# DOCUMENT-BASED DATABASES IN PLATFORM SW ARCHITECTURE FOR SAFETY RELATED EMBEDDED SYSTEM

Nahid Seidi

***THIS THESIS IS PRESENTED AS PART OF DEGREE OF BACHELOR OF SCIENCE IN ELECTRICAL ENGINEERIN DEPARTMENT***

15 ECTS

Blekinge Institute of Technology- Scania AB

*This page intentionally left blank*

# ABSTRACT

The project is about the investigation on Document-Based databases, their evaluation criteria and use cases regarding requirements management, SW architecture and test management to set up an (ESLM) Embedded Systems Lifecycle Management tool.

The current database used in the ESLM is a graph database called Neo4j, which meets the needs of the current system.

The result of studying Document databases turned to the decision of not using a Document database for the system. Instead regarding the requirements, a combination of Graph database and Document database could be the practical solution in future.

# Table of Contents


ABSTRACT ..... 3

INTRODUCTION ..... 5

1.1 Problem statement ..... 5

1.2 Scope of the thesis ..... 5

1.3 Outline of the thesis ..... 5

BACKGROUND AND RELATED WORKS ..... 6

2.1 Background ..... 6

2.2 Related works ..... 6

2.3 Comparison to this work ..... 8

DESIGN AND IMPLEMENTATION ..... 9

3.1 NoSQL databases ..... 9

3.2 Document-based databases ..... 9

3.3 MongoDB ..... 11

*I. An overview on MongoDB ..... 11*

*II. MongoDB implementation ..... 14*

3.4 OrientDB ..... 20

RESULTS AND CONCLUSION ..... 21

REFERENCE LIST ..... 25

# INTRODUCTION

## 1.1 Problem statement

The aim of this project is to propose a solution regarding the best matching database for setting up an Embedded Systems Lifecycle Management tool. The implementation on a sample of data is done, which is going to be explained in following chapters.

## 1.2 Scope of thesis work

The scope of the project is around the software development concept. The focus is on the matter of having efficient yet flexible database that can handle the requirements management. This project of course had some implementation on database as well since the incorporation of both use case study and implementation is needed for this work.

## 1.3 Outline of the thesis

There are four main chapters in this thesis, which are to be explained fully. Following concepts are going to be explained in this report.

- *Background and related work*
  - Clarifies the problem and gives more information about it.
  - Previous solutions given to the problem and attempts to solve it.
  - Comparison between this solution and previous solutions.
- *Design and implementation*
  - Requirements in order to fulfill the requisite of situation.
  - The design and the solution in details for this project.
  - Implementation of the solution considering the limits and requirements.
  - The testing and verification of the database in order to make sure of proper functionalities.
- *Results and conclusion*
  - Summary of the work and the goal that was reached.
- Future work
  - Things that were not in the scope of this project and are left for improvement in future.

# Background And Related Works

## 2.1 Background

Scania AB is a Sweden-based manufacturer of heavy trucks and buses, as well as industrial and marine engines. The company's activities comprise five business areas. The Trucks area develops, manufactures and sells trucks with a gross vehicle weight of more than 16 tons (Class 8), intended for long distance, construction and distribution haulage, as well as public services. The Bus and coaches area is concentrated on buses and coaches for use as tourist coaches, as well as in urban and intercity traffic. The Engines area includes industrial and marine engines that are used in electric generator sets, construction and agricultural machinery, as well as in ships and pleasure boats. The Service area provides service-related products for transport and logistic companies. The Financial services include such services as loan financing, leases and insurance solutions. Scania AB has operations in approximately 100 counties, and it is headquartered in Södertälje (Stockholm), Sweden.

In this project, Scania develops a tool for system and architecture recovery. This tool takes production data and source code, as input and produces an architectural model of the Electronic Control Units (ECU), as output.
A large part of Scania ECUs is developed in-house. In particular, the platform SW is developed in-house, and it has functionality similar to a real-time operating system.

In this project, the aim is to investigate how the current implementation can be further developed to meet future requirements on safety, availability, reliability, failure management, etc.

The main focus is on investigating Document-based databases to evaluate criteria and use cases for current project.

## 2.2 Related works

Since the current software used for the project is a Graph database called Neo4j, a comparison of Graph databases and Relational databases has been done to evaluate Neo4j as the database used in the project.

Following is an overview on graph databases and current database in Scania.

A graph database stores data in a graph. Data is stored in nodes, which have properties; nodes are organized by relationships, which also have properties. Nodes and relationships are fundamental units forming a graph.

Figure 1 shows the meta-model of data in stored Neo4j in Scania.

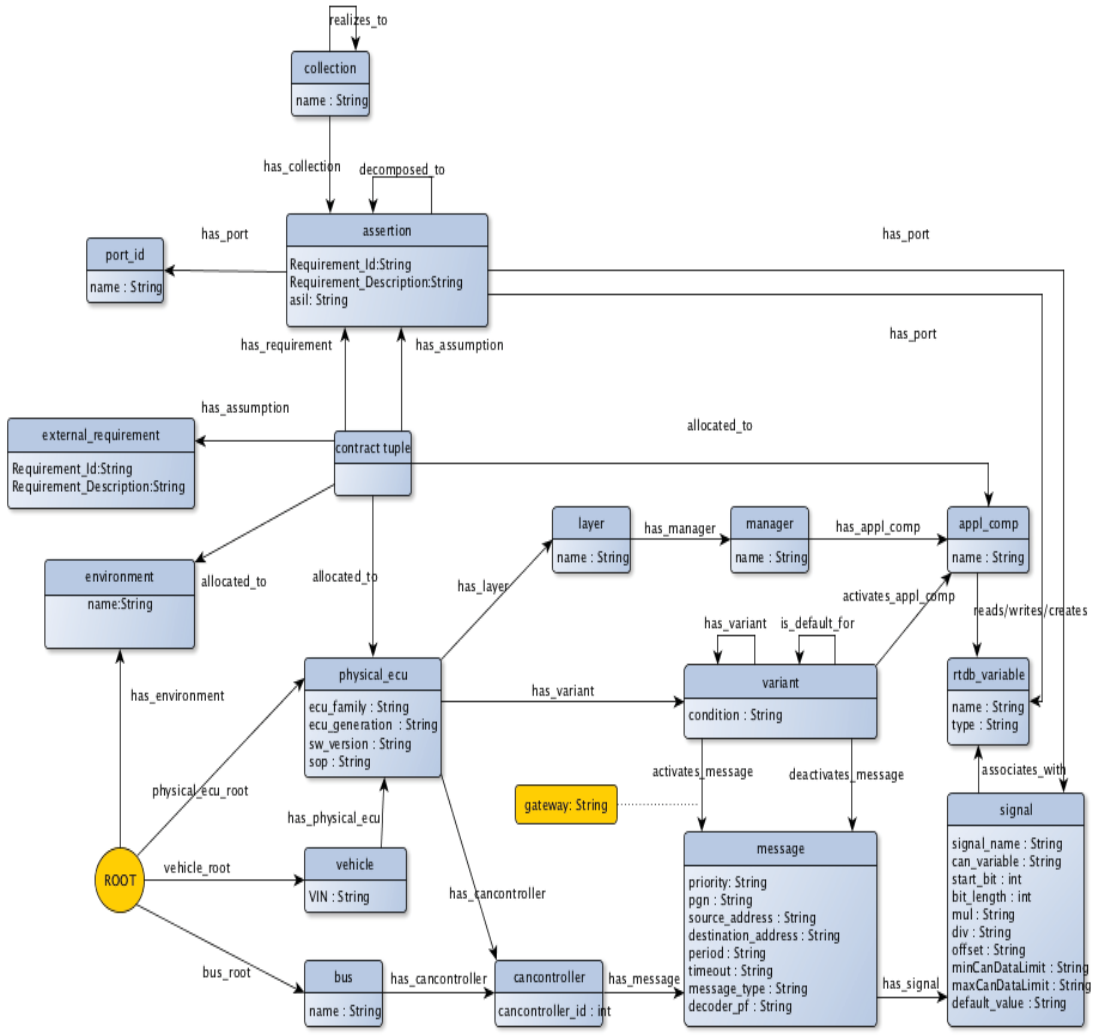


Figure 1: Meta-model of data that need to be extracted from source code

As it shows in the figure 1, data are stored in nodes and they are connected with edges which shows the relationships between nodes.

## 2.3 Comparison to this work

As it is mentioned in previous sections, this thesis project is a research on the relative usefulness of the Document-based databases and analyzing their usability for Scania.

The researches have been done before, were mostly on Graph databases and Relational databases. The study on Document-based databases needs to be done to cover a good review on NoSQL databases.

# DESIGN AND IMPLEMENTATION

## 3.1 NoSQL databases

NoSQL (Not only SQL) database system is a storage alternative to relational databases which, supports fast access to large binary objects using a key based access strategy.

The basic classification of NoSQL databases is based on data model. A few of these and their examples are:

- Key-value databases: Riak, Redis, Project Voldemort
- Document-based databases: MarkLogic, MongoDB, CouchDB
- Column family databases: HBase, Cassandra
- Graph database: Neo4j, Allegro, Virtuoso

A short explanation on each category is provided as following: [1]

**Key-value databases:** Every single item in the database is stored as an attribute name (or 'key'), together with its value.

**Document-based databases:** Pair each key with a complex data structure known as a document. Documents can contain many different key-value pairs, or key-array pairs, or nested documents.

**Column family databases:** Optimized for queries over large datasets, and store columns of data together, instead of rows.

**Graph databases:** Stores data structured in the nodes and relationships of a graph.

## 3.2 Document-based databases

The main focus of this thesis project is on Document-based databases, which are one of the main categories of NoSQL databases.
A document refers to an entity that contains a collection of named fields. In a document database, everything related to a database object is encapsulated together. [2]

Following is an example of a document:

```
{
  name: ”Jim”,
  email: ” Jim@gmail.com”
}
```

Another document in the same database might be:

```
{
   name: ”Bob”,
   email: ”bob@email.com”,
   friends : [
          { name: ”Jennifer”},
          { name: ”Jim”}
 ]
}
```

As we see, documents in a database are schema-free; they have their own schema. Each has unique elements besides some structural elements shared with one another.

Documents can be stored in different formats (JSON, XML or derivatives).

Following is the list of different types of document databases:

- ArangoDB
- BaseX
- Cassandra
- Cloundant
- Clusterpoint
- Couchbase
- CouchDB
- eXist
- FleetDB
- Jackrabbit
- Inquire
- Lotus Notes
- MarkLogic
- MongoDB
- MUMPS
- OrientDB
- RavenDB
- RethinkDB
- Rocket U2
- Sqrrl Enterprise

Characteristic comparison on some of document databases is shonwn in Table 1.

| | MongoDB | CouchDB | MarkLogic | RavenDB |
|---|---|---|---|---|
| **Format** | BSON | JSON | XML | JSON |
| **Query method** | JavaScript | JavaScript | XQuery | LINQ |
| **Implementation language** | C++ | Erlang | C, C++, Java | C#, JavaScript |
| **Best use** | Dynamic queries, frequently written, rarely read statistical data | Occasionally changing data with pre-defined queries | Media, financial, OS-intelligence | OLTP (Online Transaction Processing) applications |
| **Key points** | Retains some properties of SQL such as query and index | Database consistency, easy to use | | .NET based, Native LINQ querying, RESTful, Javascript client |

Table 1: Comparison on different types of document databases

## 3.3 MongoDB

### I. An Overview On MongoDB

In this thesis project, MongoDB is chosen as a document database to be evaluated and implemented on current data in Scania.

MongoDB is a document database in which documents are stored in BSON (Binary JSON) format.

Documents are grouped in a collection, which is equivalent of a table in relational databases.

Collections don't have a schema and documents in a collection can have different fields. They can be referenced or structured as embedded documents, figure 2.

FIGURE 2: EMBEDDED DOCUMENT

In embedded model, related data could be stored in a single document. Denormalizing data makes it possible to retrieve and manipulate related data in a single document, figure 3.

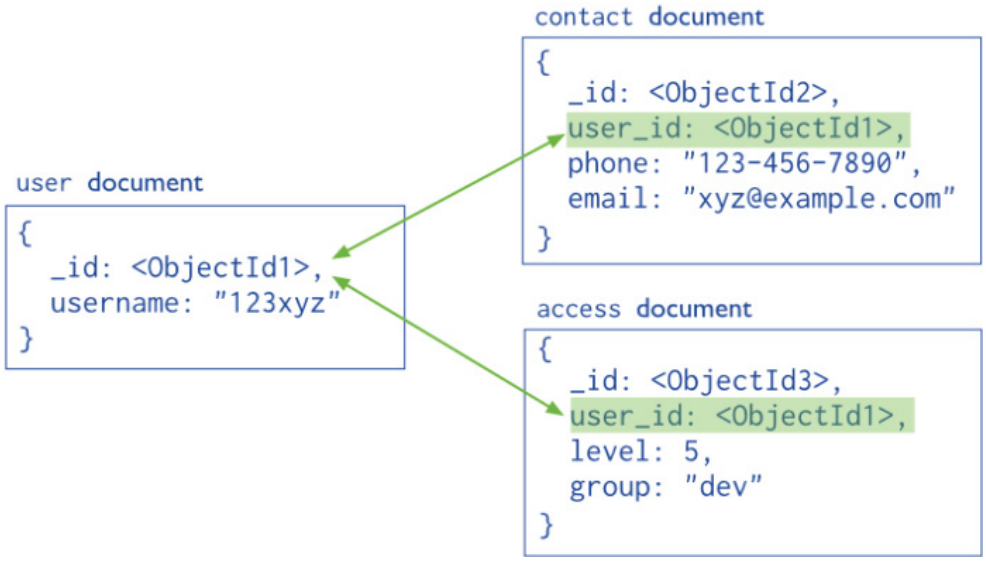


FIGURE 3: REFERENCED DOCUMENT

In referenced model, documents could be linked or referenced to each other by help of references. This way makes it possible to retrieve and manipulate normalized data by references in which stores relationships between data.

Classification of different types of data model in MongoDB is as following

- One-to-one relationships with embedded documents
- One-to-many relationships with embedded documents
- One-to-many relationships with document references
- Tree structures with parent references
- Tree structures with child references
- Tree structures with an array of ancestors
- Tree structures with materialized paths
- Tree structures with nested sets
- Atomic operations
- Support keyword search

Choosing the right data model depends on the application functionality on database and how application is going to interact with data.

In this thesis project, data is designed in the following ways:

One-to-many relationships with document references, Tree structure with child references and Tree structure with array of ancestors. [3]

# II. MongoDB Implementation

Figure 4 is a toy model in Neo4j according to the Meta-model which covers the complexity of relationships between nodes.

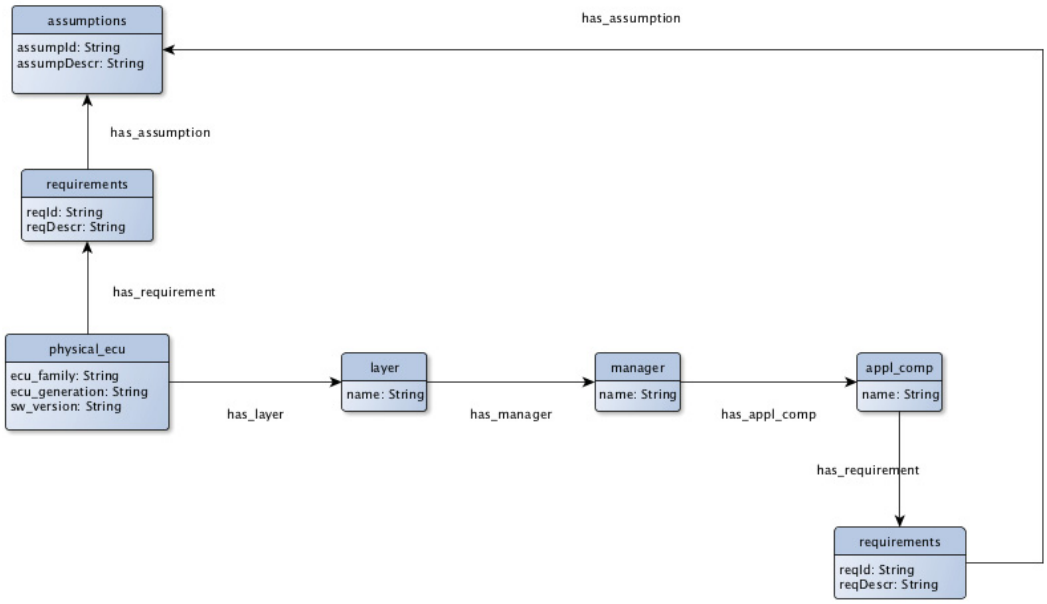


Figure 4: Toy model in Neo4j

The model in the figure 4 shows that the relationship between data is many-to-many relationship.

There are several ways to design data in MongoDB depending on type of queries we run on the data.

Some of the factors considering when designing data model in MongoDB is as follows:

- How the application retrieves and process data.
- How to divide data into documents and collections.
- How far should data be normalized or denormalized in a document.

Figure 5 shows the data model with document references. As it shows in the figure, documents are referenced with their ids. [4]

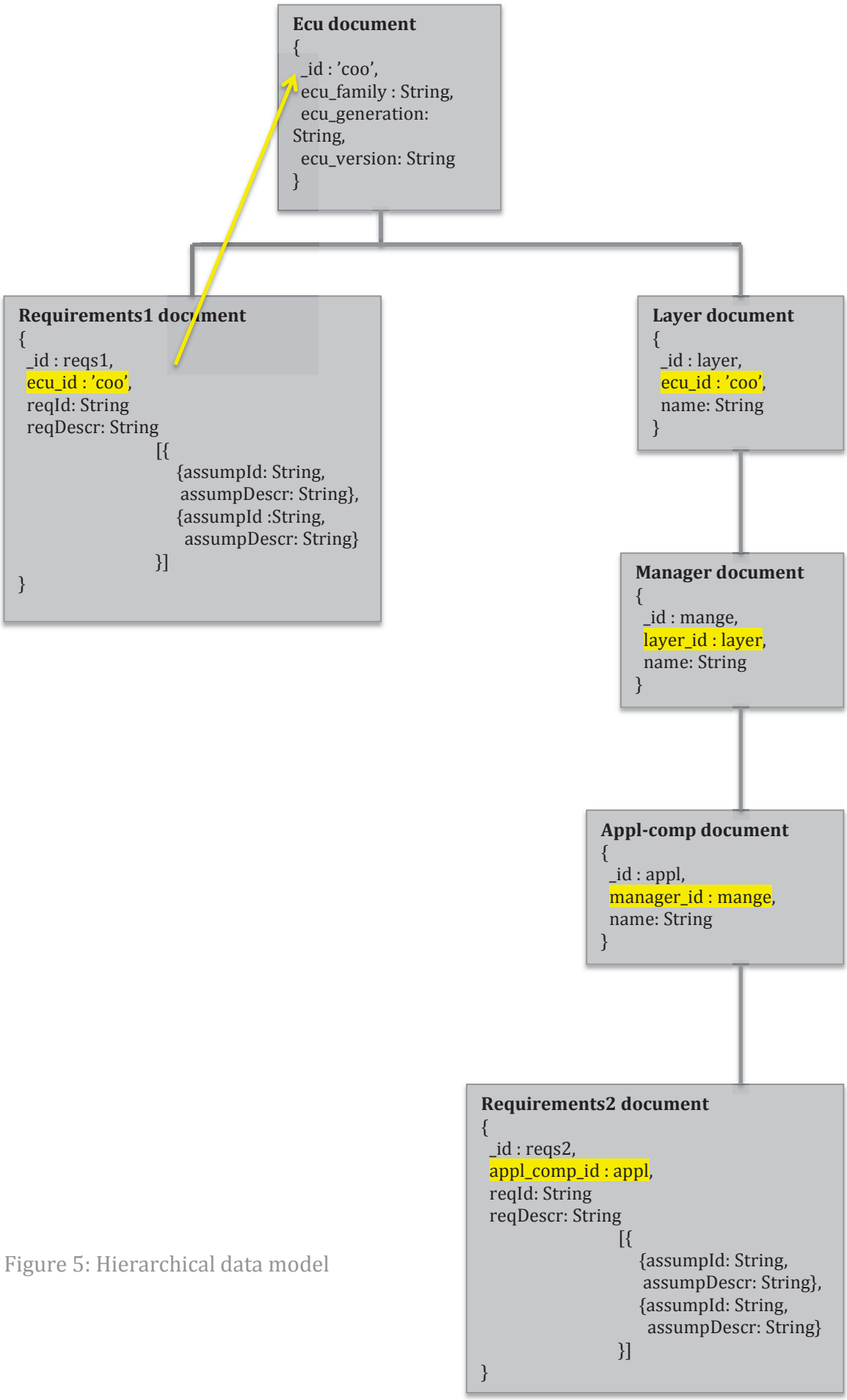


Figure 5: Hierarchical data model

As it shows in figure 5, documents can be referenced by their '_id' object. For example, if we want to retrieve the information of 'ecu' used in 'requirement1' document, we need to run two queries as following:

```
db.coo.insert( { _id: ”reqs1” } )
db.coo.insert( { _id:”coo”})
```

The first query, will retrieve the 'requirement1' document in which shows the 'ecu_id' used is 'coo'. The second document, retrieves the 'ecu' document.

The following example models a tree structure in MongoDB. Each document is stored as a tree node; in addition, the ids of children nodes are stored in an array in documents. As it shows in the following figure, references are stored to the node's children, assuming documents are stored in *coo* collection:

```
db.coo.insert( { _id: ”ecu”, children: [ ”requirements1”, ”layer”] } )
db.coo.insert( {_id: “requirements1”, children: [ ] } )
db.coo.insert( {_id: “layer”, children: [ “manager” ] } )
db.coo.insert( {_id: “manager”, children: [ “appl-comp” ] } )
db.coo.insert( {_id: “appl-comp”, children: [ “requirements2” ] } )
db.coo.insert( {_id: “requirements2”, children: [ ] } )
```

To retrieve the children of a node:

```
db.coo.findOne( { _id: ”ecu” } ).children
```

The result is:

```
[ ”requirements1”, ”layer” ]
```

The tree structure with child references is a suitable tree storage as long as no operation on sub-trees are necessary. A good solution for working with sub-trees is the tree structure with array of ancestors; the descendants and the ancestors of a node could be found, by creating an index on the ancestor's field.

```
db.coo.insert( { _id: ”ecu”, ancestors: [ ], parent: null } )
db.coo.insert( {_id: “requirements1”, ancestors: [“ecu”], parent: “ecu” } )
db.coo.insert( {_id: “layer”, ancestors: [ “ecu”], parent: “ecu” } )
db.coo.insert( {_id: “manager”, ancestors: [ “ecu”, “layer”], parent: “layer” } )
db.coo.insert( {_id: “appl-comp”, ancestors: [ “ecu”, “layer”, “manager” ],
parent:”manager” } )
db.coo.insert( {_id: “requirements2”, ancestors: [“ecu”, “layer”, “manager”,
“appl-comp” ], parent: “appl-comp” } )
```

To retrieve the path of a node:

```
db.coo.findOne( { _id: ”appl-comp” } ).ancestors
```

```
[ ”ecu”, ”layer”, ”manager” ]
```

The result is:

The tree storage is a good solution to persist hierarchical structured data, but not an useful option for storing data with complex relationship. [5]

Following code shows how to interact with data in MongoDB using Java driver.

```
public class MongoJava {

        public static void main(String[] args) throws UnknownHostException {
                MongoClient
mongoClient = new MongoClient( "localhost" , 27017 );
                DB db = mongoClient.getDB( "toymode" );

                Set<String> colls = db.getCollectionNames();

                for (String s : colls) {
                    System.out.println(s);
                }

                DBCollection coll = db.getCollection("ecu");
                DBObject myDoc = coll.findOne();
                System.out.println(myDoc);

                DBCursor cursor = coll.find();
                System.out.println("All documents:");
                try{
                        while(cursor.hasNext())
                        System.out.println(cursor.next());
                        }finally{
                                cursor.close();
                        }
                System.out.println("number of documents in " + coll);
                System.out.println(coll.getCount());



                System.out.println("find the ecu which by reqId");
                BasicDBObject
query = new BasicDBObject("REQUIREMENTS.reqId","ICL001");
                cursor = coll.find(query);
                try{
                        while(cursor.hasNext()){
                                System.out.println(cursor.next());
                        }
                }finally{
                        cursor.close();
                }
        }
```

cording to Meta-model in the project, using MongoDB has some advantages and disadvantages. MongoDB is good for storing hierarchical and unstructured data, access to documents is quick, and it is easy to scale horizontally and easy for simple queries pertaining the details of a single entity.

In other hand, in MongoDB, it is hard to find or list relationships between entities, not capable at handling relationships and complex queries are needed in terms of aggregation framework and MapReduce. Also, MongoDB returns only the whole document depending on whether it hits or not, there is no feature to return only a part of it and if filtering is needed you have to implement it with your own code.

# II. OrientDB

OrientDB is a document database with the features of graph database. It's written in Java and supports SQL as query language. OrientDB is released under the Apache2 license, which means it is free for any use except for enterprise edition.

Relationships in OrientDB are embedded or referenced. The embedded relationship configuration is the same as embedded relationship in MongoDB, but in referenced structure, OrientDB handles relationships using links instead of JOINS as in relational databases. [6]

Besides the multi functional feature of OrientDB, it has some disadvantages that cause to not consider OrientDB as a substitute of Neo4j in Espresso project.

Lack of documentation and clarification of functionality leave confusion working with OrientDB. It is supposed to perform as a document database with features of graph database. For example, a document could be embedded inside a vertex; documents could be linked together and relationships between them could be handled as relationship in graph database.

According to the scope of this thesis project, OrientDB functionality remained vague, and to make sure about it's usability it is needed to implement it on the whole data in real time that is out of the time scope of the thesis project.

# RESULTS AND CONCLUSION

If application needs to track and manage complex relationships, consider using a graph database, if it needs to store and retrieve structured data very quickly, a document database is a good solution.

An integration of a graph database and a document database can be used to apply appropriate means of data storage. Use the graph database for representing relationships and the document database for quick access to documents.

Figure 6 shows an example of storing data in a document database.

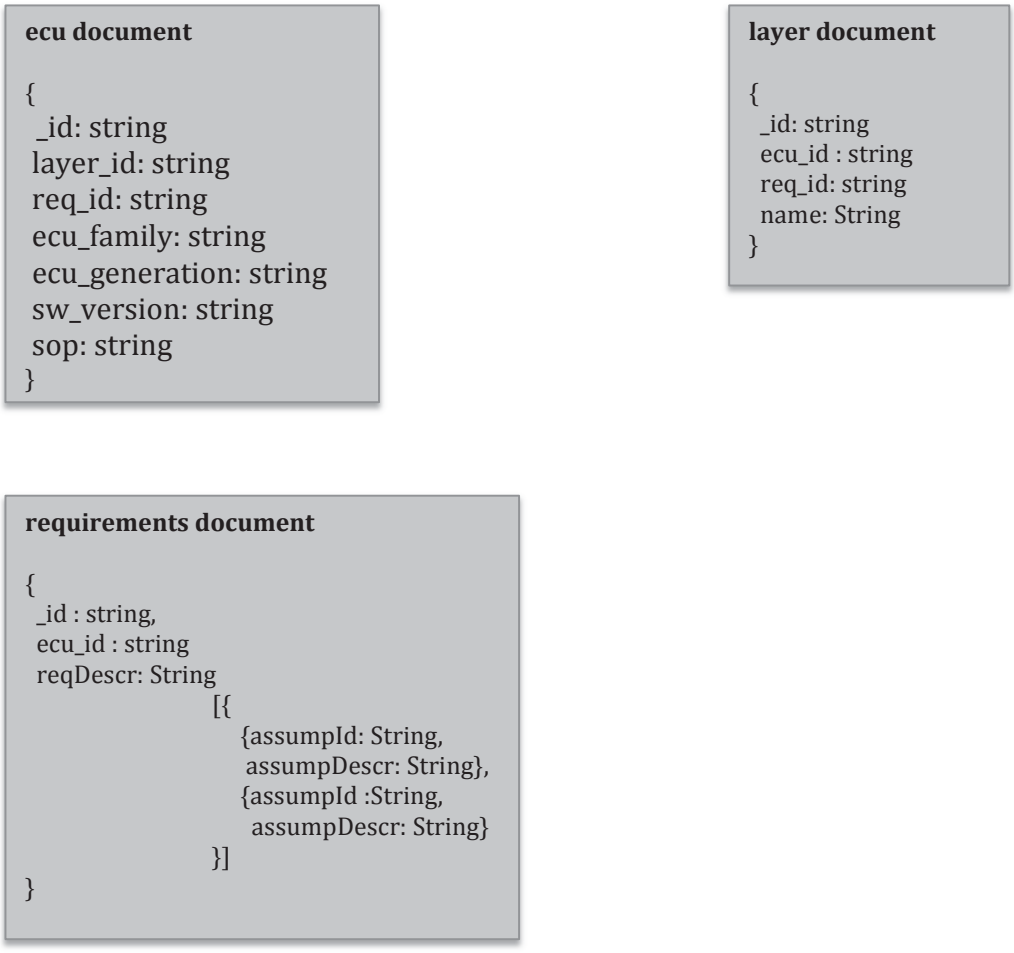


Figure 6: Storing data as a document

These documents are designed for simple queries getting information of each entity. If the application needs to retrieve data about each entity related to what other entities, the application may need to read every document in the entire collection.

One answer to this problem is to store data in a graph database as it is shown in figure 7

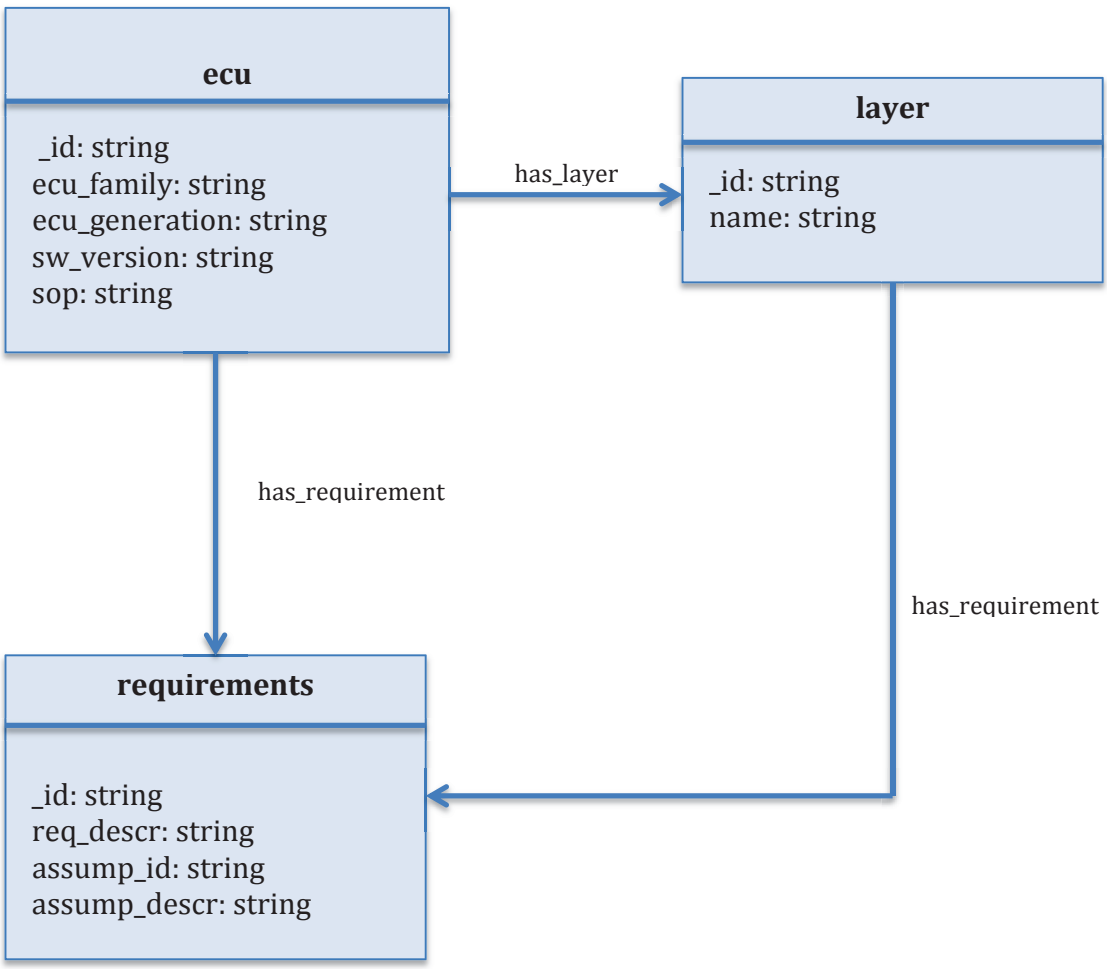


Figure 7: Storing data as a graph

Running queries on these relationships is performed ideally in a graph database. However, if the application needs to store highly structured and complex information in an entity, then a graph database might not provide the capabilities to define these structures.

A solution to this problem is to store information about relationships between documents in a graph database and add references to documents in the document database to the nodes in the graph database. In this way, the data for each entity can be as complicated as the document database will allow, and the graph database only needs to store information about relationships between documents. Figure 8 shows this polyglot solution. [7]

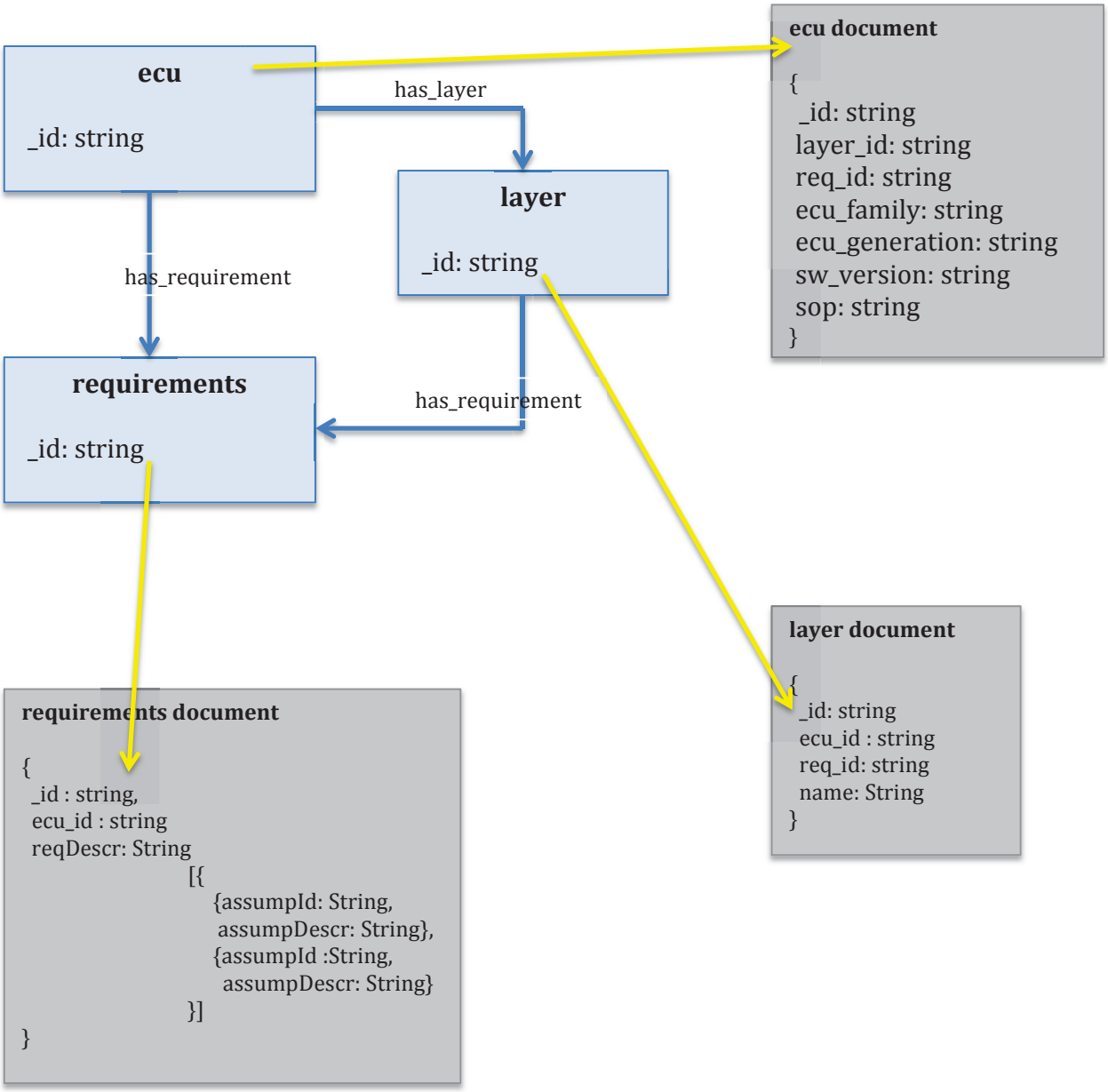


Figure 8: Storing relationship in a graph database and details of each entity in a document database

As it shows in the figure 8, the graph database contains the relationships between entities and only holds minimal details of each entity. The document database contains the full details for each entity. '_id's in the graph database is used to find details of each entity in the document database.

Using two databases have some negative points, addition of a database requires more resources such as disk space, memory, time invested in maintaining two databases and more complexity, but on the other hand, the application takes advantage of each database together.

The question is if MongoDB is really needed in future, or data is not that big and could be stuffed properly into Neo4j?

# REFERENCE LIST


[1]http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.184.483&rep=rep1&type=pdf
[2]http://www.mongodb.com/document-databases
[3]http://docs.mongodb.org/manual/core/data-modeling-introduction/
[4]http://msdn.microsoft.com/en-us/library/dn313284.aspx
[5]http://docs.mongodb.org/manual/tutorial/model-tree-structures-with-child-references/
[6] http://www.orientechnologies.com/orientdb/
[7] http://msdn.microsoft.com/en-us/library/dn313279.aspx
[8]https://www.google.com/patents/US7383272?pg=PA1&dq=ganesh+krishnan&hl=en&sa=X
[9]http://www.christof-strauch.de/nosqldbs.pdf
[10]http://www.codeproject.com/Articles/521713/Storing-Tree-like-Hierarchy-Structures-With-MongoD
[11]http://highscalability.com/blog/2011/6/20/35-use-cases-for-choosing-your-next-nosql-database.html
[12]http://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/
[13]http://users.dcc.uchile.cl/~cgutierr/papers/surveyGDB.pdf
[14]http://www.cs.utexas.edu/users/cannata/dbms/Class%20Notes/08%20Graph_Databases_Survey.pdf
[15]http://thinkaurelius.com/2013/02/04/polyglot-persistence-and-query-with-gremlin/