described earlier, each document is parsed, splitted and modified properly.

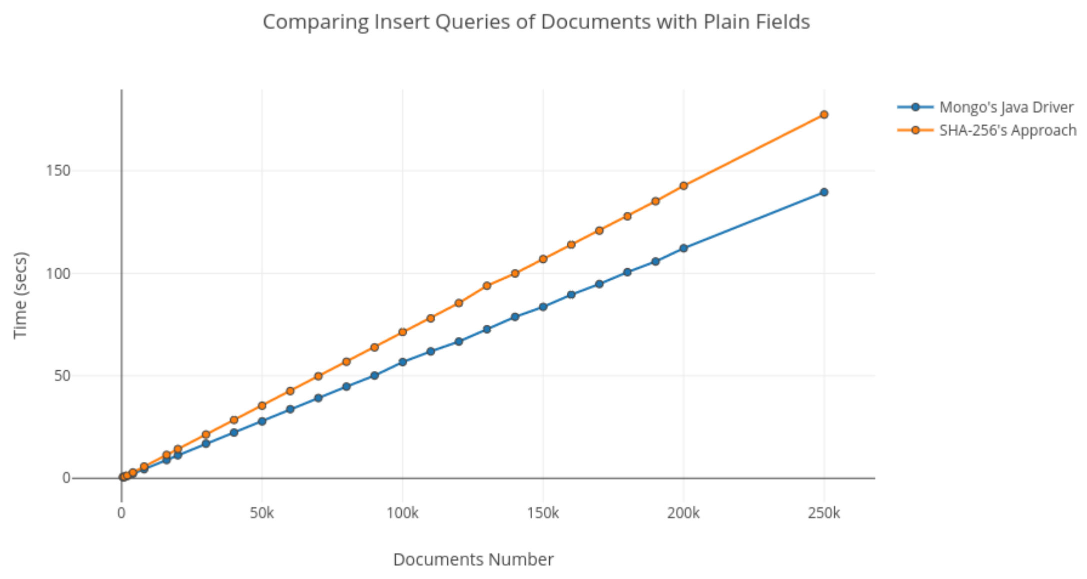


Figure 19: Insert Queries of Plain Documents, SHA-256's Approach compared to MongoDB's Driver

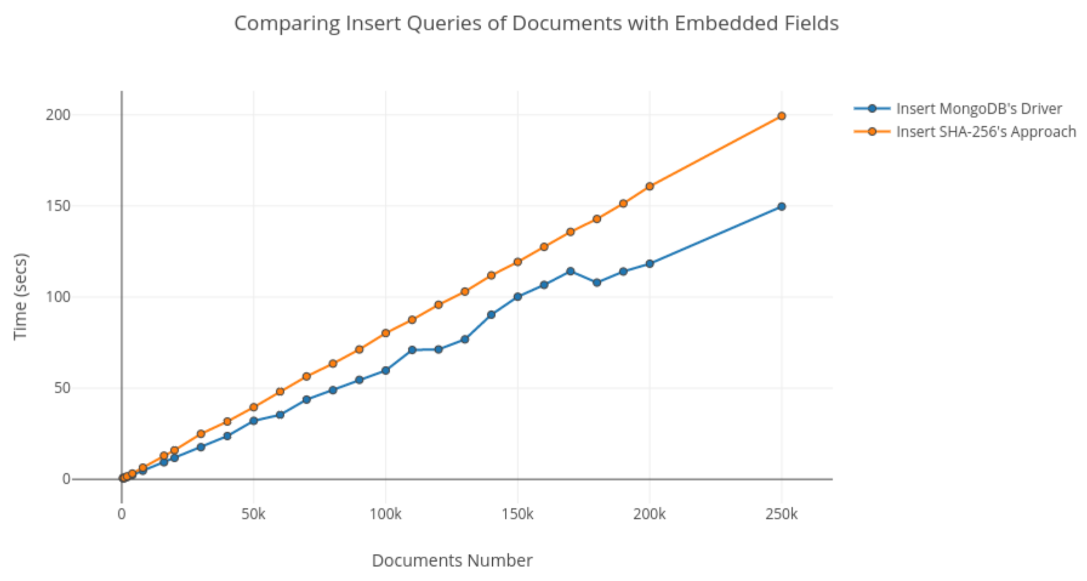


Figure 20: Insert Queries of Documents with Embedded Fields, SHA-256's Approach compared to MongoDB's Driver

5.3.2 Find All Queries of Documents with Encrypted Fields

It has been referred that the encrypted values are not revealed at the find function.

At this point are presented the time measurements of the Find All Queries tests of our SHA-256 Approach, Pseudocode 3.

- In table 6 are presented all the measurements thoroughly.
    1. The documents' number varies from 500 to 250.000.
    2. Column 2, of table 6, refers to the find query of plain documents, without embedded fields.
    3. Column 3, of table 6, refers to the find query of documents with embedded fields.
- The figures 21 ,22 visualize the measurements for the queries on plain documents and documents with embedded fields respectively.

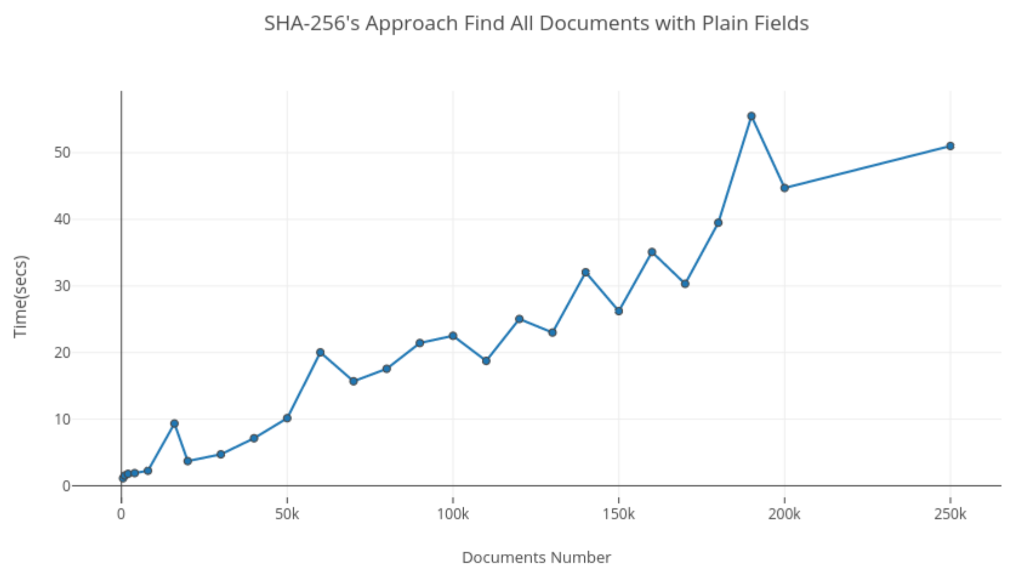


Figure 21: SHA-256’s Approach Find All Documents with Plain Fields

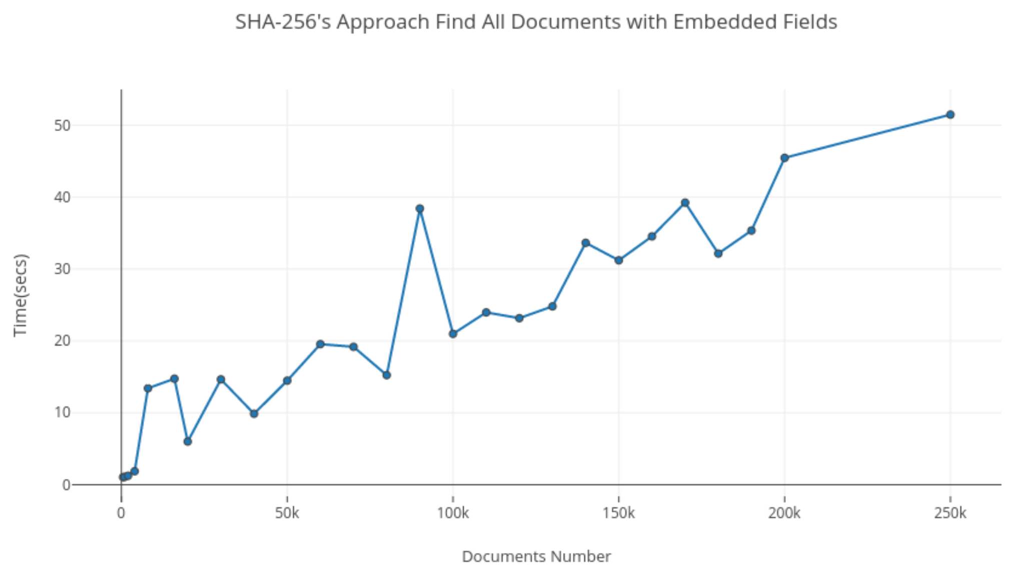


Figure 22: SHA-256’s Approach Find All Documents with Embedded Fields

Table 6: SHA-256's Approach Find All Documents Measurements

Find Queries		
Number of Documents	Time (secs) for Documents with Plain Fields	Time (secs) for Documents with Embedded Fields
500	1.1322337	1.0521961
1000	1.5329167	1.0891377
2000	1.7948279	1.2429496
4000	1.9028713	1.8752402
8000	2.257894	13.40974
16000	9.348973	14.747366
20000	3.7049177	6.0010247
30000	4.7231293	14.650656
40000	7.1344132	9.867553
50000	10.1499405	14.473209
60000	20.029806	19.54934
70000	15.686469	19.1773
80000	17.562069	15.2384205
90000	21.434359	38.409718
100000	22.51756	20.97913
110000	18.748234	23.962803
120000	25.043268	23.162762
130000	23.00232	24.800764
140000	32.0544	33.640625
150000	26.225332	31.206146
160000	35.097347	34.534893
170000	30.322437	39.228428
180000	39.49263	32.146206
190000	55.512093	35.34592
200000	44.708668	45.45475
250000	51.00995	51.468918

Observations:

- The query time for a document without embedded fields varies from 0.000204 to 0.00226 seconds.
- The query time for a document with embedded fields varies from 0.0002058 to 0.0021 seconds.
- In find queries, we do not have a linear increase, there are many abnormalities.
- Those abnormalities are caused by the network latency, as connections and disconnections occure during the testing and due to the FindIterable[18] class, as referred above.

SHA-256’s Approach VS MongoDB’s Java Driver:

Follows the visualization of the comparison between the MongoDB’s Original Java Driver and the SHA-256’s Approach find method we implemented.

The figures 23, 24 refer to the Find Queries of documents with and without embedded fields, respectively.

- Both lines show abnormalities due to the network latency and the connections / dis-connections to the database which occur from FindIterable’s[18] implementation.
- The MongoDB’s method is quicker, the line is always above in the plot graph.
- However, their distance is quite small considering the datasize methods are dealing with.
- This distance is due to the decryption time of the list of Documents to be shown. As it was described, each document is parsed, splitted and modified properly in order to not reveal it’s encrypted value.

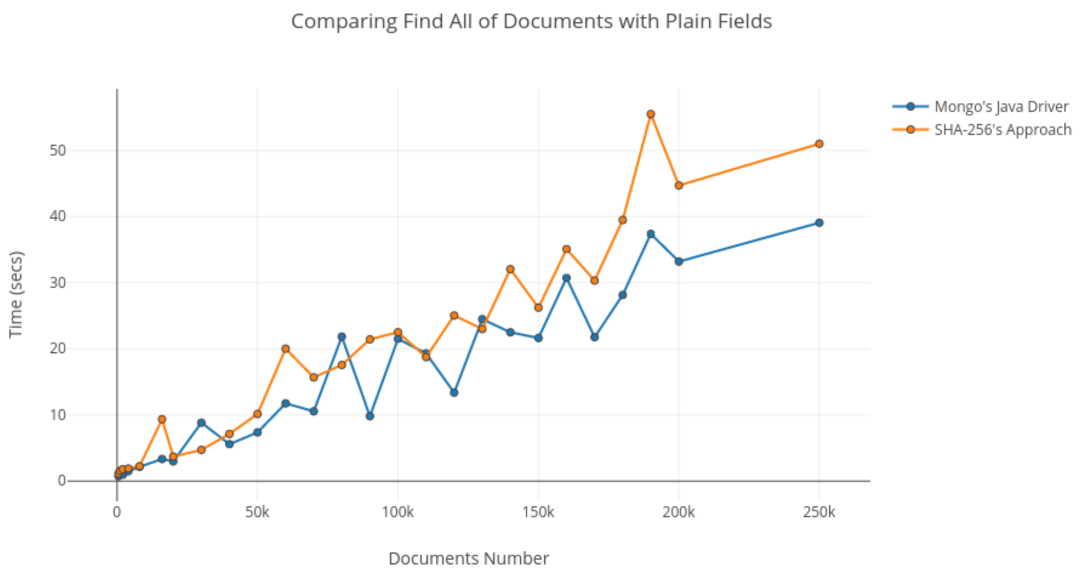


Figure 23: Find Queries of Plain Documents, SHA-256’s Approach compared to MongoDB’s Driver

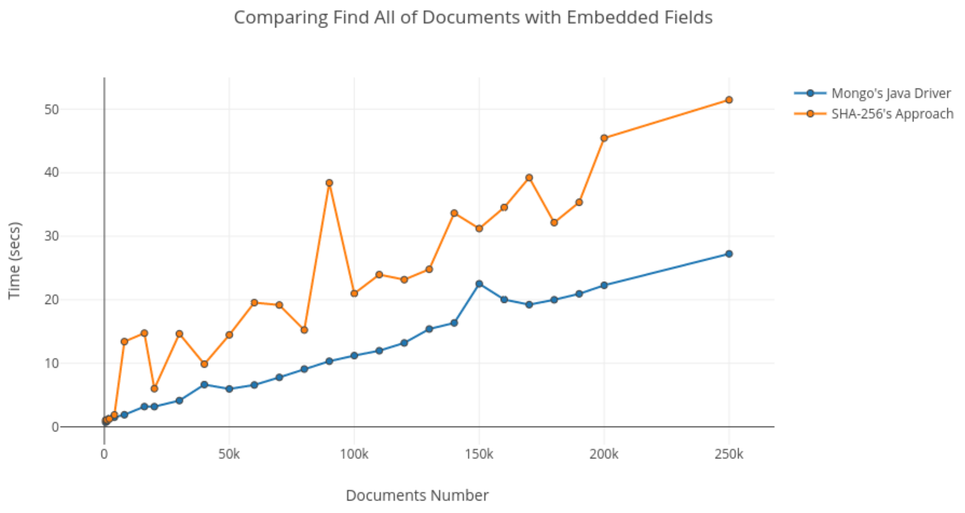


Figure 24: Find Queries of Documents with Embedded Fields, SHA-256's Approach compared to MongoDB's Driver

5.3.3 Find Queries Matching Values of Documents

Throughout the experiment process we also run some tests of find queries with parameters. Accurately, we run tests of different combinations, such as :

- "value matches unencrypted field",
- "value matches encrypted field",
- "combination of the previous two",
- "queries for embedded fields", etc.

We came to the conclusion that both methods, SHA-256's and MongoDB's Driver behave almost the same. There is a logic explanation about this behavior. In the SHA-256's approach, all the documents are finally stored in the database maintaining the encrypted form of the values. In our find implementation, Pseudocode 4, we call MongoDB Driver's find function with the encrypted form of the Document given as parameter. SHA-256's hashing method results, always, in the same output value. So, we query the database using a Document with the same value structure as the ones stored in it. The hashing method that SHA-256 provides is rapid, resulting in unnoticed time duration.

In table 7 you could see a sample of these time measurements. They refer to a database with 250.000 documents stored and present the average time duration of the queries. The differences are due to some factors such as:

- The number of values given to the query, specifies the document asked, eliminating the options. Increasing that number decreases the query time.
- Documents with embedded fields have a bigger size compared to documents with plain fields.

Table 7: Sample of Measurements. Find Queries Matching Different Combinations of Field Values

Find Queries Matching Values	
Query Type	Time (secs) for Find Query
Match Non Encrypted Value	1,03
Match Encrypted Plain Value	1,05
Match Encrypted Embedded Value	1.41
Match Combination of non encrypted, encrypted plain values	0,38
Match Combination of non encrypted, encrypted embedded values	0,41

5.4 Queries on BCrypt Encryption

5.4.1 Insert Queries of Documents with Encrypted Fields

- In table 8 are presented all the measurements of our BCrypts' Approach, Pseudo-code 5, thoroughly.
  1. The documents' number in this case varies from 10 up to 4.000.
  2. Column 2, of table 8, refers to the insert query of plain documents, without embedded fields.
  3. Column 3, of table 8, refers to the insert query of documents with embedded fields.
- The figures 25 ,26 visualize the measurements for the queries on plain documents and documents with embedded fields respectively.

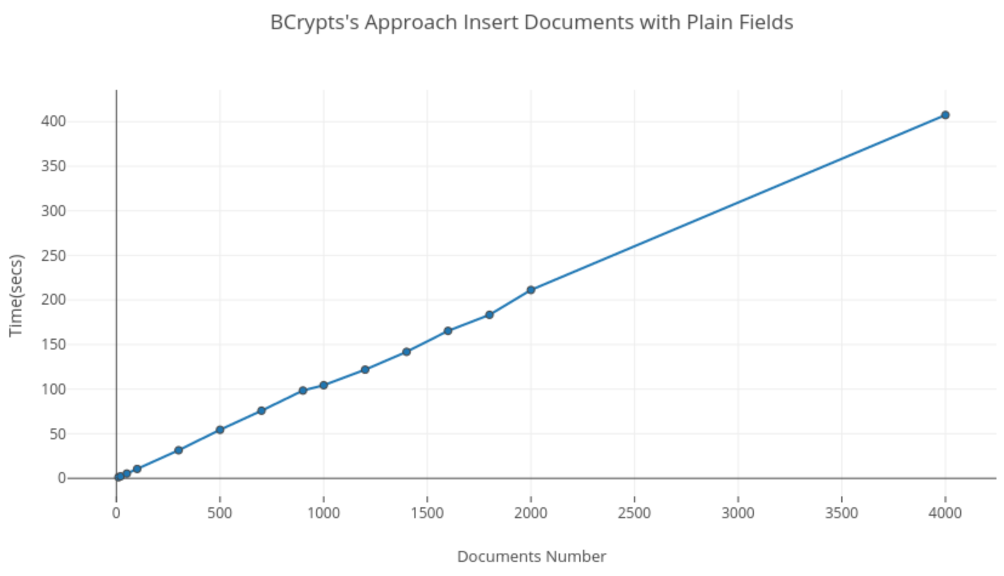


Figure 25: BCrypt's Approach Insert Documents with Plain Fields

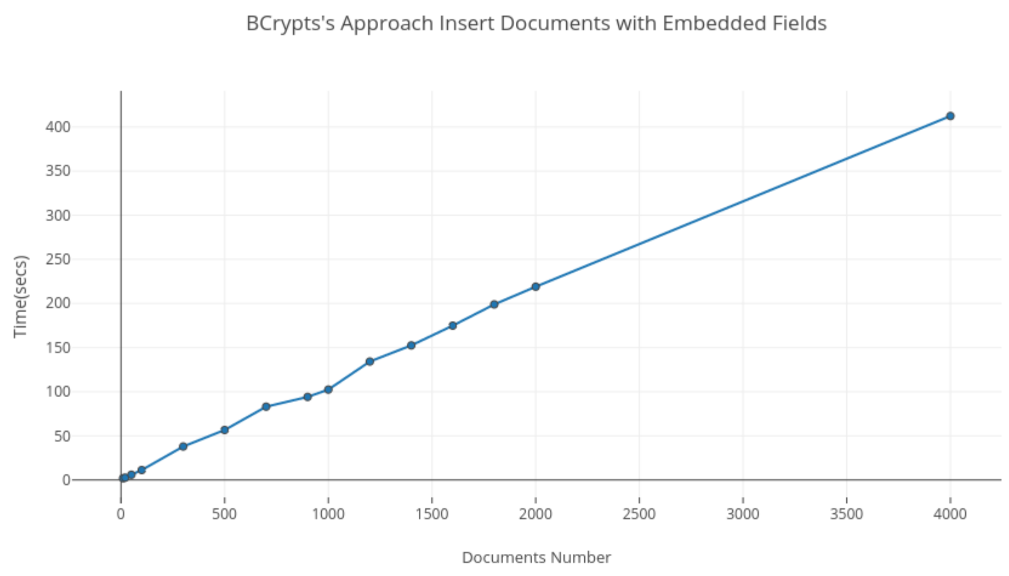


Figure 26: BCrypt’s Approach Insert Documents with Embedded Fields

Table 8: BCrypt’s Approach Insert Measurements

Insert Queries		
Number of Documents	Time (secs) for Documents with Plain Fields	Time (secs) for Documents with Embedded Fields
10	1.2195458	1.97198
20	2.2082877	2.6226175
50	5.3310995	5.9294763
100	10.54379	11.246323
300	31.495565	37.920063
500	54.42532	56.639286
700	75.834045	83.02905
900	98.46125	94.08907
1000	104.407	102.40931
1200	121.850105	134.20966
1400	141.87135	152.50159
1600	165.37697	174.87648
1800	183.35474	198.94284
2000	211.24365	218.98067
4000	407.44736	412.40356

Observations:

- The query time for a document without embedded fields varies from 0.1018 to 0.121 seconds.
- The query time for a document with embedded fields varies from from 0.1024 to 0.197 seconds.

BCrypt vs SHA-256, MongoDB Driver:

Follows the visualization of the comparison between the MongoDB’s Original Java Driver the SHA-256’s Approach and the BCrypt’s Approach find method we implemented.

The figures 27, 28 refer to the Insert Queries of documents with and without embedded fields, respectively.

- MongoDB’s, SHA-256’s methods are almost the same. Their plot lines converge.
- BCrypt is much slower than the others.

Balancing the trade offs between query delay and security

After running a big amount of tests, we came to the conclusion that the SHA-256 encryption can compete with the Java Driver implementation in the matter of both insertions and queries. On the contrary, the BCrypt encryption has a very poor performance compared to the other two methods. So why would someone use the BCrypt encryption as a valid security option when its performance is at first glance a very poor one?

On this moment, the issue of the trade-offs is raised. Considering the fact that some of the information stored in databases is very sensitive, for example sets like <Username, Password> or credit cards credentials, the applications that use and store this kind of data have to make a decision. Most of the time, this decision translates into extra security or better time performance.

When it comes to big data storage, id est the application struggles to simultaneously store a very large amount of data in a database within reasonable time limits and by default including encrypted fields, an encryption with a high insertion speed is the most appropriate method. Those insertions are called batch inserts and in our implemenation are carried out by the insertMany function, by applying the SHA-256 encryption, which provides the reasonable time limits we referred to before.

The main reasons we are using this encryption type are that SHA-256:

- is a quick encryption method as one can observe by looking at the previous plots
- is securing the data in a way that one cannot reverse the hashing and cannot access the original value of the encrypted field
- is securing the data in a way that even if the whole database is stolen, one cannot decrypt the encrypted fields without having the key, so basically, the data are useless

Apparently, not all applications are supposed to need to store such large amounts of data, but are more demanding in matters of security. Such applications or applications’ modules are e.g. a login system where the users provide the app with a username and a password and the back end is responsible to complete the verification of the username and the password in order to allow the users enter their accounts.

In this case, where we are handling passwords, a simple hash function is considered an inadequate security approach. An extra safeguard is necessary in order to ensure the users that their passwords cannot be cracked. Apart from malware and phising, which are nowadays the most common ways to ”steal” someone’s password, there are three more advanced attacks that can be used in order to crack a password. Those are:

- dictionary attacks
- brute force attacks
- rainbow table attacks

BCrypt, as we mentioned before, is a method that adds a salt, a random data, as an additional input to a one-way hash function in order to secure the data against dictionary and rainbow table attacks. This factor makes the BCrypt encryption an approach that offers additional security to the data, and like we already mentioned, this additional security is crucial to some applications.

This algorithm is mostly used in order to secure passwords and when we are handling a single password, in terms of MongoDB, we are actually handling one single document with one - or more, depends on the need of the application - encrypted field.

By observing the plots in figures 27 and 28, we realize that although the insertion times for documents using BCrypt as their type of encryption are 10 times slower than those of MongoDB's Java Driver and SHA-256, when it comes to a single document, the difference in insertion times is not that sensible. If we take into consideration the network delay and the I/O with the database, the delay of the BCrypt in comparison with the two other methods is acceptable, and at the same time, the users are offered a more secure encryption for the most important field, which is the password.

Overall, we presume that in order to have more efficient insertions, we have to partly sacrifice the protection of an encrypted field value against all three advanced attacks and in order to have a safeguard against those attacks, we have to deal with slower insertions of the records into the database. Our approach offers a complete solution to this problem because we provide the user with both options, and they get to decide whether security or quick insertions is what matters to them the most, as our implementation supports both SHA-256 and BCrypt encryption.

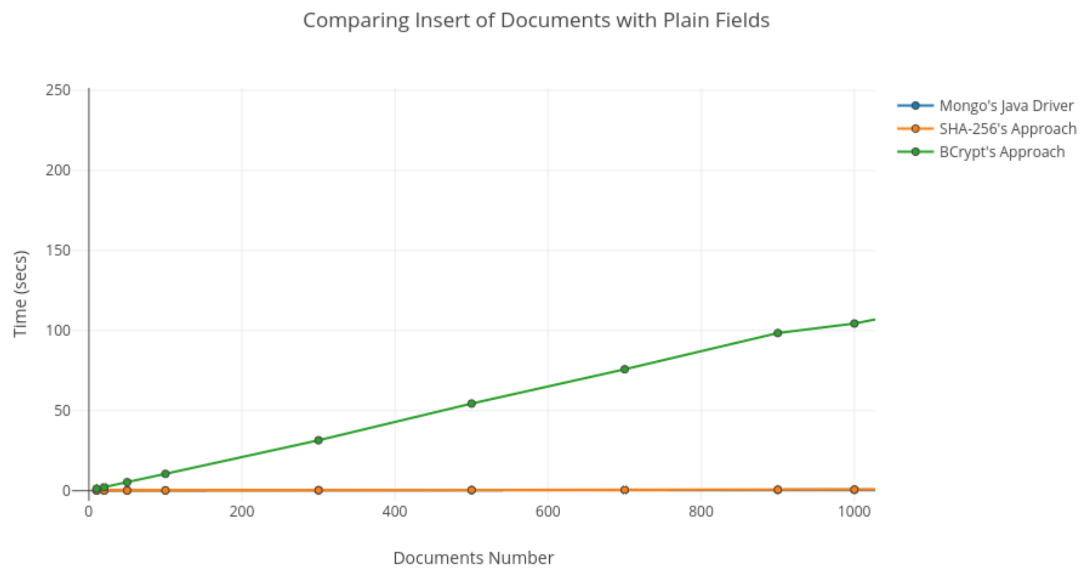


Figure 27: Insert Queries of Plain Documents, Comparing SHA-256's Approach, BCrypt's Approach, MongoDB's Driver

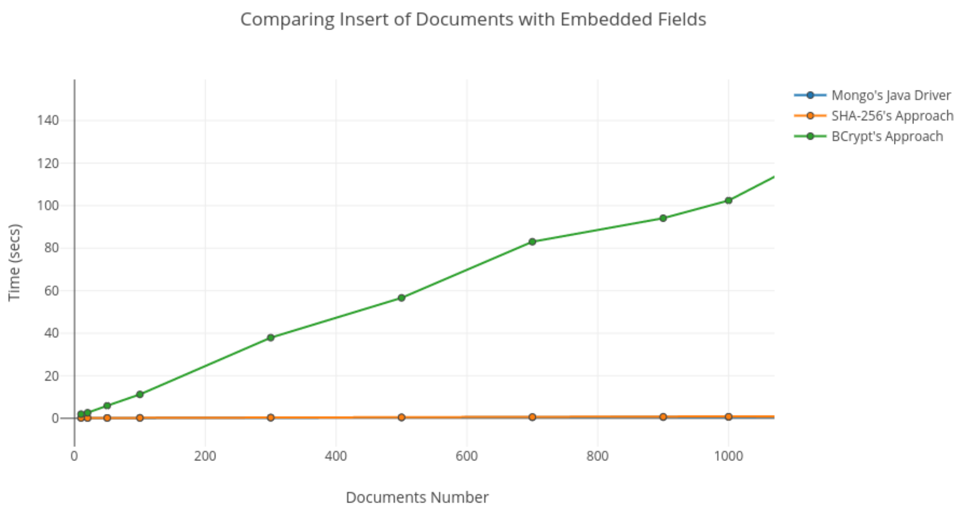


Figure 28: Insert Queries of Documents with embedded fields, Comparing SHA-256's Approach, BCrypt' Approach, MongoDB's Driver

5.4.2 Find All Queries of Documents with Encrypted Fields

It has been referred that the encrypted values are not revealed at this function.

- In table 9 are presented all the measurements of find method, Pseudocode 3, thoroughly.
    1. The documents' number in this case varies from 10 up to 4.000.
    2. Column 2, of table 9, refers to the insert query of plain documents, without embedded fields.
    3. Column 3, of table 9, refers to the insert query of documents with embedded fields.
- The figures 29, 30 visualize the measurements for the queries on plain documents and documents with embedded fields respectively.

Table 9: BCrypt's Approach Find All Documents Measurements

Find Queries		
Number of Documents	Time (secs) for Documents with Plain Fields	Time (secs) for Documents with Embedded Fields
10	0.15611355	0.11285697
20	0.16469336	0.16001631
50	0.16176115	0.2054268
100	0.32398608	0.38630733
300	0.6301775	1.2132382
500	0.81100154	1.4928799
700	0.7767368	1.391824
900	0.7737809	1.2900754
1000	0.8568008	1.669696
1200	1.0047315	1.2995803
1400	1.0407746	1.5793337
1600	1.0328734	1.7288271
1800	3.5929008	1.5828195
2000	1.6137465	1.422204
4000	2.4182215	2.5644693

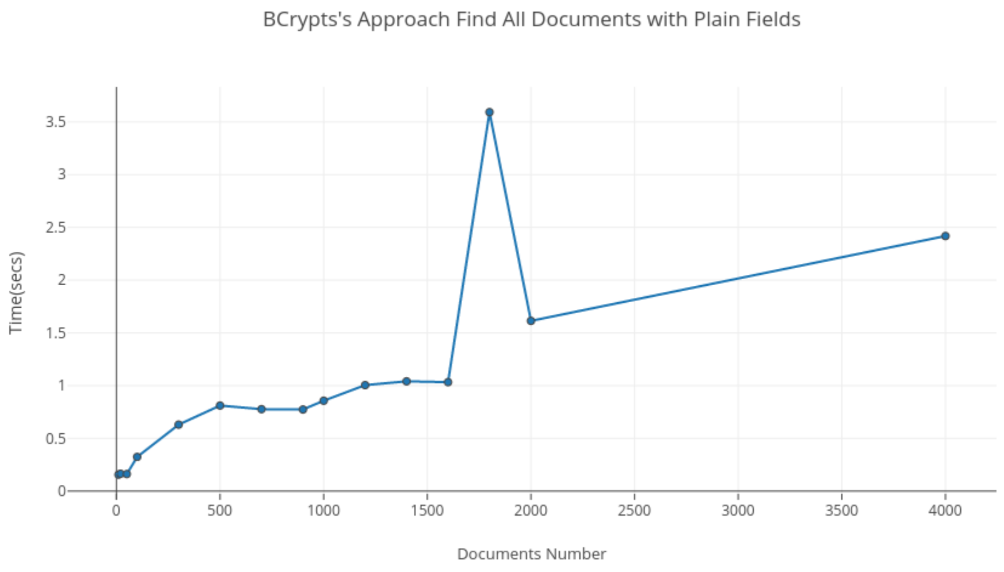


Figure 29: BCrypt's Approach Find All Documents with Plain Fields

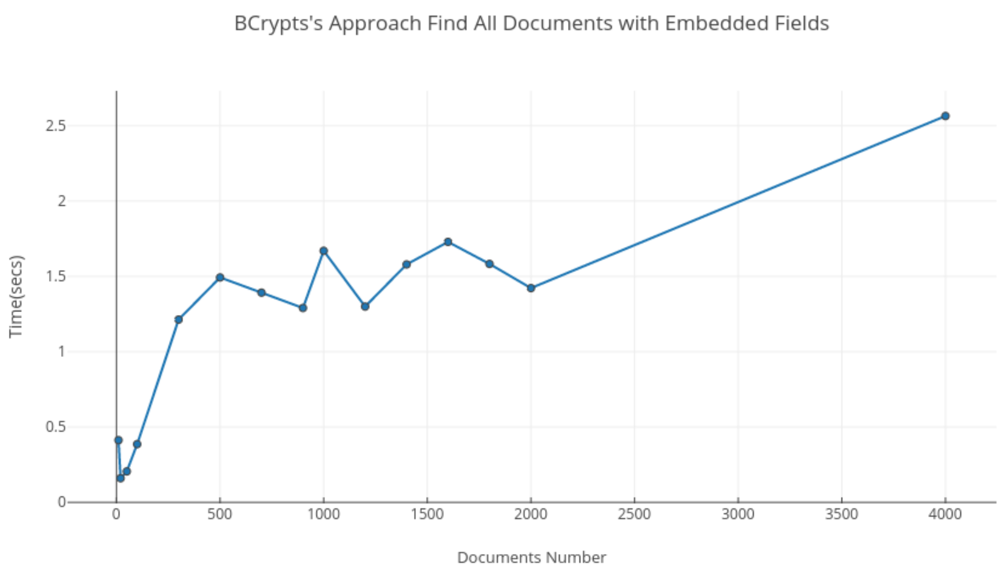


Figure 30: BCrypt’s Approach Find All Documents with Embedded Fields

Observations:

- The query time for a document without embedded fields varies from 0.0007 to 0.015 seconds.
- The query time for a document with embedded fields varies from from 0.001 to 0.011 seconds.

BCrypt vs SHA-256, MongoDB Driver: Follows the visualization of the comparison between the MongoDB’s Original Java Driver the SHA-256’s Approach and the BCrypt’s find method we implemented.

The figures 31, 32 refer to the Find Queries of documents with and without embedded fields, respectively.

- Both lines show abnormalities due to the network latency and the connections / disconnections to the database which occur from FindIterable’s[18] implementation.
- However, their distance is quite small considering the datasize methods are dealing with.
- The MongoDB’s method is quicker, the line is always above in the plot graph. This distance is due to the decryption time of the list of Documents to be shown. As it was described, each document is parsed, splitted and modified properly in order to not reveal it’s encrypted value.

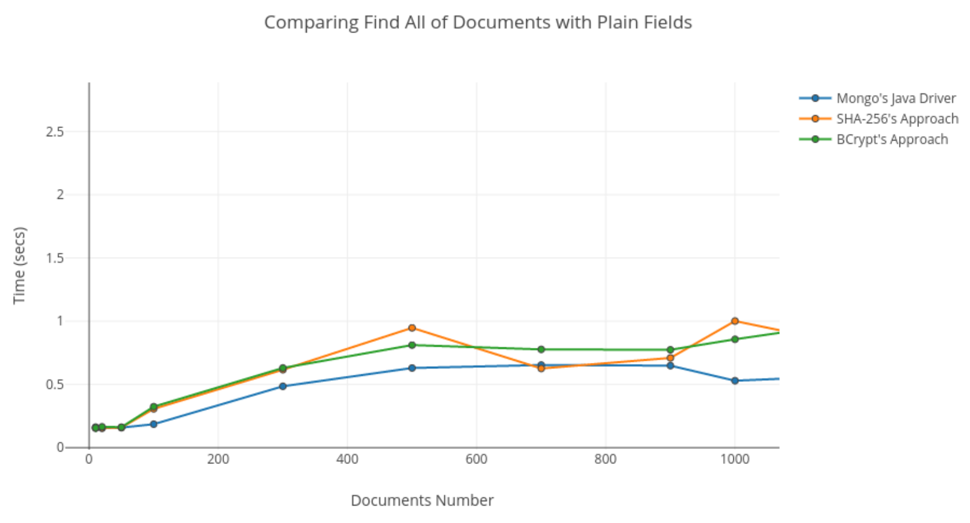


Figure 31: Find Queries of Plain Documents, Comparing SHA-256's Approach, BCrypt's Approach, MongoDB's Driver

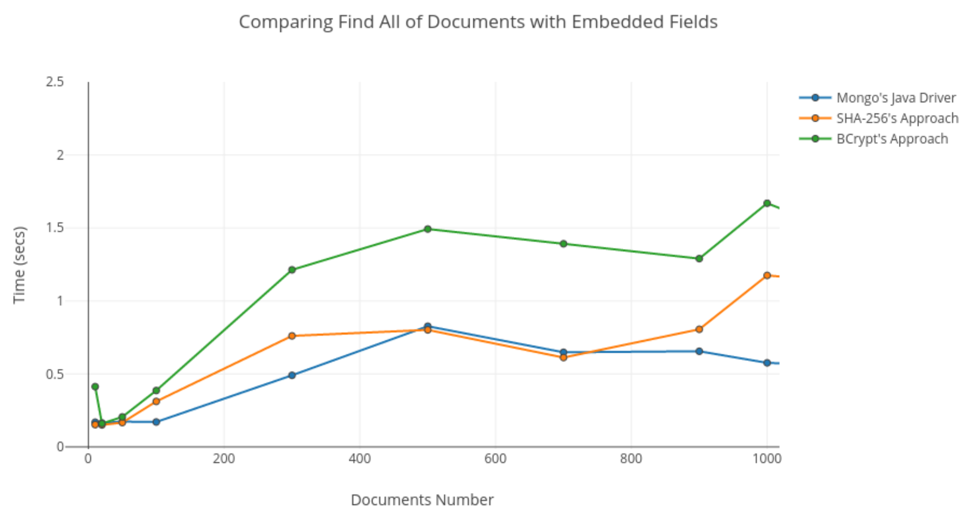


Figure 32: Find Queries of Documents with embedded fields, Comparing SHA-256's Approach, BCrypt's Approach, MongoDB's Driver

5.4.3 Find Queries Matching Values of Documents

We came to the conclusion that our BCrypt's Encryption method and MongoDB's Driver do not behave the same. In most times, BCrypt's measurements were quite longer. However, this can be explained of the implementation, Pseudocode 7. In the BCrypt's approach, as in SHA-256's, all the documents are finally stored in the database with the encrypted form of the values. BCrypt's hashing method results, almost always, in a different output value for a particular input value. So, in our find implementation, firstly, we query the database using a Document constructed of the non encrypted field values given. In that case we examine each one of these Documents matching the encrypted fields. In case non-encrypted values are not given in the query, we parse through all the documents stored in the database. In any case, the number of documents to examine can vary depending on the database's size. In conclusion, queries duration depends on the factors described above showing wide time differences. In most measurements, the time duration was slightly quicker than the insert duration.

In table 10 you could see a sample of these time measurements. They refer to a database with 1000 documents stored and present the average time duration of the queries. Below are described some observations:

- The number of values given to the query, specifies the document asked, eliminating the options. Increasing that number decreases the query time.
- If the Document given as a parameter, is composed of a field encrypted in the database then because of the implementation all the database is exported. Each document is parsed to check if the field's value matches with the given on the query.

Table 10: Sample of Measurements 2. Find Queries Matching Different Combinations of Field Values

Find Queries Matching Values	
Query Type	Time (secs) for Find Query
Match Non Encrypted Value	0,03
Match Encrypted Plain Value	102,3
Match Encrypted Embedded Value	103,1
Match Combination of non encrypted, encrypted plain values	0,2
Match Combination of non encrypted, encrypted embedded values	0,25

6. CONCLUSION

In this thesis, we have implemented a middle ware software that enriches the original MongoDB Java Driver in order to support insertions and searches on JSON-like documents that contain encrypted fields. After thoroughly analyzing the advantages and disadvantages of a wide variety of encryption modes, we decided to include in our implementation two encryption types, the SHA-256 and the BCrypt. So, we implemented the appropriate methods to support such practices.

To commence, our implementation focused on introducing these services providing the user with the option to choose between the two encryption types. After successfully developing the necessary algorithms for insertion of a single document, insertion of multiple documents at once and search queries matching one or more fields' value (in case of search parameter's absence, we return all the documents) we proceeded with some optimizations.

On what concerns the SHA-256 encryption, those optimizations resulted in achieving competitive insertion and query times, in comparison with the original MongoDB Java Driver. On the other hand, the BCrypt encryption, even though it does not compete with the SHA-256 operations, because of the nature of the algorithm (multiple hashing rounds in addition to salt), it offers additional security to multiple threats. On that grounds, it is up to the user to counterbalance the pros and cons of each method and pick the one that suits better to their needs.

In conclusion, the implementation offers the potential to be further extended and provide the user with more querying options like range queries, apart from the matching queries we already offer.

TERMINOLOGY TABLE

Σχεσιακή Βάση Δεδομένων	Relational database
Μη Σχεσιακή Βάση Δεδομένων	Non-relational database
NoSQL βάση	NoSQL database
Κρυπτογράφηση	Encryption
Αλγόριρθμος	Algorithm
Κατακερματισμός	Hashing
Ερώτημα	Query
Εγγραφή τύπου JSON	JSON document
Συμμετρική και ασύμμετρη κρυπτογράφηση	Symmetric and Asymmetric encryption
Ενσωματωμένο πεδίο	Embedded field
Κρυπτογράφημα	Cipher
Επίθεση στο Rainbow table	Rainbow table attacks
Βίαιη επίθεση	Brute force attacks
Επίθεση DDoS	DDoS attacks
Υπηρεσίες νέφους	Cloud services

ABBREVIATIONS, ACRONYMS

JSON	JavaScript Object Notation
NoSql	Not Only Sql
RDBMS	Relational DataBase Management System
RM	Relational Model
SHA-2	Secure Hash Algorithm 2
Mongo	MongoDB

REFERENCES

[1] Gilbert, Henri, and Helena Handschuh. ”Security analysis of SHA-256 and sisters.” In International workshop on selected areas in cryptography, pp. 175-193. Springer, Berlin, Heidelberg, 2003.[Accessed: Aug 22, 2017].

[2] “Spring’s Security BCrypt Class” BCrypt implements OpenBSD-style Blowfish password hashing function. [Online]. Available: https://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/crypto/bcrypt/BCrypt.html. [Accessed: Oct 15, 2017].

[3] “DigestUtils Class” Implementation of Operations to simplify common MessageDigest tasks. [Online]. Available: https://commons.apache.org/proper/commons-codec/apidocs/org/apache/commons/codec/digest/DigestUtils.html. [Accessed: Oct 15, 2017].

[4] “Spring’s Security BCryptPasswordEncoder” Implementation of PasswordEncoder that uses the BCrypt strong hashing function. [Online]. Available: https://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/crypto/bcrypt/BCryptPasswordEncoder.html.[Accessed: Oct 15, 2017].

[5] Tian, Xingbang, Baohua Huang, and Min Wu. ”A transparent middleware for encrypting data in MongoDB.” In Electronics, Computer and Applications, 2014 IEEE Workshop on, pp. 906-909. IEEE, 2014. [Accessed: May 20, 2017].

[6] Codd, Edgar F. The relational model for database management: version 2. Addison-Wesley Longman Publishing Co., Inc., 1990. [Accessed: September 18, 2017].

[7] Leavitt, Neal. ”Will NoSQL databases live up to their promise?.” Computer 43, no. 2 (2010). [Accessed: Aug 25, 2017].

[8] “How to Pick Database” [Online]. Available: http://www.zdnet.com/article/rdbms-vs-nosql-how-do-you-pick/. [Accessed: Sept 23, 2017].

[9] Moniruzzaman, A. B. M., and Syed Akhter Hossain. ”Nosql database: New era of databases for big data analytics-classification, characteristics and comparison.” arXiv preprint arXiv:1307.0191 (2013).[Accessed: Aug 22, 2017].

[10] “Compare Databases”. [Online]. Available: https://infocus.emc.com/april_reeve/big-data-and-nosql-the-problem-with-relational-databases. [Accessed: Oct 18, 2017].

[11] “Database Differences” [Online]. Available: https://www.loginradius.com/engineering/relational-database-management-system-rdbms-vs-nosql. [Accessed: Oct 22, 2017].

[12] Popa, Raluca Ada, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan,”CryptDB: protecting confidentiality with encrypted query processing.”. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pp. 85-100. ACM, 2011. [Accessed: May 15, 2017].

[13] Ge, Tingjian, and Stan Zdonik,”Answering aggregation queries in a secure system model.”. In Proceedings of the 33rd international conference on Very large data bases, pp. 519-530. VLDB Endowment, 2007. [Accessed: May 15, 2017].

[14] “MongoDB’s Definiton” [Online]. Available: https://www.mongodb.com/what-is-mongodb. [Accessed: Aug 25, 2017].

[15] “MongoDB Document’s structure” [Online]. Available: https://docs.mongodb.com/manual/core/document/#document-structure. [Accessed: Aug 18, 2017].

[16] “MongoDB Java Driver” [Online]. Available: https://mongodb.github.io/mongo-java-driver. [Accessed: Aug 18, 2017].

[17] “MongoDB’s Connection Tutorial 2” [Online]. Available: https://www.w3resource.com/mongodb/mongodb-java-connection.php. [Accessed: Aug 25, 2017].

[18] “FindIterable Class” MongoDB’s Java Driver FindIterable. Available: http://mongodb.github.io/mongo-java-driver/3.6/javadoc/com/mongodb/client/FindIterable.html. [Accessed: Aug 15, 2017].

[19] “Mlab, MongoDb’s Service” [Online Program]. Available: https://mlab.com/. [Accessed: Jun 15, 2017].

[20] “Online Graph Maker” [Online Program]. Available: https://plot.ly/create/. [Accessed: Oct 15, 2017].

[21] “insertOne function” MongoDB’s Java Driver insertOne function. Available: https://docs.mongodb.com/manual/reference/method/db.collection.insertOne/. [Accessed: Aug 15, 2017].

[22] “insertMany function” MongoDB’s Java Driver insertMany function. Available: https://docs.mongodb.com/manual/reference/method/db.collection.insertMany/. [Accessed: Aug 15, 2017].

[23] “Migrating Relation Database to Document” [Online]. Available: https://www.slideshare.net/matkeep/migrating-from-relational-databases-to-mongodb. [Accessed: Oct 15, 2017].