International Journal of Computer Science and Applications,
©Technomathematics Research Foundation
Vol. 14, No. 1, pp. 65 – 79, 2017

AN ALGORITHM FOR MAPPING THE RELATIONAL DATABASES TO MONGODB – A CASE STUDY

LIANA STANESCU

*Computer Science and Information Technology, University of Craiova, Bvd. Decebal 106,
Craiova, 200440, Romania
stanescu@software.ucv.ro*

MARIUS BREZOVAN

*Computer Science and Information Technology, University of Craiova, Bvd. Decebal 106,
Craiova, 200440, Romania
mbrezovan@software.ucv.ro*

DUMITRU DAN BURDESCU

*Computer Science and Information Technology, University of Craiova, Bvd. Decebal 106,
Craiova, 200440, Romania
dburdescu@yahoo.com*

The paper presents an original algorithm for automatic mapping of relational databases to MongoDB NoSQL databases. The algorithm uses the metadata stored in the relational system tables. It takes into consideration the concepts from Entity-Relationship (ER) model: entity type represented by a relation in the Relational Model (RM), 1:1 and 1:M relationship type represented with Foreign Keys (FK) in the RM and N:M relationship type represented in RM with a join table that contains the Primary Keys (PK) from the original tables, each representing a FK and two 1:M relationships between the original tables and the join table. Such an algorithm is very useful because in practice there are already implemented relational databases with a big number of records. It was proofed that for some of them the solution of implementation using document databases like MongoDB is a better one, from the management point of view, but also regarding the query time response. A significant amount of time and human effort is spared this way, which would have otherwise been needed to create and populate the NoSQL database from scratch Such an example is a hospital database that manages patient sheets that have different structure at different sections. In this stage, the algorithm was tested on a MySQL database containing 10-15 tables with many relationships between them and approximately 100 records/ table.

Keywords: Relational databases; document databases; MongoDb.

1. Introduction

Relational Database Management Systems (RDBMS) have became the first choice for the storage of information in databases mostly used for financial records, manufacturing information, staff and salary data, and so on starting with 1980. RDBMSs are based on the relational model defined by a schema. This model uses two concepts: table and relationship. A relational table represents a well defined collection of rows and columns and the relationship is established between the rows of the tables. Relational data can be queried and manipulated using SQL query language [Krisciunas (2014)].

In practice, there are situations in which storing data in the form of a table is inconvenient, or there are other kinds of relationships between records, or there is the necessity to quickly access the data. In order to solve such problems a new type of NoSQL has been created. A NoSQL or Not Only SQL database provides a mechanism for storage and retrieval of data that is different from the typical relational model.

Another issue that NoSQL solves is the mismatch between relational databases and object-oriented programming. It is known that SQL queries are not well suited for the object oriented data structures that are used in most applications now [Krisciunas (2014)].

Another closely related issue is storing or retrieving an object along with all relevant data. Some operations require multiple and complex queries. In this case, data mapping and query generation complexity rise too much and becomes difficult to be maintained on the application side [Krisciunas (2014)].

Some of these problems have found their answer both in Object-relational mapping (ORM) frameworks, even though it still requires a lot of development effort and also in Object-Oriented Database Management Systems (OODBMS). The down side in this last alternative is the fact that it did not gain much popularity in replacing relational databases. However, most object oriented databases may be considered NoSQL solutions as well [Krisciunas (2014)].

Another problem that relational databases cannot handle is related to an exponentially increasing amount of data. The direct consequence is the so-called big data problem. This problem arises when standard SQL query operations do not have acceptable performances, especially when transactions are involved [Krisciunas (2014)].

As a result, the subject of developing an automatic mapping instrument has been brought up. This instrument will be able to represent the existing relational databases, already populated with a large number of records, as NoSQL databases. A significant amount of time and human effort is spared this way, which would have otherwise been needed to create and populate the NoSQL database from scratch.

This paper describes a framework which implements an algorithm for automatic mapping of relational databases to MongoDB. The algorithm was tested on MySQL databases.

The paper is organized in the following way: Section 2 presents general consideration about NoSQL databases, Section 3 presents the main concepts used by MongoDB, one of the most used NoSQL database, Section 4 presents general principles of mapping relational databases to MongoDB considering the main concepts from ER model and RM. Section 5 presents in detail our proposed algorithm for automatic mapping of a relational databases to MongoDB and also an example of application of this algorithm on a MySQL database. Conclusions and future work are shown in Section 6.

2. NoSQL Databases

During the past few years, a new type of databases, known as NoSQL databases, has appeared, and has become so useful for some type of applications, that has grown to represent a challenge for the relational databases. For a long time, relational databases have been the key element of software industry, providing mechanisms to store data persistently, concurrency control, transactions, mostly standard interfaces and mechanisms to integrate application data, reporting [Celko (2014)].

NoSQL means Not Only SQL. The name suggests that when designing a software solution or product, more than one storage mechanism can be used depending on the needs. Polyglot Persistence represents the most significant consequence from the growth of NoSQL database. Some of the most important characteristics of NoSQL are: it does not use relational model, it runs well on clusters, has mostly open-source and schema-less [Celko (2014)], [Krisciunas (2014)].

NoSQL and RDBMS are designed to support different application requirements and are typically used together in most enterprises. The situations in which the NoSQL database is recommended to be used are: decentralized applications, continuous availability, high velocity data, data coming from many locations, semi/unstructured data, simple transactions, concern to scale both writes and reads, scaling out for more users/data, maintaining high data volumes [Celko (2014)].

This new type of NoSQL databases can be divided in four major categories [Celko (2014)], [Sadalage and Fowler (2013)].

A key-value store, or key-value database, is a data storage paradigm designed for storing, retrieving, and managing associative arrays which can be defined as a data structure that is commonly known as a dictionary or hash. Dictionaries contain a collection of objects, or records, which may contain many different fields within them, each storing data. Each record contains a key that uniquely identifies it and that is also used to find the data within the database in a much more efficient way, leading to a quicker search. Key-value paradigm operates in a very different manner from the well known relational databases (RDB). RDBs pre-define the data structure in the database as a series of tables containing fields with well defined data types. By exposing the data types to the database, we allow it to apply a number of optimizations. In contrast, key-value systems treat the data as a single opaque collection. The records may have different structures. As a consequence we observe that there is considerably more flexibility and certain closeness to modern concepts like object-oriented programming. In RDBs there are fields with NULL value that are allocated anyway. On the other hand key-value stores optional values are not represented by placeholders that often lead to less use of memory to store the same database and to a better performance in certain workloads.

Key-value databases are generally useful for storing session information, user profiles, preferences or shopping cart data. The use of Key-value databases should be avoided when we have to query by data, if relationships between the data exist or if we need to operate on multiple keys at the same time [Sadalage and Fowler (2013)].

The second NoSQL database type is document database, in which the main concept is the document. The database stores and retrieves documents, which can be XML, JSON, BSON, and so on. These documents are self-describing, hierarchical tree data structures which can consist of maps, collections, and scalar values. The documents stored are similar to each other but not necessary the same. Document databases store documents in the value part of the key-value store where the value is examinable. Document databases such as MongoDB feature concepts that allow easier transition from relational databases. They are generally useful to content management systems, blogging platforms, web analytics, real-time analytics or e-commerce applications. The use of these document databases is not recommended for systems that need complex transactions spanning multiple operations or queries against varying aggregate structures [Sadalage and Fowler (2013)].

Column-family databases are the third type of NoSQL database. They store data in column families as rows that have many columns associated with a row key. Column families are groups of related data that are often accessed together. Each column family can be compared to a container of rows in an RDBMS table where the key identifies the row and the row consists of multiple columns. The difference is that various rows do not have to have the same columns, and columns can be added to any row at any time without having to add it to other rows. We call a super column a column that is made of a map of columns. A super column consists of a name and a value. We can associate a super column with a container of columns. This type of NoSQL databases is generally used for content management systems, blogging platforms, maintaining counters, expiring usage, heavy write volume such as log aggregation [Sadalage and Fowler (2013)].

Graph databases are the fourth type of NoSQL databases. They allow the user to store entities and relationships between these entities. Entities are also called nodes that have certain properties. We can associate a node with the instance of an object in the application. Relations are known as edges that can have properties. Edges have directional significance. The nodes are organized according to relationships which give the user the possibility to find interesting patterns between nodes. The organization of the graph allows the data to be stored once and then interpreted in different ways based on relationships. Usually, when we store a graph-like structure in RDBMS, it's for a single type of relationship. Adding another relationship usually means a lot of schema changes and data movement, which does not occur in the case of graph databases. Also, in the relational databases the user must model the graph beforehand based on the desired traversal. Should the traversal changes, the data will also have to change.

In graph databases, traversing the joins or relationships is very fast because they are not calculated at query time as they are persistent. Nodes can have different types of relationships between them. Since there is no limit to the number and kind of relationships a node can have, they all can be represented in the same graph database.

They are recommended for applications that manage connected data, such as social networks, spatial data, routing information for goods and money or recommendation engines [Sadalage and Fowler (2013)].

3. MongoDB

MongoDB is a cross-platform, document oriented database that provides high performance, high availability and easy scalability. The main concepts in MongoDB are collection and document. Database is a physical container for collections. Each database receives its own set of files on the file system. A single MongoDB server typically manages multiple databases [Kvalheim (2015)], [Trivedi (2014)], [Membrey *at al.* (2014)].

The collection is a group of MongoDB documents. It has as correspondent a RDBMS table. A collection exists only within a single database.

A MongoDB document is a set of key-value pairs. The documents do not have to necessarily respect a schema. Typically, all documents in a collection are of similar or have a related purpose. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

Table 1 shows the relationship of RDBMS terminology with MongoDB [Kvalheim (2015)], [Trivedi (2014)], [Membrey *at al.* (2014)].

Table 1. The relationship of RDBMS terminology with Mongodb

RDBMS	**MongoDB**
Database	Database
Table	Collection
Tuple/Row	Document
Column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by MongoDB itself)

Any relational database has a certain design schema that shows the tables and the relationships between them. In MongoDB there is no concept of relationship.

The advantages of MongoDB over RDBMS are [Kvalheim (2015)], [Trivedi (2014)], [Membrey *et al.* (2014)]:

- Schema-less: MongoDB is a document database in which one collection holds different documents whose number of fields, content and size can be different from one document to another.
- Structure of a single object is clear
- No complex joins
- Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that is almost as powerful as SQL
- Tuning
- Ease to scale-out
- Conversion / mapping of application objects to database objects is not needed
- Uses internal memory for storing the working set, enabling faster access to data

Some of the reasons why MongoDB should be used are [Kvalheim (2015)], [Trivedi (2014)], [Membrey *at al.* (2014)]:

- Document oriented storage : data is stored in the form of JSON style documents
- Index on any attribute
- Replication and high availability
- Auto-Sharding
- Rich queries
- Fast updates
- Professional support by MongoDB

MongoDB is recommended to be used in the following applications [Kvalheim (2015)], [Trivedi (2014)], [Membrey *at al.* (2014)]:

- Big Data
- Content Management and Delivery
- Mobile and Social Infrastructure
- User Data Management
- Data Hub (a collection of data from multiple sources)

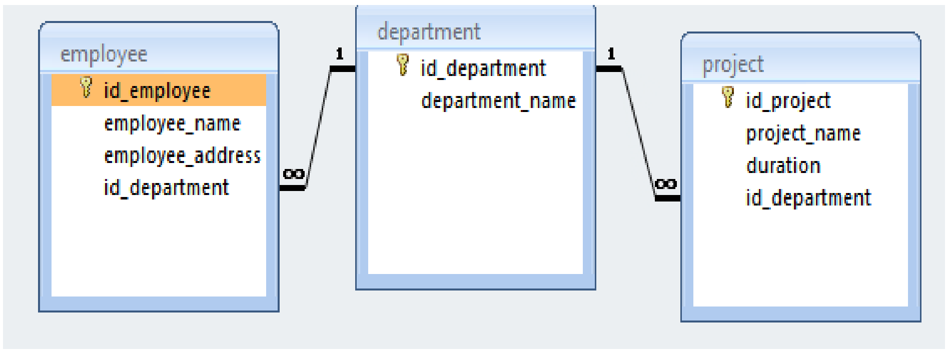
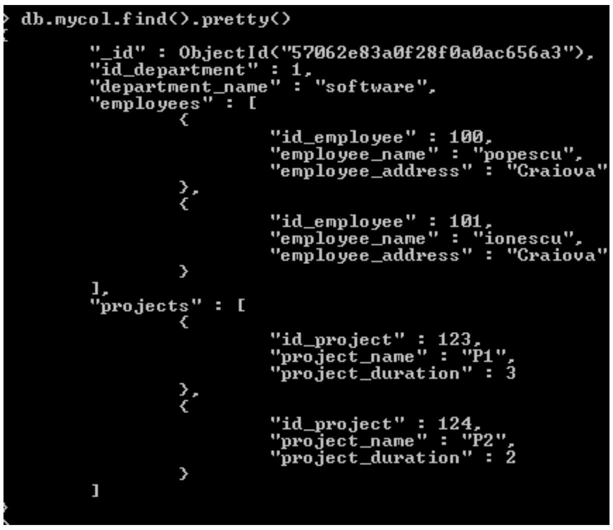


Fig. 1. A relational database.



```
> db.mycol.find().pretty()
{
        "_id" : ObjectId("57062e83a0f28f0a0ac656a3"),
        "id_department" : 1,
        "department_name" : "software",
        "employees" : [
                {
                        "id_employee" : 100,
                        "employee_name" : "popescu",
                        "employee_address" : "Craiova"
                },
                {
                        "id_employee" : 101,
                        "employee_name" : "ionescu",
                        "employee_address" : "Craiova"
                }
        ],
        "projects" : [
                {
                        "id_project" : 123,
                        "project_name" : "P1",
                        "project_duration" : 3
                },
                {
                        "id_project" : 124,
                        "project_name" : "P2",
                        "project_duration" : 2
                }
        ]
}
>
```

Fig. 2. A MongoDB collection with documents.

Figure 1 presents a small relational database with three tables and two 1:M relationships. While in MongoDB schema design will have one collection departments as in figure 2. So while showing the data, in RDBMS you need to join three tables and in MongoDB data will be shown from one collection only.

4. The General Principles of Mapping Relational Databases to MongoDB

In MongoDB the relational database remains a database. A relational table is mapping to a MongoDB collection. The tuples or rows become documents inside MongoDB collections [Kvalheim (2015)], [Trivedi (2014)], [Membrey *at al.* (2014)].

The 1:1 relationship describes a relationship between two entities. For example a Student has a single Address relationship. A Student lives at a single Address and an Address only contains a single Student. The 1:1 relationship can be modeled in two ways using MongoDB. The first is to embed the relationship as a document and the second is as a link to a document in a separate collection [Trivedi (2014)], [Membrey *at al.* (2014)]. Let’s look at both ways of modeling the one to one relationship using the following two documents:

An example of Student document:

```
{Name:”Popescu Alin”,
Faculty:”A.C.E”}
```

An example of Address document:

```
{City:”Craiova”,
Street:”T. Lalescu”,
Nr:10}
```

The first approach is simply to embed the Address document in the Student document [Trivedi (2014)], [Membrey *at al.* (2014)].

An example of Student document with embedded Address:

```
{Name:”Popescu Alin”,
Faculty:”A.C.E”,
Address:{
    City:”Craiova”,
Street:”T. Lalescu”,
Nr:10}}
```

The strength of embedding the *Address* document directly in the Student document is that we can retrieve the student and its address in a single read operation versus having to first read the user document and then the address document for that specific student. Since addresses have a strong affinity to the student document the embedding makes sense here [Trivedi (2014)], [Membrey *at al.* (2014)].

The second approach is to link the address and student document using a foreign key [Trivedi (2014)], [Membrey *at al.* (2014)].

An example of Student document:

```
{id:1,
Name:”Popescu Alin”,
```

Faculty:”A.C.E”}

An example of Address document with Foreign Key:

{Student_id:1,
City:”Craiova”,
Street:”T. Lalescu”,
Nr:10}

This is similar to how traditional relational databases store the data. It is important to note that MongoDB does not enforce any foreign key constraints so the relation only exists as part of the application level schema.

In the one to one relationship embedding is the preferred way to model the relationship as it’s more efficient to retrieve the document.

The 1:M relationship describes a relationship where one side can have more than one relationship while the reverse relationship can only be single sided. An example is a Department where a department might have many Employees but an Employee is related only to a single Department.

The 1:M relationship can be modeled in several different ways using MongoDB. The first model is embedding, the second is linking and the third is a bucketing strategy that is useful for cases like time series [Trivedi (2014)], [Membrey *at al.* (2014)].

An example of Department document:

{id:1,
Dname:”Software”,
Dlocation:”Craiova”}

An example of Employee documents:

{eid:100,
Ename:”Popescu Ion”}
{eid:121,
Ename:”Cornescu Alin”}

The first approach is to embed the Employees in the Department:

{id:1,
Dname:”Software”,
Dlocation:”Craiova”,
Employees:[
{eid:100,
Ename:”Popescu Ion”},
{eid:121,
Ename:”Cornescu Alin”}]}

The embedding of the employees in the department means that we can easily retrieve all the employees belong to a particular department. Adding new employees is very simple. However, there are three potential problems associated with this approach [Trivedi (2014)], [Membrey *at al.* (2014)]:

- The first is that the employees array might grow larger than the maximum document size of 16 MB.

- The second aspect is related to write performance. As employees get added to Department over time, it becomes hard for MongoDB to predict the correct document padding to apply when a new document is created. MongoDB would need to allocate new space for the growing document. In addition, it would have to copy the document to the new memory location and update all indexes. This could cause a lot more IO load and could impact overall write performance. It's important to note that this only matters for high write traffic and might not be a problem for smaller applications. What might not be acceptable for a high write volume application might be tolerable for an application with low write load.
- The third problem is exposed when one tries to perform pagination of the employees.

The second approach is to link employees to the department using a more traditional foreign key. An example of Employees documents with Foreign Keys:

```
{id:1,
eid:100,
Ename:"Popescu Ion",
Bdate:"1967-01-03"}
{id:1,
eid:121,
Ename:"Cornescu Alin",
Bdate:"1968-02-05"}
```

An advantage of this model is that additional employees will not grow the original department document, making it less likely that the applications will run in the maximum document size of 16 MB. It's also much easier to return paginated employees as the application can slice the employees more easily. On the downside if we have 1000 employees working in a department, we would need to retrieve all 1000 documents causing a lot of reads from the database [Trivedi (2014)], [Membrey *at al.* (2014)].

The third approach, called bucketing is a hybrid of the two above [Trivedi (2014)], [Membrey *at al.* (2014)]. Basically, it tries to balance the rigidity of the embedding strategy with the flexibility of the linking strategy. For this example, we will split the employees into buckets with a maximum of 50 employees in each bucket.

An example of Employee bucket:

```
{id:1,
Page:1,
Count:50,
Employees:
[{eid:100,
Ename:"Popescu Ion",
Bdate:"1967-01-03"   },
{eid:121,
Ename:"Cornescu Alin",
Bdate:"1968-02-05"   },…]}
```

The main benefit of using buckets in this case is that we can perform a single read to fetch 50 employees at a time, allowing for efficient pagination. When the user has the possibility of splitting up the documents into discreet batches, it makes sense to consider bucketing to speed up document retrieval [Trivedi (2014)], [Membrey *at al.* (2014)].

A N:M relationship in the ER model is an example of a relationship between two entity types where they both might have many relationships between entities. An example might be a Book that was written by many Authors. At the same time an Author might have written many Books.

N:M relationships are modeled in the relational database by using a join table that contains the primary keys from the original ones, each representing a foreign key, and two 1:M relationships.

In MongoDB we can represent this situation in many ways. The first way is called Two Way Embedding [Trivedi (2014)], [Membrey *at al.* (2014)].

In Two Way Embedding we will include the Book foreign keys under the book field in the author document. Mirroring the Author document, for each Book we include the Author foreign keys under the Author field in the book document.

An example of Author documents:

```
{id:1,
Name:”Popescu Ion”,
Books:[1,2]}
{id:2,
Name:”Popa Alina”,
Books:[2]}
```

An example of Book documents:

```
{id:1,
title”Multimedia Databases”,
authors:[1]}
{id:2,
title”Multimedia”,
authors:[1,2]}
```

Another way of modeling N:M relationships is called One Way Embedding [Trivedi(2014)], [Membrey *at al.* (2014)]. The One Way Embedding strategy chooses to optimize the read performance of a N:M relationship by embedding the references in one side of the relationship. An example might be a N:M relationship between books and categories. The case is that several books belong to a few categories but a couple categories can have many books. Let’s look at an example with the categories represented into a separate document. An example of Category documents:

```
{id=1,
Cname=”Multimedia”}
{id=2,
Cname=”Databases”}
```

An example of a Book document with foreign keys for Categories:

{id:1,
title"Multimedia Databases",
categories:[1, 2],
authors:[1]}
{id:2,
title"Multimedia",
categories:[1],
authors:[1,2]}

The reason for choosing to embed all the references to categories in the books is due to the fact that being lot more books in a category than categories in a book. If the user embeds the books in the category document it's easy to foresee that is possible to break the 16MB max document size for certain broad categories [Trivedi (2014)], [Membrey *at al.* (2014)]. To choose two way embedding or one way embedding, the user must establish the maximum size of N and the size of M. For example if N is a maximum of 3 categories for a book and M is a maximum of 500000 books in a category you should pick One Way Embedding. If N is a maximum of 3 and M is a maximum of 5 then Two Way Embedding might work well [Trivedi (2014)], [Membrey *at al.* (2014)].

5. Algorithm Description

It was created a framework that implements an algorithm of automatic mapping of relational databases to MongoDB. The algorithm is explained and tested on MySQL databases. The algorithm uses the relational (MySQL) INFORMATION_SCHEMA that provides access to database metadata. Metadata is data about the data, such as the name of a database or table, the data type of a column, or access privileges. INFORMATION_SCHEMA is the information database, the place that stores information about all the other databases that the relational server maintains. Inside INFORMATION_SCHEMA there are several read-only tables. They are actually views, not base tables [DuBois (2013)], [DuBois (2014)]. For examples we use a database db1 that contains 5 tables: Employee, Department, Project, Child and Works_on. The relationships are 1:M between Department and Employee, Employee and Child, Department and Project (figure 3). In the ER model there is a N:M relationship, implemented in the relational model by the join table Works_on and two 1:M relationship between Employee and Works_on and Project and Works_on.

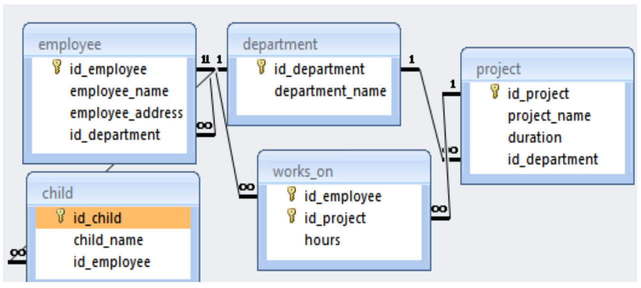


Fig. 3. A relational database.

The steps of the algorithm implemented in our framework are presented next.

(1) Creating the MongoDB database

The user must specify the relational (MySQL) database that will be represented in MongoDB. The database is created with the following MongoDB command: use DATABASE_NAME [Membrey *at al.* (2014)].

```
>use db1
switched to db db1
```

(2) Creating tables in the new MongoDB database. The algorithm verifies for each table in what relationships is involved, if it has foreign keys and/or is referred by other tables.

   (i) If the table is not referred by other tables, it will be represented by a new MongoDB collection

   (ii) If the table has not foreign keys, but is referred by another table, it will be represented by a new MongoDB collection.

   (iii) If the table has one foreign key and is referred by another table, it will be represented by a new MongoDB collection. In our framework, for this type of tables we use linking method, using the same concept of foreign key.

   (iv) If the table has one foreign key but is not referred by another table, the proposed algorithm uses one way embedding model. So, the table is embedded in the collection that represents the table from the part 1 of the relationship.

   (v) If the table has two foreign keys and is not referred by another table, it will be represented using the two way embedding model, described in section IV.

   (vi) If the table has 3 or more foreign keys, so it is the result of a N:M ternary, quaternary relationships, the algorithm uses the linking model, with foreign keys that refer all the tables initially implied in that relationship and already represented as MongoDB collections. The solution is good even the table is referred or not by other tables.

In order to find the name of the tables stored in the relational (MySQL) database the next Select command is used:

```
Select table_name
From information_schema.tables
Where table_schema='db1'
Order By table_name;
```

The INFORMATION_SCHEMA.TABLES provides information about tables in databases [DuBois (2013)], [DuBois (2014)].

The table TABLES_CONSTRAINTS from INFORMATION_SCHEMA database describes which tables have constraints [DuBois (2013)], [DuBois (2014)]. It must be executed a Select command on each table in database, as in the next example:

```
Select constraint_type
From information_schema.tables_constraints
Where table_schema='db1'
And table_name='department';
```

The CONSTRAINT_TYPE value can be unique, primary key or foreign key. We are interested by primary key and foreign key constraints [DuBois (2013)], [DuBois (2014)].

The table REFERENTIAL_CONSTRAINTS from the same relational (MySQL) system database provides information about foreign keys. The attributes constraint_schema and constraint_name identify the foreign key. The attributes: unique_constraint_schema, unique_constraint_name and referenced_table_name identify the referenced key [DuBois (2013)], [DuBois (2014)].

With the data from these system tables: tables, tables_constraints and referential_constraints the framework can establish what step of the algorithm (2.i-2.vi) must be applied.

Relational tables become collections in MongoDB. The collections are created using createCollection() method.

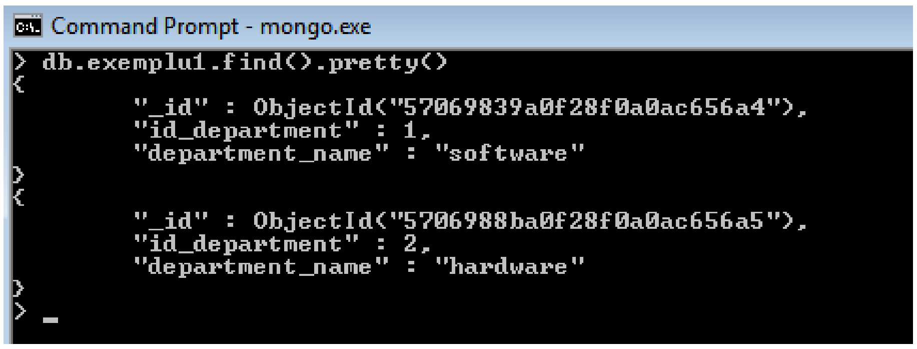
Basic syntax of **createCollection()** command is as follows [Membrey *at al.* (2014)]:

Db.createCollection(name, options)

Where name represents the collection name and options specify options about memory size and indexing.

For the relational database from figure the mapping according to the presented algorithm is presented next.

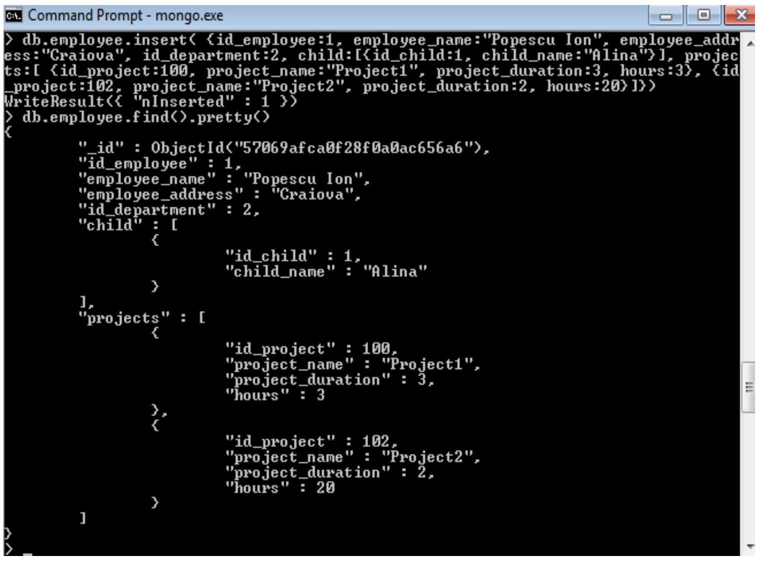
The table Department has no foreign keys but is referred by other two tables, so it becomes a MongoDB collection (step 2.ii). The table Employee has one foreign key and is referred by the table Works_on, so it becomes a collection (step 2.iii). The table Project has one foreign key and is referred by Works_on, so it becomes also a collection (step 2.iii). The table Works_on has two foreign keys and is not referred by other tables, so it will be implemented using two way embedding model (step 2.v). The projects will be assigned to each employee and also, to each project will be assigned the employees that work on that project. The table Child has foreign key but is not referred by another table, so it will be represented using one way embedding model (step 2.iv). So it will be embedded in Employee collection. The five relational tables will be represented by three MongoDB collections. Next, there are some samples of the MongoDB collections generated by the presented algorithm.



```
> db.exemplu1.find().pretty()
{
        "_id" : ObjectId("57069839a0f28f0a0ac656a4"),
        "id_department" : 1,
        "department_name" : "software"
}
{
        "_id" : ObjectId("5706988ba0f28f0a0ac656a5"),
        "id_department" : 2,
        "department_name" : "hardware"
}
> _
```

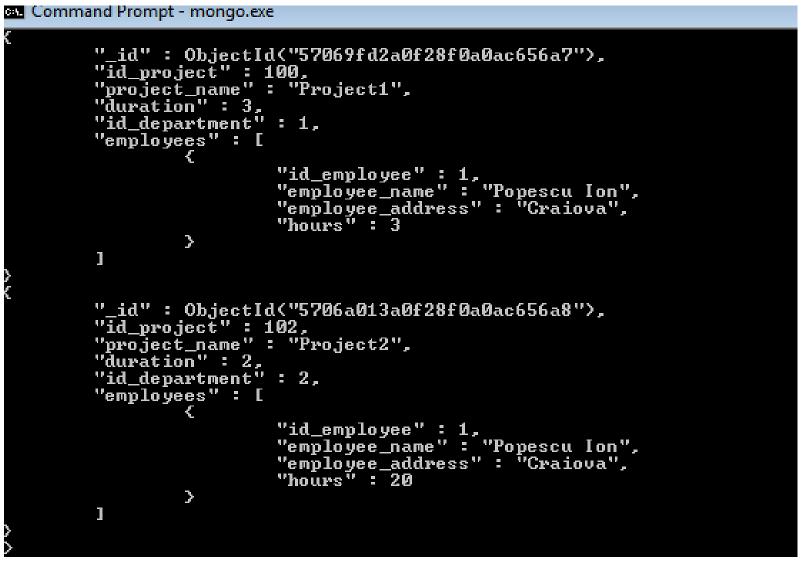
Fig. 4. MongoDB collection that represents the Department table.

Department collection is presented in figure 4. Employee Collection that contains the document Child and Projects is presented in figure 5. Project collection that embeds the documents employees that work on these projects is shown in figure 6.



```
> db.employee.insert( {id_employee:1, employee_name:"Popescu Ion", employee_addr
ess:"Craiova", id_department:2, child:[{id_child:1, child_name:"Alina"}], projec
ts:[ {id_project:100, project_name:"Project1", project_duration:3, hours:3}, {id
_project:102, project_name:"Project2", project_duration:2, hours:20}]})
WriteResult({ "nInserted" : 1 })
> db.employee.find().pretty()
{
        "_id" : ObjectId("57069afca0f28f0a0ac656a6"),
        "id_employee" : 1,
        "employee_name" : "Popescu Ion",
        "employee_address" : "Craiova",
        "id_department" : 2,
        "child" : [
                {
                        "id_child" : 1,
                        "child_name" : "Alina"
                }
        ],
        "projects" : [
                {
                        "id_project" : 100,
                        "project_name" : "Project1",
                        "project_duration" : 3,
                        "hours" : 3
                },
                {
                        "id_project" : 102,
                        "project_name" : "Project2",
                        "project_duration" : 2,
                        "hours" : 20
                }
        ]
}
> _
```

Fig. 5. MongoDB Employee collection.



```
{
        "_id" : ObjectId("57069fd2a0f28f0a0ac656a7"),
        "id_project" : 100,
        "project_name" : "Project1",
        "duration" : 3,
        "id_department" : 1,
        "employees" : [
                {
                        "id_employee" : 1,
                        "employee_name" : "Popescu Ion",
                        "employee_address" : "Craiova",
                        "hours" : 3
                }
        ]
}
{
        "_id" : ObjectId("5706a013a0f28f0a0ac656a8"),
        "id_project" : 102,
        "project_name" : "Project2",
        "duration" : 2,
        "id_department" : 2,
        "employees" : [
                {
                        "id_employee" : 1,
                        "employee_name" : "Popescu Ion",
                        "employee_address" : "Craiova",
                        "hours" : 20
                }
        ]
}
>
```

Fig. 6. MongoDB Project collection.

6. Conclusion and Future Work

The paper presents a framework that implements our original algorithm of automatic mapping a MySQL relational database to a MongoDB NoSQL database. The algorithm uses the metadata stored in the MySQL system tables. It takes into consideration the concepts from Entity-Relationship (ER) model: entity type represented by a relation in the Relational Model (RM), 1:1 and 1:M relationship type represented with Foreign Keys (FK) in the RM and N:M relationship type represented in RM with a join table that contains the Primary Keys (PK) from the original tables, each representing a FK and two 1:M relationships between the original tables and the join table.

The algorithm was presented in detail, on steps. Also, the paper contains an example of automatic mapping of a MySQL database to MongoDB using our algorithm.

The paper also presents the initial results of our algorithm that was tested on small size databases (10-15 tables with many relationships and 100 records/table), the results being encouraging.

The future work will include the next steps:

- Experiments on complex databases (many tables and a large number of records/table)
- Taking into consideration the number of records in the tables and the operations on the database (insert, update, delete query) in order to implement the more appropriate model of mapping to MongoDB
- Modeling tree structures with parent references
- Extending the framework to execute mapping to MongoDB of other relational databases (Oracle, MS SQL Server and so on)

References

Celko, J. (2014). *Joe Celko's Complete Guide to NoSQL: What Every SQL Professional Needs to Know about Non-Relational Databases,* 1st Edition. Elsevier, USA

DuBois, P. (2013). *MySQL (5th Edition) (Developer's Library) 5th Edition*. Addison Wesley, New Jersey

DuBois, P. (2014). *MySQL Cookbook: Solutions for Database Developers and Administrators*. 3rd Edition. O'Reilly Media, USA

Krisciunas**,** A. (2014). Benefits of NoSQL**:** https://www.devbridge.com/articles/benefits-of-nosql/

Kvalheim, C. (2015). *The Little Mongo DB Schema Design Book*. The Blue Print Series

Membrey, P., Hows. D., Plugge, E. (2014). *MongoDB Basics*. Apress

Sadalage, P.J.; Fowler, M. (2013). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, 1st Edition, Addison Wesley, USA

Trivedi, A. (2014). Mapping Relational Databases and SQL to MongoDB: http://code.tutsplus.com/articles/mapping-relational-databases-and-sql-to-mongodb--net-35650