Published by
NIGERIAN SOCIETY OF PHYSICAL SCIENCES
Available online @ https://journal.nsps.org.ng/index.php/jnsps

J. Nig. Soc. Phys. Sci. 5 (2023) 1089

Journal of the Nigerian Society of Physical Sciences

An Integral Approach for Complete Migration from a Relational Database to MongoDB

Abdelhak Erraji[a,*], Abderrahim Maizate[b], Mohamed Ouzzif[b]

[a]*RITM ESTC Laboratory Hassan II University, ENSEM Casablanca, Morocco*
[b]*RITM ESTC Laboratory Hassan II University, ESTC Casablanca, Morocco*

Abstract

Recently our world has recognized a very important revolution in all sectors of life; this evolution comes following the good exploitation of the data generated by intensive human communication, which uses social networks and computer tools. NoSQL systems appear to resolve some limitations of relational databases in the management of BigData. This situation has created a crucial need to migrate relational databases to NoSQL systems, especially for companies that want to keep their old data accumulated for years, which describes their experiences, market studies, and the behavior of customers, competitor positions, and future strategic alignment. Our contribution comes to developing our complete approach to migrating the relational database to MongoDB. This article will start with an introduction, after showing related work with a discussion then we move on to present our analysis and modeling methodology, during it, we develop our models and meta-models of the two systems: source and destination of migration. After we present our global approach called "TMSDRDND", which divides its treatment into two layers: "TSRSNLayer" and "MDRSNLayer". The first deals with the transformation of the structure and the semantic data, and the second takes care of data migration using an ETL to be developed according to a specific concept and architecture and exploiting the results of the first layer. During these two layers, we treat three axes, each of which processes a part of the RDB according to their nature: data stored in tables, data carried on the structure, and data coming from the semantics of relational databases.

DOI:10.46481/jnsps.2023.1089

Keywords: Data Migration, Database Transformation, NoSQL, Big Data, Data, ETL, approach

Article History :
Received: 24 September 2022
Received in revised form: 16 March 2023
Accepted for publication: 25 March 2023
Published: 24 April 2023

Communicated by: T. Latunde

1. Introduction

The use of information, data processing, and communication tools, as well as the web 2.0 revolution, has caused the creation of an enormous amount of data that evolves continuously every time until we have arrived at a big data level. The latest study announces that globally, data is doubling its volume every two years [1] and that an amount of data will be created in 2025 in the order of 163 000 milliards gigabytes [2]. This huge amount of data comes from various data sources, in different natures, and is exploited by several tools, in different sectors, for different purposes, and in many ways [3-4]. This information was used in data warehouses and DataMart to calculate performance and objective indicators, to feed decision-making and strategic systems, that were considered at the time to be a revolution in storage and operation data, knowing that

*Corresponding author tel. no: +212 601486615
Email address: erraji.abdelhak@gmail.com (Abdelhak Erraji)

they store and manage a considerable amount of data and perform analyzes based on data accumulated over several years to be able to make the right decisions, even if it uses the relational model and a relational databases management system. This information has played a very important role in reading the present situation of institutions, publicizing their products and services, and also planning and building a good future, which guarantees the existence and resistance in the market and also its evolution on a scale, national or international [5-6]. While data is big then it will be more important and constructive in the derivation of other information. Also, their role is important in the precise analysis of the present, the efficient construction of the future, the overtaking of the competitors, the distinction and the control of the field of activity, and also in the good control and management of processes, as well as the management of the institution for the future. To do this, we need to adopt a system and model for managing this data. The system and the relational model have confirmed its success for small and medium-sized databases, with its strict mathematical model making both its strength and its weakness; in fact, it has dispersed the data in several coherent, compact tables linked to each other by foreign keys. Each table contains a set of fields all dependent on the primary key to forming entity integrity. This structure has succeeded in considerably reducing the redundancy of information, and succeeded in extracting new information from other data stored at the base, without forgetting the power to realize complex and interesting calculation formulas using the power of the system's relational content in joins. However, when the databases became very large, this model presents several weaknesses in the storage of this data, its management, the response time to requests, the availability of data, and the evolution toward very large sizes. This has raised the limits of the relational model, and to overcome it, database designers have designed new systems that do not implement the relational model, called NoSQL (Not Only SQL) systems. These systems include several tools that implement four models by forming four categories of these systems: key-value store, column-value store, document store, and graph store. Each category is adapted to a specific need the rules for data management, storage, and use as well as the expected need of the data management system are not the same from one institution to another. For example, the needs of international banks such as PayPal, and eBay are not the same as Facebook and Twitter, which in turn are different from WhatsApp or Google. These NoSQL systems implement simple and flexible schemas and structures in data management, accepting data from many sources and in different forms, with simple mechanisms to manage the flat and basic files containing the data. These systems implement the horizontal scalability of database servers, which makes it possible to add and link several servers together to form a single data container, to build a distributed database, and with replications in such a way as to guarantee the real-time response and availability of data at all times. On the contrary, relational databases use a single data server for their hosting, by implementing vertical scalability. Also, if we want to create a distributed database in relational systems that requires techniques such as CORBA to deal with the heterogeneity of the systems used, the difference between the ways of managing and exploiting the internal files containing the data, and the means of communication between them. This will significantly reduce their efficiency and performance. Relational database management systems have known several limitations in the management and exploitation of BigData generated by the accumulation of data for years of exercises. This data represents a real treasure for the organizations, because it describes their experiences in the past, and their opportunities of the present by studying the market and the competitors, also it makes it possible to better plan the future. These organizations are adopting NoSQL systems, which come to overcome these limitations of relational systems. At this point, the use of both systems is not the optimal solution, because it considerably increases the costs and expenses in terms of software, technical support, formations, and user training in the specificities of the two systems. In this situation, these institutions should adopt the NoSQL system and migrate the old databases to this system. In this optic, many studies and approaches have been developed, but they present a lack either in ways of transformation or by the neglect of important data during their migration. At this level, it is legitimate to ask the following questions: What is the analysis methodology, which follows to encompass all the elements of the relational database, and to guarantee good migration results? May we migrate all the data and functionalities of the relational system to another NoSQL, considering their differences? How do you transfer the operational elements of relational systems such as triggers, views, and joins? What is the NoSQL category that promotes a full migration to the NoSQL system?

2. Related Work and Discussion

2.1. Related Work

Currently, in the scientific forum, research and discussion are taking place on the best way to migrate and transform data from the old relational system to big data to keep pace with the current development in the field of data storage and exploitation, knowing that NoSQL systems present a better way to store Big Data. In this regard, we present the results of the research through the most important research and presented as follows :

Mearaj and al. [7] transformed the data from its native form into documents. Result files and their collections take their definitions during creation. The profitability of this solution has led to the use of the cloud in data storage. They use MongoDB as their target system because it can generate data directly and with its latest version supports multi-document ACID transactions to maintain data integrity. Gopalan M and al. [8] propose to address the translation engine from relational databases to NoSQL Cassandra databases. They concluded that Oracle and MySQL are better than Cassandra and these new NoSQL products are purely commercial. However, this conclusion is still very subjective and not based on objective indicators. Yansyah and Arry [9] propose a framework that can migrate data in a real operating environment and a framework that can serve as a reference for developers to migrate data between databases NoSQL. NoSQL systems are only studied in a limited way that does not

go beyond core NoSQL, and the data security aspects of the migration process are not considered part of the job. Kachaoui and Belangour [10] propose an approach for converting relational databases to NoSQL databases using MySQL and MongoDB systems. The study describes the transformation method in detail and evaluates it using a small database as the scope of the proposed algorithm. These authors consider his work to be the result of intensive research in related research, filling gaps overlooked by concerned researchers in the field and making small contributions to data transformation. Ganesh and Rajeswari [11] propose a method for migrating data from MySQL to MongoDB and a Naive based on MapReduce model for efficient classification of Big Data. The author mentioned that after migrating data to MongoDB for CRUD operation, query execution performance was improved. The proposed model is implemented using the census income dataset. Cerjeka and al. [12] propose an initial set of transformation rules that will be used in the integration process to develop a new catalog of data warehouse systems based on data warehouses that track changes applied to data sources relational and NoSQL. The transformation rule set includes a total of five rules and was developed for the MongoDB system. Namdeo and Suman [13] propose the use of schema design for NoSQL databases to develop an SDAM schema design model to transition from RDBMS to NoSQL databases. It has three phases: creating the NoSQL database schema (especially document databases), creating all possible database schema combinations, and costing. They assume they can automatically create all possible database schemas. They treat select queries and update queries as inputs used in relational models. Ghule and Vadali [14] propose a flexible, modular data adapter for hybrid database systems. The data adapter uses a generic SQL layer that accepts requests from application services. The data adapter also controls the flow of requests during database conversion. They took an approach where calculations were performed on existing large-scale data in an object storage system without moving the data anywhere. For the author, these NoSQL databases offer only weak consistency, which makes them unsuitable for applications requiring strong consistency. Yassine [15] uses MongoDB as a NoSQL database and MySQL as a relational database. His experiments are based on three NoSQL structures, including embedding-only documents, citation-only documents, and both. They use a large number of records and a set of five queries as a benchmark to measure recovery time. Shiromoto and al. [16] specialize in NoSQL databases, they use a combination of HBase and relational databases. They evaluated the performance of table join execution time and system memory consumption on real machines. These performances turn out to be tolerable in practical use compared to the embedded use of only relational databases. They also showed that there are benefits to changing the join order or saving column families as unique Hbase items.

2.2. Discussion

In the previous approaches, we noticed that the authors of the papers quoted above tried to treat databases integrally, but they neglected very interesting parts of relational databases during their transformation according to their approaches. The majority of these authors did copy/paste from a relational system to a non-relational one. Indeed, the relational system has dispatched their data in several tables, which are linked together by foreign keys, representing implicit relations between these tables. These foreign keys show that these tables will be joined to reconstruct the entire data dispatched in several tables. However, the transformation of these tables by keeping their original structures in the relational system towards a Non-relational system, means that these tables will also be joined in the new non-relational system in the same way as in the relational system, either to restore global information distributed in several tables or to derive new information not stored in the database. This situation will create a new relational database in a non-relational system that does not perform joins the same as in the relational system, plus there are some NoSQL systems that do not allow doing joins, also others try to consolidate all the data into a single table which organizes the data into columns. The relations between the tables build the fundamental element in the relational philosophy, also the operations of joins present the radical technique to exploit this kind of database, forming its force in the restitution of the global data, in the elimination of the redundancy of the data during its storage, in the optimization of the storage capacity and especially in the derivation of the new data not stored in the database, and at the same time forms its weakness, especially in the joining of the giant tables which requires a complex mechanism that creates an enormous volume of data in memory, and loads the processor with its processing to retrieve useful information. The relations in the relational system, represent its semantic and living aspects which encompass and hide at the same time a lot of information that describe the action and the operational aspect of this system and also the computing power. This semantic aspect is well neglected in all the previous approaches and also the relations are badly transformed if we consider the destination system of the transformation. With this in mind, we see that improper transformation of relations and foreign keys will waste significant parts of the data when transforming it from the relational system to another NoSQL and even makes the result database incomplete is useless! For this reason, we see that the relations in the relational system and in particular their transformations in the form of foreign keys in this system must have another form, which can be different from one NoSQL system to another, to maintain the data integrity, to transform the semantic aspect of the relational system to other NoSQL without conflicting with the specificities of the recipient systems. Indeed, to clarify our point of view, we will base ourselves on a practical case presented in figure 1 which proposes a conceptual model of data for the relational system. Figure 1 presents a model that has three elements: student, school, and registration. However, the two entities student and school model the students

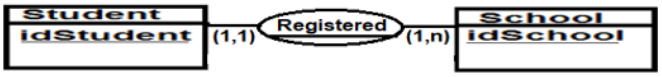

Figure 1: Conceptual Data Model

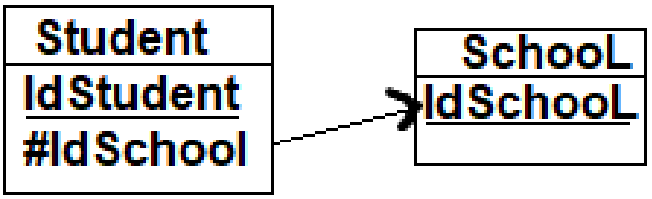

Figure 2: Example of a logical data model (MLD)

and the schools, while registration models the effect, the action, and the operation which link a student to a school, by forming the semantic and living aspect in this system, which always remains implicit and fundamental in this kind of system. This last element will disappear in the transformation of this model towards the logical data model presented in figure 2 and be replaced by a foreign key, but its spirit remains alive in the model thanks to the foreign key added to the logical model and will be restored by join operation. this semantic aspect is the real manner to restore the global information dispatched in several tables and at the same time the only way to derive other new information not stored in the system altogether. Figure 2 comes to illustrate the existence of the foreign key in such a model, to replace the relationship that has been between the tables and especially between their rows and that it is recoverable thanks to the join, without forgetting that the location of this foreign key determines whose table holds the relation. However, in NoSQL systems, we speak another language according to the category of the system. For example, the conceptual model above will be transformed in the relational model into two separate tables, and the relationship between them will be disappear and replaced by a foreign key in the student table, on the other hand in MongoDB as a NoSQL database using JSON files to store their data, we follow the logic of the objects, and therefore the best way to represent the same conceptual model is to create objects which materialize a single class called school which contains among its attributes a list of objects student without showing any foreign key, this if and only if the list of internal objects do not participate in any other relationship, so the sense of membership of the relationship, in this case, is the reverse of the relational model (see figure 2). This last way allows us to model the same logical model above, but without resorting to the need for the join operation for MongoDB, as well as the foreign keys will be superfluous and therefore make no sense in this case. Among the old researchers, we notice that they have done a copy/paste the relational system into the non-relational system, which makes it possible to lose the relational aspect in a non-relational system. Us, we believe that certain relations must undergo a particular transformation and that some foreign keys must disappear during the transformation of certain relations.

3. Methodology and Modeling for our Approach

3.1. Methodology for our modeling in our approach

We begin our approach with this crucial step, which will be the basis of our modeling and the phase fundamental of our analysis, for establishing our modeling toward the implementation phase at the end of this paper. Our methodology recommends following some steps to have a complete migration of the database from the relational system to the MongoDB system in an intelligent way. To this end, we will publish our analysis and modeling approach, which involves performing the following ten steps:

- The first is to identify all the elements that make up a relational database.
- The second consists of grouping all the elements of the relational database according to their category: data, structure, or semantics.
- The third is to identify all the elements that make up the migration target system.
- The fourth is to come up with a diagram that summarizes these elements and build a representative model.
- The fifth includes a presentation of the target system metamodel.
- Six is to reclassify these elements according to categories: elements that already exist in the target system (supported by MongoDB), elements that are already guaranteed by MongoDB (to be reassigned), and components to be ignored (their principles contradict the principles of MongoDB), components whose presence has no value and components that require special migration.
- The seventh consists of developing the modeling of the elements which require a migration to MongoDB and proposing a summary diagram.
- The eighth is the modeling of the relational database created from the elements of step 7, linked by the MongoDB model.
- The ninth step consists in developing a mapping between the two source and target systems as an operating result of steps three and eight. This correspondence will serve the implementation steps thereafter.

The following diagram explains as better the different previous steps and the sequence between them and their dependencies: This figure 3 makes it possible to present the different stages of our methodology explained above, as well as their sequence, also the stages that can be executed in parallel to arrive at our framework of the correspondence between the two source and destination systems of the migration, whose objective is to train the model of the database resulting from this migration

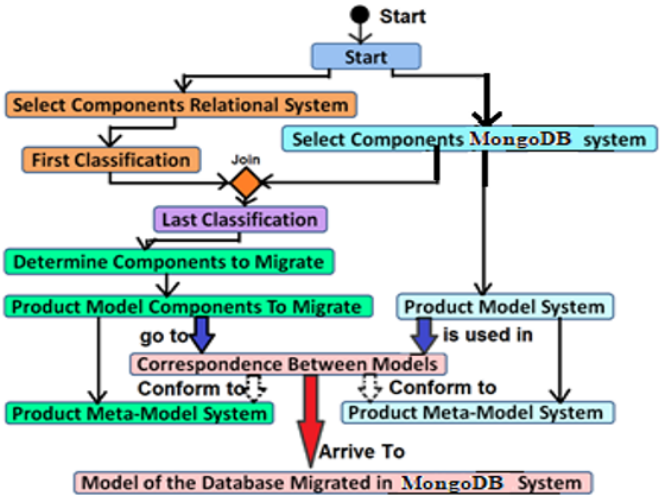

Figure 3: Our detailed methodology by steps and their dependencies

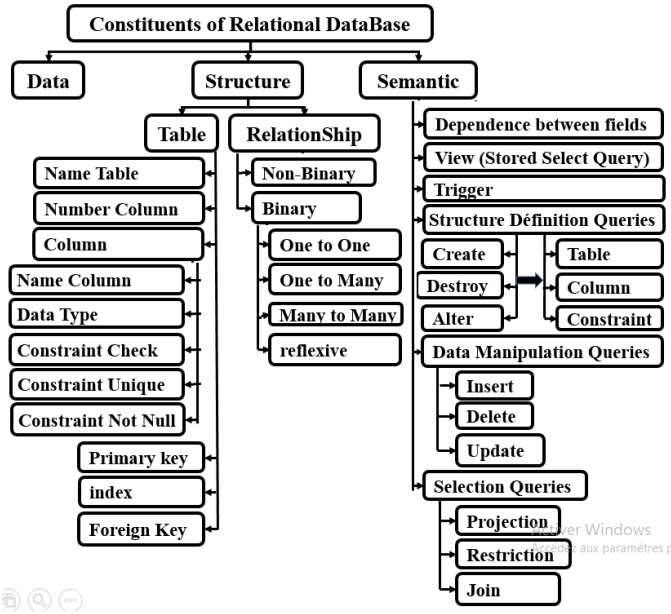

Figure 4: List of all components of relational database

3.2. Components Of Relational Databases And Their Modeling

To have a complete migration, we must list all the components of the relational database to process them and to define the manner of their transformations. With this in optic, we present the Figure 4: Figure 4 lightens an exhaustive list of relational system components to be migrated to NoSQL systems in Figure 4, classified according to their three expressive properties: structural, data, and semantic. NoSQL systems do not support queries that define and manipulate structures in relational database systems, because NoSQL systems implement simple and flexible information structures in their files, which can store many objects in different structures without any problem.

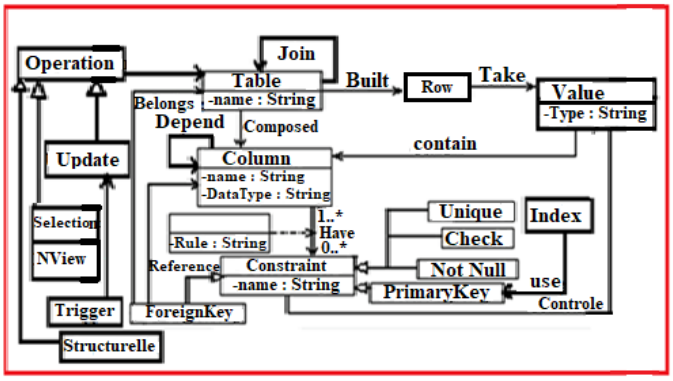

Figure 5: The relational database model

3.3. Modeling of the relational data model

After having presented the different elements of the relational system that we will migrate to the MongoDB system, we propose their modeling in the figure 5: Note that this modeling in figure 5 covers all the elements of Figure 4, including the triggers that bridge the gap between update operations applied to a table and programming at the database level. Thus, it covers the various constraints and the various relations between the tables and the other constituents. This modeling will play a very important role during the correspondence between the two pole systems of the migration following our approach.

3.4. MongoDB: Storage and Structure

MongoDB stores data as documents and collections, knowing that a document is just a JSON or BSON (Binary JSON) object and a collection is a set of documents by building JSON files. JSON represents the notation specific to the JavaScript language for representing data in the form of objects, whose constituents are formed by a set of key-value pairs. Keys act as attribute names of the object, defining the data structure of that object, while values represent the data stored by this object. In this case, it is useful to say that the keys are distinct words and specific to the object itself, while each value can be a numeric value, a word, another JSON object, a list of the values of the same type, or heterogeneous types including other JSON objects, in turn, can be of different structures. This storage philosophy can serve the transformation of a row of data to a JSON object by forming a Document under MongoDB. This last object will be of simple structure and with simple values. This allows us to transform all the rows of a given table into a collection of JSON objects, with the possibility of extending each object by other fields, which can have non-simple values. As well as the horizontal link that can be found between two lines of two different tables can be transformed into two JSON objects according to a vertical inclusion relationship. Contrary to the spirit of object-oriented, which requires the definition of an abstract model called the class to represent the structure of all objects created based on this model, JSON allows you to create your objects without resorting to an abstract model, which describes their structure, and each JSON object has its description independent of other objects. However, a

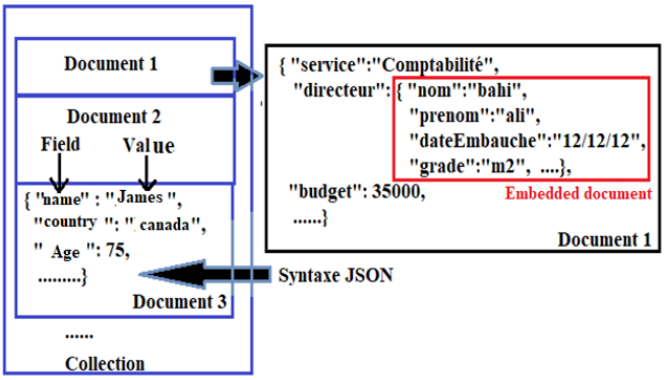

Figure 6: Logical storage structure in MongoDB

JSON file forms a data exchange medium that is light, simple, readable, flexible, understandable, and usable by humans and by different programming and data representation environments. This major advantage is the source of the success of MongoDB. At the same time the pivotal element of the success of our migration approach because JSON is usable by any operating system and any development environment, such as Java, Microsoft, Oracle, UNIX, etc., without forgetting that it can be extended by JavaScript programs to simulate certain processing which brings the semantic aspect of relational databases closer. Thanks to its basic language (JavaScript), you can implement regular expressions to enrich the schemas under Mongoose to add and simplify the control of certain data, bringing closer certain control functionalities implemented in the management systems of relational databases. In the figure 6, we show a simple data structure design to approximate the storage model of MongoDB. Figure 6 illustrates the storage model presented in the previous paragraph, showing a collection that contains three documents as examples of data storage. The first document presents an example formed by Key and Value pairs and that a value is a JSON Object too.

3.5. Architecture and Functioning of MongoDB

This section aims to describe the operating mechanism of MongoDB, as well as its internal architecture, to analyze its processing and operation, and this is to be able to propose an adequate transformation to the few parts of the semantics of the relational management system database. For this reason, we will limit ourselves to what will serve us in achieving our objective and briefly review the rest of the MongoDB mechanism. We can summarize these features as follows [17]:

- **Data container management**: MongoDB allows for horizontal scalability, as table joins in the original MongoDB philosophy are not essential as they are in conventional RDBMS, intending to improve the overall system that manages the data. In addition, it adopts a bursting policy, which allows a linear scaling of the cluster, to give the possibility of adding more machines. Thanks to the bursting, it is possible to bear the increase of the unexpected load on the web and increase the efficiency of the system.
- **Data management**: With this in mind, MongoDB allows the replication of sets of collections, which is the grouping of servers that maintains the same set of data to increase data availability.
- **Data processing**: MongoDB has within each machine a data server, which is part of our data container, storage, and processing engines. Each server in MongoDB has four different mechanism engines to achieve its mission, and among these engines is the WiredTiger engine, which does concurrency control and native compression properties among its tasks. There is a default WiredTiger engine that provides the best competition performance and storage efficiency and native compression. In addition, it serves for better storage efficiency and performance.
- **Distributed database**: MongoDB has the property of automatic partitioning, and through it, multiple replication server nodes are added to the system. It allows very large databases to be split into smaller, faster, and more manageable parts called shards.
- **Data search and identification**: it uses for documents and sub-documents indexes on one or more attributes to make access to this database faster.
- **Communication between servers**: MongoDB adopts the Master/Slave topology between its data server machines, so a master server must be designated in the establishment of this topology to organize and manage communication between these servers, especially in the management of shared data between several servers or in the management of data replicated under several sites. In addition, this topology must manage the role of the servers in the event of a failure. This mechanism is the most useful for our migration approach. Indeed, when it comes to a data extraction request, the master server must replicate it for all the slave servers, to gather and integrate their responses, the objective of which is to reformulate the overall response. On the other hand, when it comes to an update request, then this request goes through a system that checks whether there is a schema or a model to respect for this update and if it is compliant, then the master server reaches the various slave servers to execute it will propagate it. This mechanism balances the load between servers and applies queries in a partitioned way under small simultaneous tables instead of dealing with a single giant table. In this mechanism, we take the opportunity to enrich the schema or the model to be respected in the case of the update requests to serve the transformation of the referential integrity constraints or the other constraints, as well as the triggers, which we will deal with in the following.

After revealing the essential points of the functioning of MongoDB, quoted above, we propose the figure 7: Figure 7 models the architecture and the operating mechanism of MongoDB. In this architecture, we have modeled any clients, including applications, by an application, that types and sends a request to the

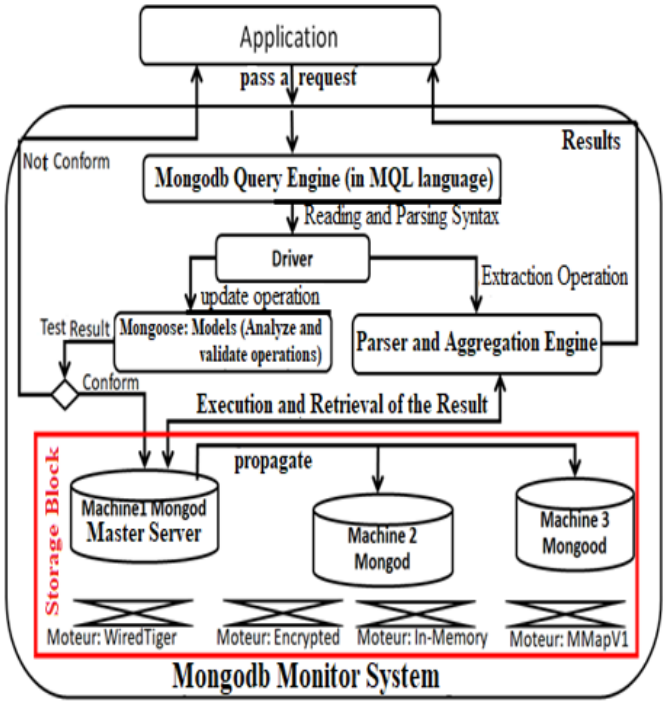

Figure 7: The simplified system architecture of MongoDB with Mongoose

MongoDB system. The latter will pass this request to a lexical and syntactic analyzer to check its validity at the lexical and syntax level, and to transmit it to the engines to evaluate it. After the evaluation of this request, it will be transmitted to the master server to launch its execution if it is an extraction or it will be transmitted to the Mongoose if it is an update operation. In the last case, if it complies with the models concerned and is defined at the Mongoose level, then it will be transmitted to the master server to launch its execution, otherwise, it will be rejected by the system.

3.6. Mongoose: towards improved structuring

Mongoose is a JavaScript library, which works with MongoDB and its runtime environment. It allows the definition of the different schemas and models of the documents stored in MongoDB using JavaScript Node.js. It is equipped with positive features and characteristics that make this system efficient and increase its power, its scalable aspect, and its security component. Mongoose when installed in the MongoDB server, it allows you to define schemas, which are implemented by the system, and that they are respected when creating and manipulating documents stored in MongoDB. When defining these schemas, Mongoose can define required attributes that must exist in all documents and other attributes that must be unique or respect a regular expression. Also during schema definition, Mongoose can control CRUD operations by adding to them the "validator" attribute, which can be set to either "true" to allow this operation or "false" to cancel it. In the latter case, Mongoose can define JavaScript functions that return the value either "true or false" following a specific treatment applied to the documents concerned either to authorize the actions in question on them or to cancel them. This last mechanism makes additional processing to better control the various CRUD actions, that will be applied in MongoDB documents. This mechanism can simulate the same mechanisms seen in relational database management systems, such as integrity constraints, foreign keys, policies for violating these constraints such as (ON update set Null, etc.), check, unique, not Null, default, identity constraints and controls applied by triggers in the relational model. The characteristics: of flexibility, suppleness, simplicity, and understanding of JSON, as a means of storing data, are the strength and the weakness of this system at the same time. Indeed, the definition of a data container that accepts storing any information under heterogeneous structures complicates the analysis, extraction, and derivation of the data because of the absence of axes of projection and research that offer the unified structure applied for the different Documents. Thanks to Mongoose, we can endow the JSON files of MongoDB, with given structures that can define obligations for one or more attributes, as it can leave JSON documents free without any structure. This situation allows keeping flexibility in the structuring of the documents or allows defining of a certain level of obligatory structuring, forming a new power of MongoDB by the fact of changing and managing the level of structuring requirement, contrary to the relational database management systems that have only fixed and constrained structures. Therefore, Mongoose plays an interesting role as a flexible system, which can go from an unstructured level to a very interesting and rich structure level. MongoDB can use Mongoose, to better store and manage their documents. Mongoose allows you to define schemas of documents, which especially helps the analysis and exploitation component. In the parts, which come, these diagrams play other roles, which we have to define the continuation. The figure 8 shows how mongoose works with MongoDB: In figure 8 above, Mongoose can have multiple schemas. Each schema validates a type of document, by defining its structure with or without a validator. The validators used in the schemas are fields that can have the value true or false following a condition or the Boolean return of a validation program. This situation will help us, particularly on the semantic level, to develop our complete and intelligent approach to migration, especially in the transformation of foreign keys and their integrated mechanism.

3.7. MongoDB Design and Modeling

In this chapter, we present the results of the previous chapters. They use the concepts and principles developed and exploit the architectures, structures, mechanisms, and basic designs previously proposed, to propose a modeling of a database under MongoDB, which installs Mongoose. We will consider that the database to be modeled under MongoDB is managed by a single server machine and therefore, we propose the following design, which will be extensible by the design linked to the mechanism of databases distributed in several data servers according to the need for extension and evolution by other approaches or other research studies. For this purpose, we propose the model and the meta-model of MongoDB in the figure

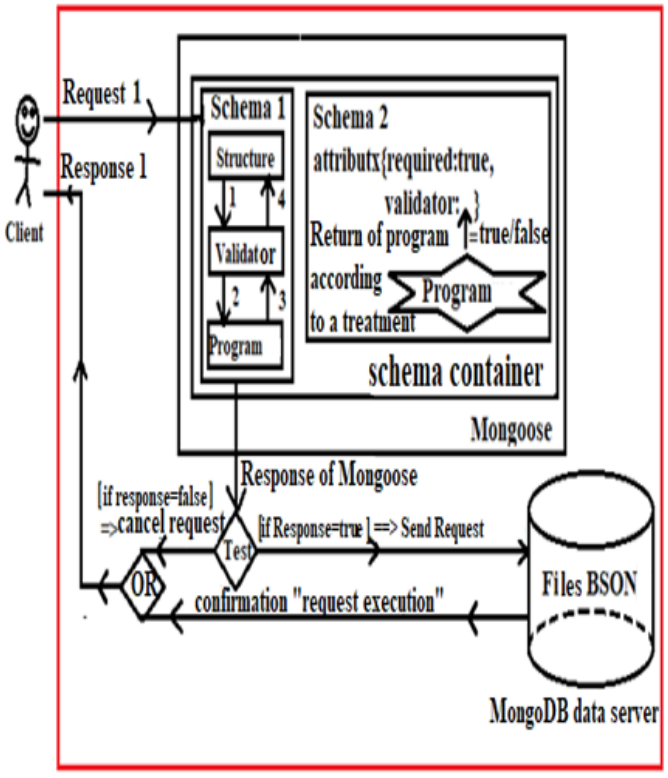

Figure 8: The Functioning of mongoose with MongoDB

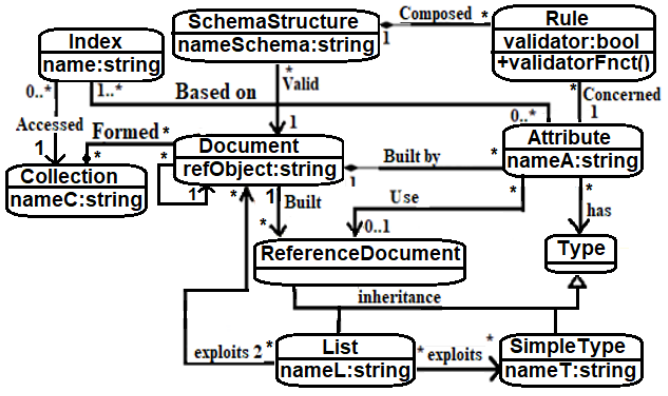

Figure 9: The conceptual model of MongoDB

9 and in the figure 10: Figure 9 presents a diagram that models the MongoDB storage model by showing that each data collection can have its index to simplify and improve the search for the documents that constitute it. Each index is created by one attribute or two for the double indexes, so these attributes are the keys of the search inside the collections, knowing that these attributes are of simple or complex type. Some attributes can reference another document in another place. We need a schema for each type of document to validate its structure and content based on rules formed by conditions or uses of a specific treatment. Based on the conceptual data model of MongoDB developed in the figure9, we can propose the logical data storage model of MongoDB, presented in the figure 10: Figure 10

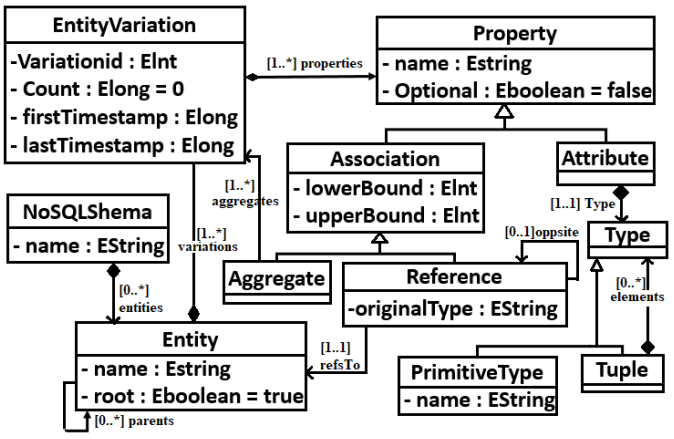

Figure 10: The MongoDB logical Meta-Model [18]

presents a logical storage model established under Mongoose to master storage under MongoDB. This modeling in Figure 10 shows that MongoDB's NoSQL system is built by entities that materialize models that represent "EntityVariation". this last element is formed by properties with the possibility of using aggregations, which in turn represent groupings of properties. According to figure 10, a property bears its name and represents an attribute or an association and can be a simple type, list, or reference to another object with another characteristic that specifies its level of obligation of its presence and the cardinalities established between related objects.

3.8. Mapping of relational database components to MongoDB

We present in this part, the result of our analytical work to define the scope of our migration approach, as well as the transformation of the elements to be migrated in the table 1, to define the correspondence between the components of the relational database and those of MongoDB:

Table 1 illustrates the list of components that make up the relational databases and their projection in the MongoDB system, which offers an equivalence of some components, allows to redeploy of others, and presents the rest of the components which have two types: components without added value to databases established under MongoDB and components that require a new transformation to be developed. Also, the third column presents the correspondence, according to our approach, of the first two types of components. After carefully reading Table 1, we can assume that our migration approach can convert some components directly,while others require them to be converted and migrated to their respective roles in the target system. In the diagram below, we represent the components that need the new transformation into different forms to accommodate the new system and its roles: Figure 11 shows just the items in Table 1 that require a special migration to develop again. These elements are the relationships between tables and the join between them that represent the semantics of relational systems, as well as the triggers and views that represent the elements that guarantee the proper functioning of relational systems. Figure 11 frames the special work that can be an axis of criticism of our

Table 1: Transformation of Relational Components to MONGODB

Element in RDBMS	In MongoDB or Mongoose	Elements in MongoDB
Records(values)	Exist	Object (values)
table	Redeploy	Collection
Row	Redeploy	Document
Name of table	Redeploy	Name of collection
Column	Redeploy	Field
Column Name	Exist	Field Name
Data type	Redeploy	Data type
Check constraint	Redeploy	Mongoose + schema+ attribute Validator+ function JavaScript
Unique constraint	Redeploy	Mongoose + schema + attribute unique
Not null constraint	Redeploy	Mongoose + schema + attribute required
Primary key	Redeploy	identifier when creating the document object
Index	Redeploy	index
Foreign key	Redeploy	Foreign key
relational Ship	To migrate	To migrate
Dependence between fields	Redeploy	Schema in Mongoose
View (stored select query)	To migrate	To migrate
Trigger	To migrate	To migrate
Structure definition queries	Not supported and without added value	not supported and without added value
CRUD operation	Redeploy	Insert, delete , update, and find operations
join tables	To migrate	To migrate
usual functions	Redeploy	aggregation, time management, and string functions

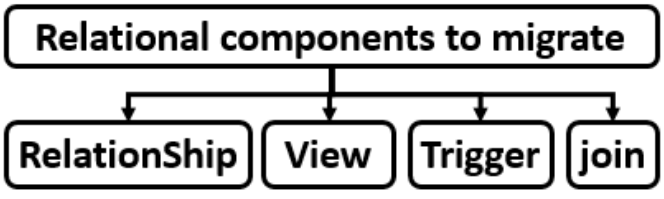

Figure 11: Elements to migrate and find its implementation in MongoDB

approach to improve it by other scientific researchers. During this section, we unveiled our approach to analyzing and modeling the two pole systems of migration: source and destination, by presenting the elements of the source system to be migrated, then the proposal for a storage model to simplify the tasks, which come after, as well as the data storage models for the other system, without forgetting the modeling of the functioning of MongoDB and Mongoose to understand our projection system better. Finally, this part is concluded with the proposal of our philosophy of correspondence between these two systems. The work during this section builds our guide which will guide us in the steps that follow, thus delimiting the perimeter, the framework of our work, and the lines of action for our integral approach. This Analysis and Modeling form a research stage, presented to the community of researchers either to use it for other reasons or to criticize them to improve and enrich it.

4. Foundations of our TMSDRDND Migration Approach

4.1. Alignment and Adopted Principles

Before starting to define our approach, which is based on the analysis and design made in the chapters above, we have deemed it useful to outline a set of principles and characteristics to be attributed to our approach for the highlight. These characteristics and principles align with the development of the approach, and during its stability phase reach the maturity phase. These characteristics and principles can be summarized:

- **Modular**: this is the first principle adopted which makes it possible to divide the overall objective and the integral treatment of the approach into sub-parts, to serve the simplicity of its perception, the understanding of its objective, their maintenance, their improvement, their evolution, their correction and criticism by other researchers or interested units, as well as its development, the verification and validation by unit tests, the speed of execution because they make it possible to launch the execution of several parts simultaneously and to increase the performance
- **Structured**: to improve the internal architecture of the approach and facilitate its design and implementation.
- **Global**: because it covers all elements of the relational database.

When designing its foundations, and given that MongoDB as the destination system for this migration uses JSON as the data storage base, and Mongoose to mount the data structure, as well as JavaScript or NodeJS as the language to enrich the Mongoose schemas and to add other features that do not exist in the native MongoDB system, also without forgetting the characteristics and principles mentioned above, we have decided to divide our approach into three axes, the first two of which run in parallel and that these three axes complement each other to end up in a single point. These axes will attack the structural component which is oriented towards the creation of schemas equivalent to the data structures of relational systems under Mongoose, the second axis aims to recreate the semantic components with the appropriate transformations, in JavaScript or NodeJS functions and programs in one file which is separate from the first file of schemas, to build the functional and semantic aspect of BDDs in the MongoDB system, as for the third axis, it will use an

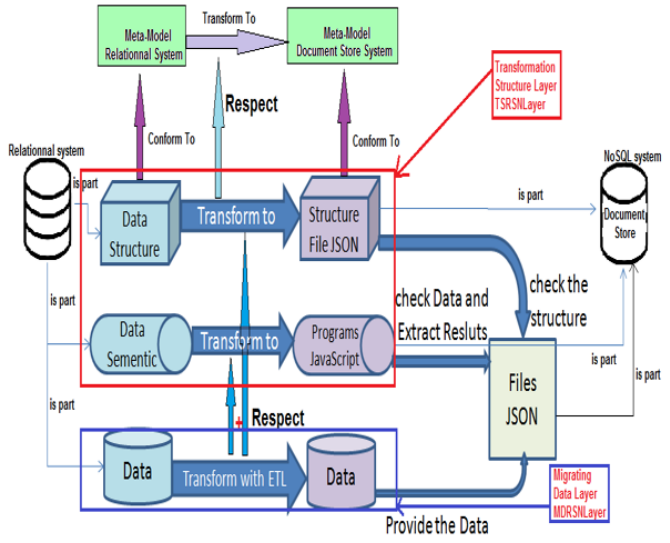

Figure 12: The architecture of "TMSDRDND Approach"

Table 2: Matching Table Constituents in MONGODB

Element	Corresponding
Table	Collection
Column	Field
Row	Document
Data Type	Type of each field
Dependency between fields	Schema in Mongoose
Null values	Deleting the Field from the document
Automatic fields	Field with a value returned by a JavaScript function
Constraint	will be covered in the next section

ETL to extract data from relational databases and perform the necessary transformations using results from the first two axes to load the data into JSON files, forming the database resulting from the migration.

4.2. Global Architecture

The architecture of our approach, named "TMSDRDND Approach", aims to define a precise framework for its processing, which subdivides the databases into three types of data, each of which is transformed according to a specific model to MongoDB system. This architecture is clear in the figure 12: Figure 12 above presents the architecture of our overall approach, showing its layers and steps, as well as the results that will be established following the models and meta-models developed in the previous chapter. For the TMSDRDND Approach to be complete, it must be based on the models that describe the relational system and the MongoDB system, and in turn, they must be following the meta-model of each system. In the definition of the TMSDRDND Approach, we will go through two stages: the first, in the "TSRSNLayer" layer, which aims to transform the structural and semantic data into the MongoDB model according to a set of transformations, which respect the model defined previously, by producing a file which controls the structure and another in JavaScript which contains the functions stored and ready to be used. While the "MDRSN Layer" layer, consists of using an ETL, which will begin its processing by extracting the data according to the proposed models, then it carries out the necessary transformations according to the transformation rules of the "TSRSNLayer" layer. Finally, it loads the data from the source into JSON files by exploiting the result files of the processing of the "TSRSNLayer" layer. During processing in the first layer, two operations are launched: the first to process the structural component and the second to process the semantic component. These two layers cannot run in parallel hence the reason for the separation into two layers. In the following paragraphs, we will reveal the different rules for transforming each element of the relational database into the MongoDB system according to each component, whether structural, semantic, or data.

4.3. The structural component of the TMSDRDND approach

4.3.1. Transform tables

The tables in a relational database represent the most interesting structure that includes the database data, which will be exploited by the other components of the database. Note that this table has a name, and a set of columns, each of which has a specific name and type. This structure implicitly defines the dependency between the different fields of the table. Tables also contain the rows of data and other rules that ensure that the data is valid. For this transformation, our approach proposes the table 2 to define the corresponding of each element of the table:

considering that the table is a main element of relational systems by representing data containers; Table 2 above presents the conversion and the correspondence between its elements towards the constituents of the MongoDB documents developed in the models of the previous chapter. Since the different elements of the second column of the table already exist in the MongoDB system, we can consider that the table 2 defines the different rules for transforming the components of a table from the relational database to MongoDB, respecting the specifics of MongoDB. This correspondence will be applied during the transformation phase of the data structure. tables in the relational model use a fixed structure, which will be applied to all rows of data. In the case of the absence of data in a line compared to the structure of the table, then the system stores in its place the NULL value to fill its reserved space. However, in MongoDB, storage is based on documents in JSON files, whose structure is flexible and does not add NULL values to documents. For this reason, a transformation has been proposed that processes tables vertically and at the same time horizontally, considering the particular cases of automatic columns, constraints, NULLS values, and the other elements of Table 3. This way will make it possible to transform each line into a Document and the whole of the table into a collection of documents. At this level, there will be a program that takes care of

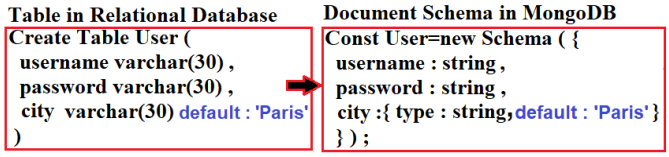

Figure 13: Transformation of ”default value” constraint in MongoDB

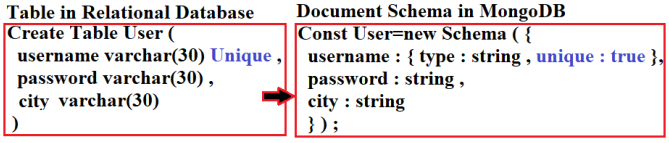

Figure 14: Transforming the “Unique” Constraint in MongoDB

applying the rules of Table 3 during the transformation of each Document, which guarantees a total and fluid transformation of all the tables.

4.3.2. Transform Rules: default, unique, Not Null, and check

In the definition of the tables, sometimes to check the validity of the data to be entered for some columns, we must add to some columns rules that control their data. For example, entering a negative age makes no sense and can generate false results during calculation operations. To this end, database designers have created check rules to define the validity interval of data to simplify its control. Also, they created a unique constraint, not null and default to better control the domains of validity of the data. In this section, and when defining the rules for transforming the various components of the relational database to the MongoDB system, we will transform each element when defining the schemas of the corresponding documents thanks to the attributes offered by Mongoose. In the figures 13 to 18, we present the transformation rules of these constraints in the form of examples:

- The constraint that defines the Default value:

Figure 13 illustrates how to transform the ”Default” constraint applied by relational systems. The addition of this constraint in the mongoose schemas allows the application of the same rule of the default value applied to a data column in the relational system. This constraint will force the existence of a data field in all the documents formed based on this schema, and if the document is provided without this value, then the system will add this mandatory attribute with a value proposed by default. This mechanism allows us to not lose this information building the structure of the relational systems during the migration, by finding its corresponding and at the same time allowing us to keep the existence of certain attributes so that they are used during the analysis.

- Constraint defining the uniqueness of the Value in the column:

Figure 14 illustrates how to transform the ”unique” constraint applied by relational systems. The addition of this constraint in the mongoose schemas allows applying the same rule of the unique value applied to a data column in the relational system. This rule made it possible to store a set of data and at the same time preserve the uniqueness of some information for each stored document. This mechanism can serve the search by this attribute later

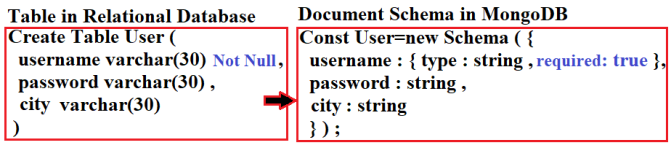

Figure 15: Transforming the “Not Null” Constraint in MongoDB

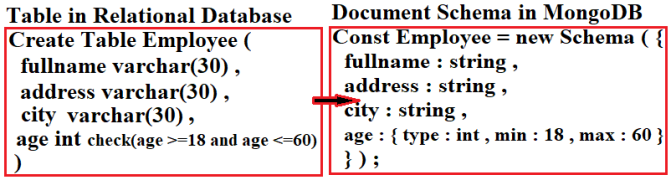

Figure 16: Value interval constraint transformation

- The constraint that prohibits the null value in a column (Not Null):

Figure 15 illustrates how to transform the ”Not Null” constraint applied to values of one column by relational systems. Adding this constraint in Mongoose schemas enforces the same non-empty value rule applied to a data column in the relational system. This rule made it possible to oblige the stored documents to have this attribute obligatorily and with a value. this rule makes it possible to specify the minimal structure of the documents of this schema.

- Check the validity domain of the data (Check constraint)/For data that belongs to an integer interval:

Figure 16 illustrates how to transform the check constraint applied by relational systems to define a validity interval for numerical data. Adding this constraint in Mongoose schemas enforces the same rule applied to a column of data in the relational system to define the valid range of data stored in all documents. This rule made it possible to apply functional rules that meet the requirements of database users so that the information stored in the documents makes sense and allows us to make decisions. The figure 16 shows an example in which the valid range of data is based on numeric values, which we can specify as their minimum and maximum value.

- Check the validity domain of the data (Check constraint)/For data that belongs to a list of values:

In figure 17, we present our transformation to define valid data, this rule allows you to specify the valid data, but this time in the form of a list of accepted values.

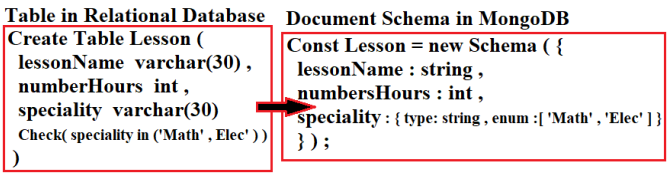

Figure 17: Transformation of the constraint that checks membership in a list of values

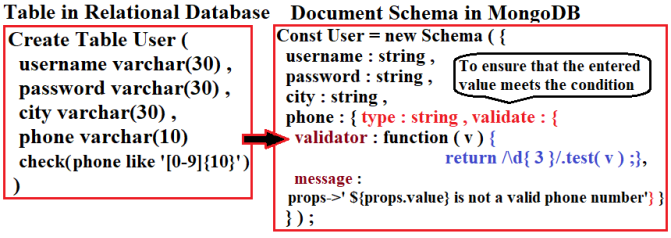

Figure 18: Transformation of the constraint that checks regular expressions

- Check the validity domain of the data (Check constraint)/For data that respects a rule or regular expression:

Figure 18 illustrates how to transform the "check" constraint applied by relational systems, which aims to define regular expressions to validate the data to be stored according to a specific field. These rules and constraints come for more control over the data to be stored in our database and to offer more credibility and reliability of the stored data, in projection in the functional requirements of the database. These constraints represent information from the database that frames other information. Older researchers neglect this information in their migration approaches. In our approach and thanks to Mongoose, we found an adequate transformation to these constraints for MongoDB to keep the advantages offered by the relational model.

4.3.3. Transform primary keys

The primary key in the tables of relational databases plays the role of the identifier to identify one row among several, and therefore its value must be unique and not null to meet the need. Generally, relational databases build an index for each primary key. In MongoDB, we find that for each newly created document, MongoDB assigns it a unique and non-zero identifier to identify it among several. This is exactly the role of the primary key in the relational system, except that sometimes the primary key in the relational system carries in addition to its role additional and useful information, which does not exist in the MongoDB mechanism. So in our transformation, we keep the identifier assigned by MongoDB and we add a new field to the documents to represent the transformation of the primary key from the relational tables. For this transformation, we propose the following method which transforms the primary key into an ordinary field with unique and not null characteristics, without forgetting its index. But for document id, it will be added automatically by MongoDB. The figure19 shows an example that explains this transformation: Figure 19 illustrates

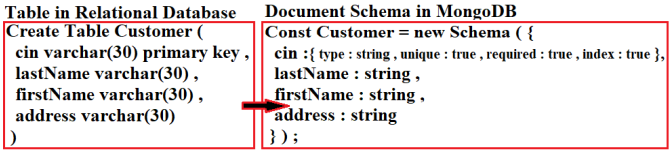

Figure 19: Transformation of the primary key in MongoDB

how to transform the primary key applied by relational systems. This figure shows that the transformation of the primary keys is a composition of the previous transformations of the Not Null and unique constraints with the establishment of an index linked to this primary key. This mechanism is implemented automatically by relational systems to guarantee that information having a meaning and generally entered by a user, is unique and not empty throughout the data container so that it is the identifier of a line of data and the base of the search indexes. This mechanism does not exist in MongoDB, and for more precision, MongoDB adds for each Data document new automatic and unique information as an identifier in the form of a code without any meaning whose objective is to keep the uniqueness of this document. this last mechanism makes it possible to store the same document several times without any control over the data contained, but rather MongoDB adds for each one a new code so that it is the identifier without any final intention. However, the notion of the primary key relates to a verification of uniqueness based on the stored data themselves and is no longer based on new information added systematically. This can generate errors when extracting or deriving new data because of ambiguities during processing. Older migration approaches neglect the transformation of primary keys and treat them as ordinary attributes. Thanks to the Mongoose schemes, we managed to find an adequate transformation to the primary keys to offer their reasons for being in the relational system alongside the identifier offered by MongoDB.

4.3.4. Transform foreign keys

Foreign keys in relational databases and MongoDB

Foreign keys in relational databases play a very important and complex role. Beginners believe they are simple fields that carry values like any other. However, foreign keys refer to other information in other tables or the same table, representing a relationship or association between several tables, and defining rules that control the addition, update, and deletion of rows in different tables, linked together by these foreign keys. These last rules are named by the referential integrity constraints. Referential integrity constraints play a very important role to avoid entering unmatched data into the other tables, the purpose of which is to link the tables so that they receive data that describes a single global entity, but this information is dispatched on several tables according to their structures and fields of information. This relationship, which represents an association between several tables and is materialized by the primary key and the foreign key, can be used in several forms to generate a lot of direct or indirect useful information. This is the power of the relationship system. In relational databases, it is possible

to enable or disable the mechanism related to foreign keys, and also possible to change this mechanism a bit by what is called a violation of referential constraints, whose objective is to override the rejection of a delete or modify operation in a table related to the primary key of another table by setting to NULL as value, set to a default value or propagation of the triggering update operation this event. The mechanism related to foreign keys is very powerful, which aims to harmonize data between tables, and that an operation on one table can affect another, to create integrity between the data entered, also to keep a level of the data link between the tables, to not have data in tables which do not have a continuation or complement in the other tables. This says that the tables even if they are visually independent, basically are linked together or functionally form a single table without data redundancy and with minimal storage. According to this concept, it can be said that thanks to the foreign keys, the tables form a single table with articulations, which escape the duplicate data and increase the optimization of the stored data with a definition of the dependency between the fields, the determination identifiers of the units stored in the tables and the creation of supervisors to supervise the rows of the tables in relation. This mechanism subsequently made it possible to generate more information not stored in the tables and to develop the model and the data exploitation languages and other very interesting tools. Unfortunately, this concept does not exist in MongoDB, nor through Mongoose, and MongoDB users add fields in the same way just to create the opportunity to be able to link rows from these tables to render dispatched information across multiple data collections or to generate other useful information. To better understand the notion of the foreign key, we can summarize the functionalities of this special field as well as the operations linked with it by the following list:

- Linking rows of related tables
- The link between the two tables
- Supervision of CRUD operations of related tables
- Reading data dispatched between tables
- Generate other information not stored in tables and expand calculation possibilities
- Prohibit certain operations
- Propagate or correct other operations and information (properties on update cascade, etc.)
- Verify data integrity (types, size, and match)
- Admit the activation and deactivation of its service
- Allow creation or deletion operations.

In this phase, it is legitimate to ask the following question: what is the point of recreating the same concept of the foreign key in a MongoDB database? In MongoDB, we can add a field to a document by specifying that it is referenced by another field in another document (the form of foreign keys in MongoDB),

```
// User Schema
const userSchema = mongoose.Schema({
    username:{
        type: String,
        required: true
    },
    password:{
        type: String,
        required: true
    },
    company:{
        type:{type: Schema.Types.ObjectId, ref: 'Company'},
        required: true
    },
});
```

Figure 20: Setting a foreign key in MongoDB by reference

but for MongoDB, it stops there, and it doesn't have any additional effect. Of course, you can notice that in this case the two fields are linked, but the action on one of the two fields of the first document will have no effect on the other field of the other document, which allows us to say that even this writing is possible, but the two fields are independent and each can have any value without any effect on the other. In MongoDB, a document can play the same role as a row of data in the relational model. By the same logic, we can say that documents in MongoDB are not linked to each other even if they have fields whose definition references another field in the other document. In the figure 20, we present an example that defines the writing of the foreign key in MongoDB: Figure 20 illustrates how to define foreign keys in MongoDB using schemas in Mongoose, providing an example of a schema whose name is "userSchema" to enclose the Documents represented by this schema. The schema of the example in figure 20 includes an attribute named "company" to reference another document from another schema named "Company" and its type is exactly the type of the identifier of this latter schema. At this level, it is fundamental to say that the power and the spirit of the relational system are contained in the associations and their operations. These associations will disappear from the conceptual level to the logical level during Merise modeling and are replaced by one or more foreign keys so that they are exploitable by the join operation at the time of data selection to reap its fruits. From this explanation, it is clear that a transformation of a foreign key is part of the transformation of a trace of a relation between two tables. To eliminate confusion, we explain each feature of the foreign key with a simple description as follows:

- The link between the rows of related tables: it allows you to define a link between two fields, the first of which plays the role of a primary key and the other the role of the foreign key.
- The link between two tables: through the link between the primary key and the foreign key, it is clear that one contains data and the other contains their complement of

data.

- Allow or refuse creation or deletion operations: it prohibits the deletion of a row from the source table as long as the value of the primary key is used in the other table as the value of the foreign key, also it prohibits the entry of a foreign key value without it existing in the other table as a primary key.
- Supervise CRUD operations in related tables: Monitor CRUD operations in foreign key-related tables to verify data and achieve the previous functionality.
- Render the data dispatched between tables: it allows the link of the rows of the tables in relations by using the similarity of the values of the primary key and the foreign key to building a long row of data and form from the two rows coming from the two tables.
- Generate other information not stored in the tables and expand the possibilities of calculation: it allows the application of various conditions between the values of the primary key and the foreign key to building each time a new calculation formula according to the need.
- Propagate or correct other operations and information: By violating constraints by adding one of the properties: on delete set to null, on delete set default, on delete cascade, on update set null, on update set to default, on update cascade
- Verify data integrity: the foreign key requires referential integrity constraints that ensure equality of data types, sizes, and values between the primary and foreign fields
- Admit the activation and deactivation of its service: the existence of the foreign key implements the functionalities mentioned above, which makes the insertion, deletion, and modification of data quite restrictive, which blocks the system in certain cases particular, and makes this mechanism beneficial, it is possible to activate or deactivate it as needed without it being deleted.

From this, it is clear that writing the foreign key in a document under MongoDB only favors the restitution of data sequences dispatched in several documents, as well as the generation of new data not stored in the documents. In the figure 21, we present the different advantages of the foreign key in the relational model: Figure 21 comes to offer an overview of the functionalities offered by foreign keys and the implicit mechanism implemented by relational systems to guarantee the role and operation of foreign keys. From figure 21, it is clear that foreign keys represent the only way to link the rows of data to extract the integral information dispatched between the tables linked and the containers, and also to derive other new information, without forgetting that foreign keys form the means of relational systems to guarantee consistency by supervising CRUD operations in tables linked together by foreign keys. Based on the figure 21 and before diving into the proper transformation of the foreign key, it is useful to discuss their functionalities

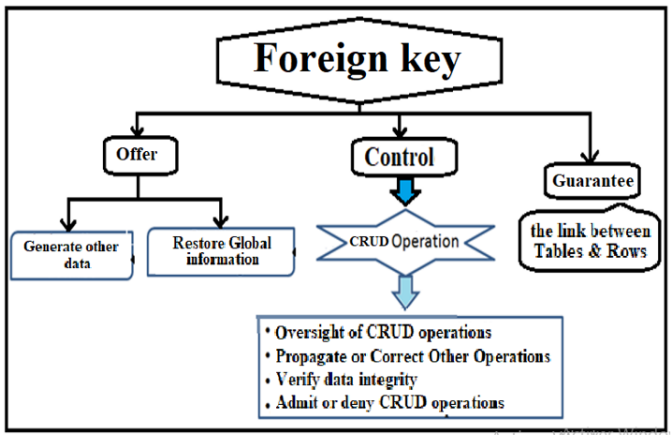

Figure 21: Foreign key features and mechanisms

concerning the destination system (MongoDB), to see which functionalities require a transformation to MongoDB and the others which their transformation does not have any benefit in the future database in the destination system.

The projection of foreign key functionality in the MongoDB system

After watchful reading of the two systems, it is clear that:

- In the relational model, the foreign key defines a relationship between two tables to supervise the CRUD operations applied to these tables to keep the consistency between their data by a close link between their rows. If we want to transform this mechanism in MongoDB, then we will have difficulties at the level of collections (Corresponding table in MongoDB), because they bring together documents of different natures and structures. In addition, documents can have different data and structures. At this level and thanks to Mongoose, we can define a schema that defines a structure for Documents that represent the same species or the same class of data. In this structure, you can define a relation between the structures of all documents. Mongoose does not allow the definition of a structure on the collections, and therefore one cannot define a binding between the collections at the MongoDB or Mongoose level, so the binding between the documents is largely sufficient to replace the bindings defined by the foreign key.
- Thanks to Mongoose, you can add a control that controls the validity of a document when creating, modifying, or deleting through a method to be developed. This method will offer control of CRUD operations in all the concerned Documents, as well as their descendants according to the figure 8.

For this, we propose the following list which summarizes an analysis of the nine features of the foreign key cited above to have for each feature a value among these three values: exists, needs to define its transformation, or has no value added in MongoDB.

Table 3: Actions Embedded in Foreign Keys

Functionality	Positioning in MongoDB
Linking rows of related tables	To migrate
The link between the two tables	To migrate
Allow or deny create or delete operations.	To migrate
Monitoring CRUD operations of related tables	To migrate
Restitute data dispatched between tables	Exist
Generate other information not stored in tables and expand calculation possibilities	Exist
Propagate or correct other operations and information (properties on update cascade, etc.)	To migrate
Verify data integrity (types, size, and match)	Without adding value
Admit the activation and deactivation of its service	To migrate

Table 3 presents an exhaustive list of foreign key features and classifies them into three categories: features that already exist in MongoDB, features that require a specific migration to create real foreign keys in MongoDB, and the rest of the features whose transformation are irrelevant for MongoDB.

The transformation of the foreign key to the MongoDB system

To transform a foreign key from a relational database table to MongoDB, this field must be added to the schema defining the documents of the same species, with the addition of a validator as an attribute in this schema, to validate or invalidate the CRUD operations applied to their documents. This validator is developed as a JavaScript method to concretize the mechanism of the foreign key. This validator should be added to all document schemas affected by this foreign key relationship, each of which monitors one or more CRUD operations. By bringing this mechanism closer, we add validators to the documents that hold the primary key to control the modification and deletion operations, also for the schema that holds the foreign key, we add validators to control all possible update operations. In the figure 22, we present a small example that shows the general structure of this transformation with the validator field and the foreign key field: Figure 22 shows how to transform foreign keys by our approach, exploiting the MongoDB way explained in figure 20, and adding another validator attribute to guarantee the implementation of JavaScript functions to represent the features of the foreign keys presented in Figure 21, as well as to migrate the functionality explained in Table 2. The JavaScript functions represent a direct implementation of several implicit functionality and operations that we need to develop as part of this transformation. These are attributes with an automatic mechanism implemented by the relational system to manage the update of dispatched rows in several tables linked

```
User . create(
  { username: {type: string , required:true, unique: true},
    email: {type: string, required:true, unique: true },
    password: {type: string, min:8,max:20, required:true},
    service:{type:Schema.Types.ObjectId, ref: Service, validator: v=>{
                if(service.find({'ObjectId':v})....
                    return true;
                else
                    return false;
            }
    }
});
```

Figure 22: Our foreign key transformation according to the TMSDRDND approach

together via these attributes, intending to guarantee consistency between the stored data. Also, they play a very important role in the restitution of data dispatched in several tables, as well as the derivation of new data. Knowing that these attributes have no presence during the entity-association diagram, which presents the basis of the relational model, and they would appear to replace or to represent the relations that have been suppressed at the physical level, presenting one of the pillars of the semantic aspect of relational systems alongside its structural role. So its transformation leads to one of the solutions: find a new transformation to the relationships between the tables and get rid of the foreign keys or find a new transformation of the foreign keys with their built-in mechanisms. In our approach, we have proposed a transformation that is based on the second choice knowing that it is close to relational data containers and promotes all the advantages offered by foreign keys, which involves the transformation of consistency from relational systems to MongoDB. It should be noted that researchers transform these attributes as being ordinary fields, and neglect their functions automatically integrated into relational systems.

4.3.5. Transform associations

During this section, we deal with the most critical part, which transforms the relational aspect in relational databases towards a non-relational system, keeping the possibility of generating other new information not stored in the database and avoiding joins as much as possible. During this part, we will see the transformation of associations between tables in a way adapted to the object-oriented spirit used by JSON in MongoDB, without losing the powers of relational systems. To better understand our contribution in this part, we invite you to discover the different rules for transforming the relationships between the tables, which we have classified according to 6 types presented in the following graph: Figure 23 helps to classify the different relationships between tables into 6 different types, each of which requires a specific rule to transform it to fit the spirit of MongoDB. Watching the complexity of the transformation of foreign keys, as well as the complexity of its

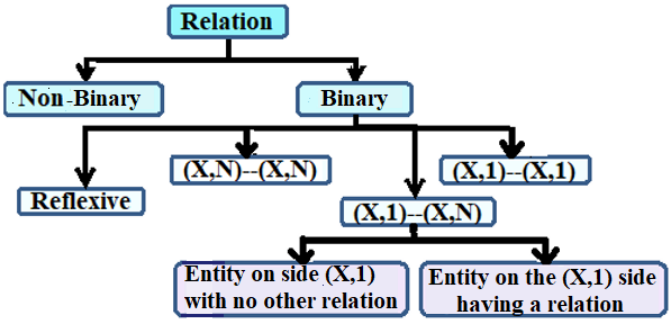

Figure 23: The different relations of the relational model

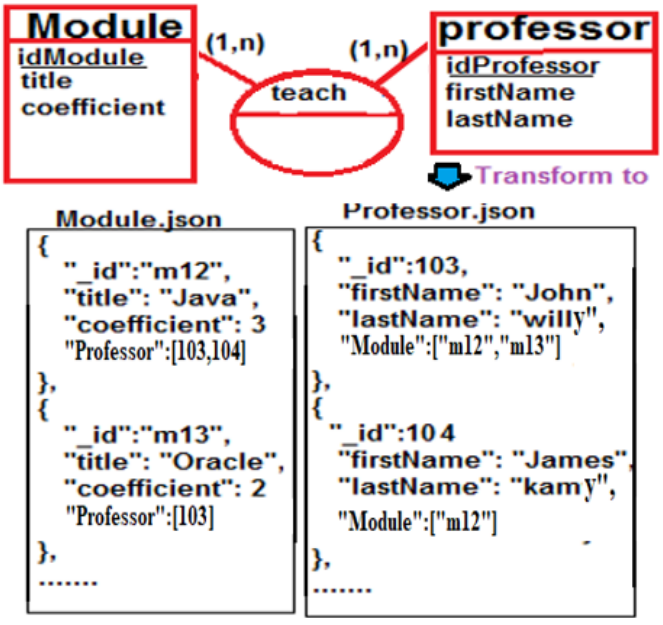

Figure 24: Transformation of many-to-many binary relationships in the TMSDRDND approach

exploitation by MongoDB, then we decided to try as much as possible to avoid or reduce foreign keys, by reducing the related collections. For this purpose, we propose the following rules to transform the different types of associations between tables, into forms adapted to the functioning of MongoDB and close to the manipulation of objects and their exploitation.

Transformation rule 1: for binary relations of the type (many-many): this type of association is considered complex, and we propose for it as a transformation rule, the creation of a list of identifiers of the documents with the left collection in each related document from the right collection and another list of identifiers of the related documents from the right collection in each related document from the left collection. The association will disappear and in return, the two lists appear in each document with the disappeared association. This transformation is explained by an example presented in the figure 24: Figure 24 illustrates how to transform a relationship of type (many-many) according to rule 1 presented above, by presenting an example, which includes two entities linked together via a relationship

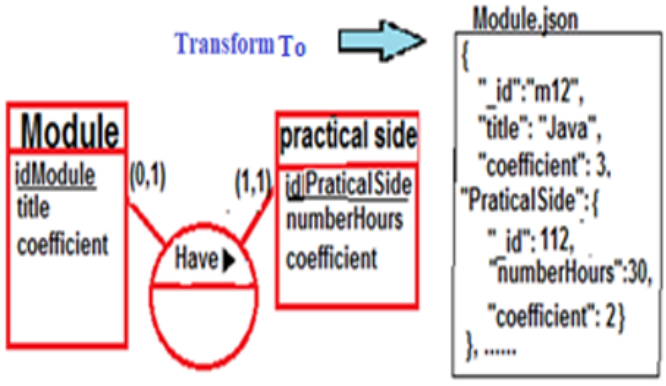

Figure 25: Transformation of one-to-one binary relationships in the TMSDRDND approach

of this type. This transformation will generate two collections whose documents each include a list of identifiers of related documents on the other side to simplify searches and minimize the need for joins between these two collections. This way is close in mind to dealing with objects on the one hand, and on the other hand, it will greatly reduce the need to join the two Collections, and it conforms to the join operation proposed in our approach as a translation of relational join. These structures must be defined in Mongoose.

Transformation rule 2: for binary relations of type (one-one): We propose for this association to be removed and translated to an internal document into a larger document, like the spirit of the internal objects. The figure 25 shows an example to better understand this transformation: Figure 25 illustrates how to transform a relation of type (one-one) according to rule 2 presented above, by presenting an example, which includes two entities linked together via a relation of this type. This transformation will generate a single collection whose documents each include another linked document on the other side to simplify searches and minimize the need for joins between these two collections. The internal document may be absent to represent the minimum cardinality 0 seen in the example in Figure 25. In this case, and using the example in the figure 25 to clarify our idea, the Module document on the (0,1) side will be the largest element that will contain one more attribute, whose name is exactly the name of the other participating entity in the relation, ”PraticaleSide” in our case, and its value is indeed the document which represents a row in the table ”PraticaleSide” which relates to its corresponding row in the table ’ ’Module”. The minimum cardinality 0 will be translated in the Module document’s schema in Mongoose by the [required: false] attribute, and its meaning is that there are modules without the Practice Side. In the exploitation phase of this model, we resort to using the notation adopted by the objects to access the internal objects without the need to do any joint operation. This transformation eliminates the joins resulting from an association of this type, knowing well that the main objective of this type of association is the verification of the existence of a correspondent of each document. This significantly reduces the total number of joins in this new system.

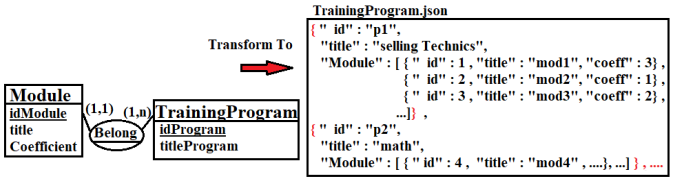

Figure 26: Transformation 1 of one-to-many binary relations in the TMSDRDND approach

Transformation rule 3: for binary relations of type (one-many), whose entity on the side of the cardinality (one) doesn't participate in another relation: Also for this association, it is proposed that it be deleted and translated by a list of internal documents in a larger document. The figure 26 shows an example to better understand this transformation: Figure 26 illustrates how to transform a relation of type (one-many) according to rule 3 presented above, by presenting an example, which includes two entities linked together via a relation of this type. This transformation will generate a single collection whose documents each include a list of related documents on the other side to simplify searches and minimize the need for joins between these two collections. In this case, and using the example in the figure 26 to clarify our idea, the TrainingProgram document on the (1, N) side will be the largest element that will contain one more attribute, whose name is exactly the name of the other participating entity in the relation, "Module" in our case, and its value is indeed a list of the Modules documents which relate to the only TrainingProgram they contain. We can have the cardinality (0, N) instead of (1, N), and in this case, the minimum cardinality 0 will be translated in the schema of the TrainingProgram documents in Mongoose by the attribute required: false, and its meaning is that there is TrainingProgram which does not hold any module. This structure brings the notion of internal or embedded objects closer to other larger objects as in the case of rule 2, which allows eliminating joins operation by this transformation, creating a default join between external and internal documents of our collections.

Transformation rule 4: for binary relations of type (one-many), whose entity on the side of the cardinality (one) participates in another relation: this association is considered a complex relation, of which we propose for it to be transformed according to a rule similar to the second transformation rule presented above. The figure 27 shows an explanatory example of this rule: Figure 27 illustrates how to transform a relationship of type (one-many) according to rule 4 presented above which treats the type of relationship with an option of participation of the entity on the one side in another relationship, by presenting an example, which comprises two entities linked together via a relationship of this type. This transformation will generate two collections: the first is identical to the situation in figure 26 and the second is formed by documents of the entity on the one side to simplify searches and minimize the need for joins between these two collections and also to offer the possibility of participating in other calculations depending on the relationship with the entity on the one side. **Transformation rule 5**: for reflexive binary

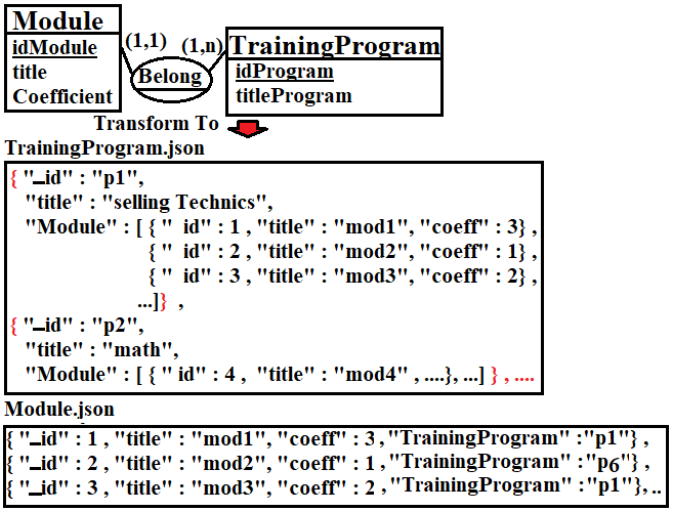

Figure 27: Transformation 2 of one-much binary relations in the TMSDRDND approach

relations: This type of relationship is designed in the logic of relational databases to be the subject of successive joins and as many times using the same table, also by building an object hierarchy according to the direction of the relationship. This type is generally approximated by ordinary binary relations, but it is a very special type in its logic, modeling, and processing. Theoretically, this kind of relationship favors an infinite number of joins of such a table with itself using aliases. In this relationship and especially at the moment of joining, we will create two virtuous tables' copies from the same table, so that the one placed on the left plays a different role than the one that will be placed on the other side, even if they are from the same table. But in practice, we make a single join or double join of such a table with itself. Consequently, we will transform this binary association according to the rules defined before transforming the binary associations by considering their cardinalities.

Transformation rule 6: for non-binary relationships: for this kind of complex association, it is proposed to convert it into an independent file, by adding to the documents in the collection of this file attributes of the association if they exist and other additional attributes, each of which represents a participating table in the association as a foreign key. The structure of these documents must be defined using Mongoose. The figure 28 shows an example to better understand this transformation: Figure 28 illustrates how to transform a non-binary relationship according to rule 6 presented above, by presenting an example, which includes three entities linked together by this type of relationship. This transformation will generate four collections, each of which groups the documents from one entity among the three presented in the example, except for the fourth collection, which models the relationship between the documents of the other collections using foreign keys, which each refers to a participating document in this relation. It is the only transformation identical to that used in the relational system, and luckily this kind of relation represents almost 5 percent of the relations forming a relational database. In the relational model,

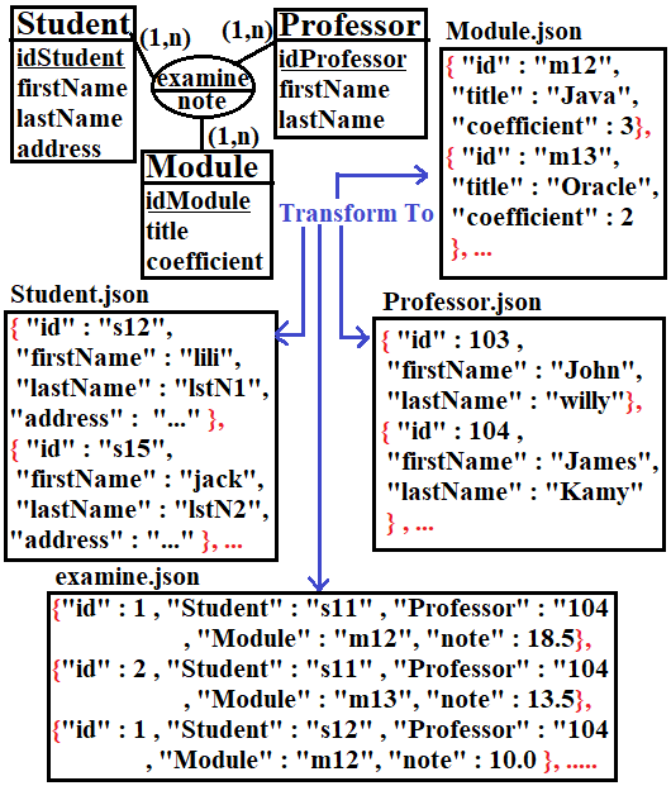

Figure 28: Transformation of non-binary relations in the TMS-DRDND approach

```
db.collection.createIndex(< champ_et_type >, < options >)
```

Figure 29: Defining indexes in MongoDB

these relations disappear and are replaced by foreign keys to be restored thanks to the join operation. To transform the relations, we must rely on the relations themselves to find a new form without foreign keys and joins or we must base ourselves on the transformation of the foreign keys and the join operation. Given the difficulty of merging all tables into a data container without going through foreign keys and given the importance of the join operation in deriving new unstored data, so we leaned towards transforming foreign keys and the join operation, as well as the relationship transformations as shown above. This way guarantees the power of derivation of the new data and benefits from the resemblance between the tables to the relational systems with the Documents in MongoDB to validate this transformation.

4.3.6. Transform indexes

When a collection is created, MongoDB automatically generates an index on the -id field. This index cannot be deleted, because it guarantees the uniqueness of this identifier. To create an index yourself, you must use the create index function, the syntax of which is as follows: Figure 29 shows the syntax for creating an index yourself using the createIndex function. This function accepts two parameters: the first represents the

```
db.personnes.createIndex({"prenom": 1}, {"unique": true})
```

Figure 30: Transformation of simple indexes using the TMS-DRDND approach

```
db.personnes.createIndex({"age": 1, "nom": 1}, {"name": "idx_nom_age"})
```

Figure 31: Transformation of complex indexes using the TMS-DRDND approach

attribute that makes the base of the index and the second allows to express of the list of index options such as unique, name, partialFilterExpression, sparse, expireAfterSeconds, hidden, and storageEngine. The options parameter can be formed by one or more options among these options presented [19]. **Simple indexes**

Unique indexes guarantee that a given value will appear at most once in the index. We have seen previously that such an index was systematically placed on the id field of documents in a collection and that it was also impossible to delete it! In the current state of our people collection, we couldn't put that kind of index on the name field because most of the values in this field have multiple occurrences. On the other hand, we could create a unique index on the firstname field, if each one of their values is unique: Figure 30 illustrates how to define a simple index based on a single attribute, through an example that shows that an index is created for the attribute called "prenom". This kind of index requires a unique option to guarantee the functioning of the index which represents a means to speed up the search in this collection. From now on, any attempt to insert a person with a first name already present in one of the documents in our collection will be refused. Putting a unique index on a field that has a very high probability of containing duplicates is a pretty bad decision. Moreover, now that our first name field is subject to a uniqueness constraint, we will not be able to insert more than one document that does not contain this field.

Complex indexes

An index can cover more than one field: this is called a compound index. In this type of index, the order in which the fields are listed is important. Let's create a composite index named idxNomAage which will focus on the name and then on the age of the people: Figure 31 shows an example to clarify how to create a complex index. The index created in this example will be ordered documents in collections by increasing age values, then within each of the different age values, this index will be ordered alphabetically by name. When a compound index whose prefix is not a character string, an array or a subdocument is used with a collation. A query that doesn't use the correct collation for the indexed text field can, however, rely on the index prefix. This element is not neglected by our integral approach since it forms part of the data of relational databases so as not to have the loss of data during migration between the two systems: relational and MongoDB.

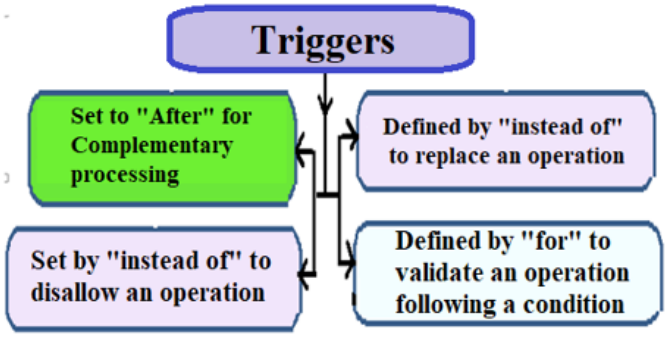

Figure 32: The different types of triggers (Triggers)

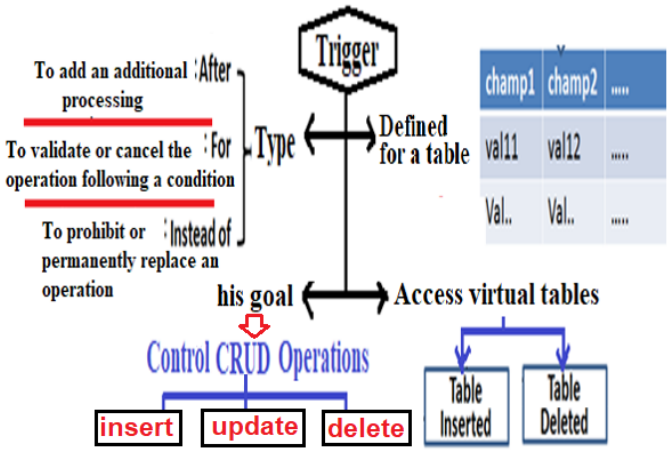

Figure 33: The Modelization of the Functioning of the Triggers

4.4. The transformation rules of the Semantics pane

4.4.1. Transform Triggers

Triggers are programs whose definitions contain two parts: the second defines the processing and the service provided by this program and the first presents the terms of its call, which is automatically triggered following an insertion, modification or delete a row from a specific table; when the trigger is called by the ”Delete” operation, it therefore keeps the history of deleted rows in a virtual table named ”deleted”, and if its call is defined by the ”insert” operation, then it keeps the history of new rows inserted in another virtual table named ”inserted”, also if its call is triggered by an ”update” operation then this operation considers that the old values of these rows are considered deleted and stored in the ”deleted” virtual table, at the same time, it considers that the new version of these rows are considered as newly inserted rows, whose values will be stored in the ”inserted” virtual table of the new version of the rows of the table will be readable by the trigger Also the triggers can use any physical table of the database in their treatment. The main purpose of triggers is to control the movement of data for a given table, embodied by insert, delete, or update operations. In relational databases, these operations interact on the rows, but in the case of MongoDB, we reason by the documents. On the other hand, you can transform a row of data from a table in the relational database into a document under MongoDB. This can mean that the check that the trigger performs on a row of data in the relational system will be translated in MongoDB by a control of the document corresponding to this row of data. In the figure 32, we present the four types of triggers: All of these triggers presented in figure 32 have one of these objectives: replace one or more operations with another processing or prohibit this or these operations outright, so the trigger can be defined to add additional processing of a CRUD operation to the table of the trigger or others, and finally, we can have a trigger to validate or cancel the CRUD operation triggering the trigger following a condition to be respected. To better see the interest of triggers, we propose the example of an air travel management database and its reservations: theoretically in this example and without the existence of triggers, the ”Reserver” table accepts all the operations of insertion of a new line, this can generate several reservations in a flight that far exceeds the capacity of the aircraft used in this flight, creating the mismanagement of customers and reservations. In this example, the other rules and constraints can only control the data according to data columns, but the management of flight reservation lines is only possible through the mechanism of triggers which validate the reservation operations as long as they do not exceed the capacity of the aircraft used in the flight concerned. The participants in a trigger can be summarized by the figure 33: Figure 33 summarizes the axes of action of the triggers, which are defined to serve one single table, to ensure control of the operations for updating the data stored in this table. Also, this figure shows that it is a type of action of the trigger in the case of the execution of a monitored update operation, and the power to do actions on the virtual tables of its table to develop its treatment. To transform each trigger from the relational system to the MongoDB system, we will create specific functions in Node.js, and place them in Mongoose in the schema that defines the structure of the documents concerned by this trigger. These schemas define the structure of all documents in the collections stored in the JSON file. Our functions were developed by NodeJs, which is considered their mother tongue to use JSON files. To complete the principles of triggers, we need to bind the concerned operation, which triggers the trigger, with its function developed through the validation attribute in the schema hosted in Mongoose.

- To do this, we must identify the table monitored by the trigger.
- The CRUD operation(s) that trigger execution of the trigger.
- The nature of the trigger processing (complementary, validation, or replacement).
- The condition to validate or invalidate the CRUD operation in the second case is the nature of the trigger.

During the transformation phase, we will develop the transformation of the table into a schema of the corresponding documents, by adding a part that adds the redefinition of the CRUD operation that triggers the trigger, to add the method that performs trigger processing in the same way as foreign keys. Relational databases without triggers can lead to disasters. For

```
db.createView(< nom >,< source >,< pipeline >,< collation >)
```

Figure 34: Views in MongoDB

OR

```
db.runCommand( { create: < nom >, viewOn: < source >,
pipeline: < pipeline >, collation: < collation> } )
```

Figure 35: Views in MongoDB – second version

example, the flight reservation table, which uses aircraft with a specific capacity, can allow new rows to be inserted until the capacity of the aircraft is exceeded, unless there is a trigger controlling this insertion so that it does not exceed the capacity of the aircraft. In our case, we consider any transformation of databases without trigger as a reform of a handicapped database. Old migration works neglected this axis when processing it, but we tried to find an adequate transformation to the triggers. This step opens the door to future research to validate or improve this transformation.

4.4.2. Transform Views

Views in relational databases are queries for the selection or extraction of data stored permanently on the hard disk, the result of which is calculated instantly following a call, and according to a simple calculation formula or complex. The result of a view is visualized as a table and also used as a table to use in join operations or as a data source of another extraction query or to interact on their rows either for l addition, deletion, or modification. When creating a view, you must specify the formula of its query to save. This query represents a calculation formula, an extraction that aims to select or derive other information not stored in the database. This technique aims to subdivide a problem to simplify its solution or save a calculation formula that must be re-executed several times. Our approach proposes to transform this technique into MongoDB by one of the following two methods: Figures 34 and 35 show two different examples to present two different ways to create a view in MongoDB. The first one uses the createView function and the second uses the runCommand function which represents a command ready to be executed at any time, and both functions need four parameters to specify the name of the view, the data source of the view, and the treatment that forms the basis of the view. Views operate equally on collections or other views and necessarily reside in the same database. They are sort of read-only collections and when they relate to collections, they can use the same indexes as these. It is however impossible to modify the indexes of a collection from a view based on this one, also it is impossible to rename a view (it will have to be destroyed with a drop operation, exactly as we did for collections that were used for our many examples, then recreated it). Of these four parameters, collation is the only one that is optional. The view name will be the first of the required parameters, followed by the source which designates the target view or collection, and the pipeline

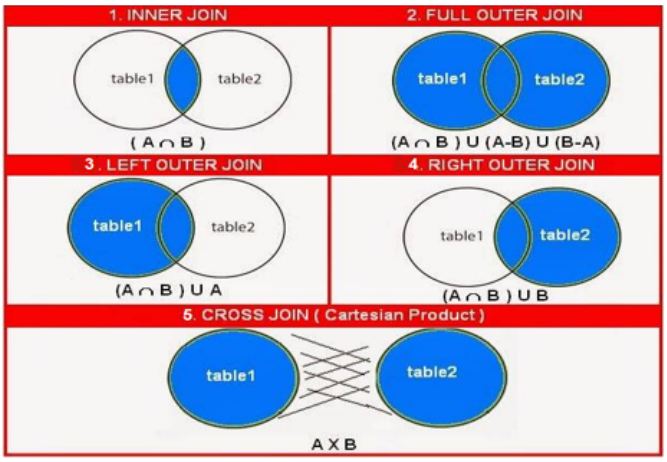

Figure 36: Joins in the relational model [1]

which is an array containing your aggregation pipeline. If you prefer to use commands, you will prefer the last syntax above: The views are part of the components of relational databases but unfortunately, it has been neglected in old migration work. In our approach, this element has found a new transformation. Its concept can be used in improving queries under MongoDB.

4.4.3. Transform the join operation

The power of relational databases has presented by the joins between the tables because, with their help of them, we can reconstruct the global information contained in all the tables. This global information holds information on the objects in action and their actions between them with visibility on the multiplicity of each action. Also with the help of joins, we can deduce other information, describing the flight of operations, effects, and actions contained in our database, revealing the semantic side of the data stored and not stored explicitly in the tables. The joins show this information and represent a vital operation that holds the spirit of the relational model. Without a join, no one has the courage in the relational model to talk about the normalization of tables, the separation of data by the entity, the dependence between fields of the same entity, or the relationship between tables. Joins are operations that tell developers ”go ahead and do whatever you want, I’m able to reconstruct data distributed across tables, provided they have a common field, and they’re chosen carefully ”. Therefore, the migration and transformation of the join from the relational system to the MongoDB system is an essential operation. In the relational system, there are several types of joins, which are presented in the following graph: Figure 36 shows the different types of join: inner join, right join, left join, cross join, full join, and outer join. These types of joins are used to extract the global information dispatched in several data containers through the inner join, on the other hand, the other types present non-normal join options to serve essentially the derivation of new data which requires a gymnastic join. To be able to transform the joins from the relational system to the MongoDB system, it must be understood that the join involves two stages: the Cartesian product between two tables and the restriction according to a condition,

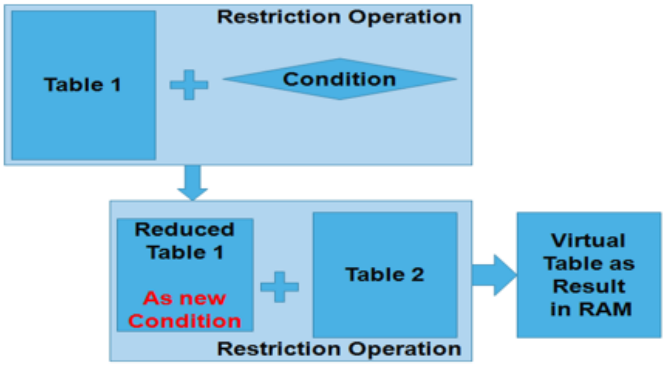

Figure 37: Join operation in the TMSDRDND approach

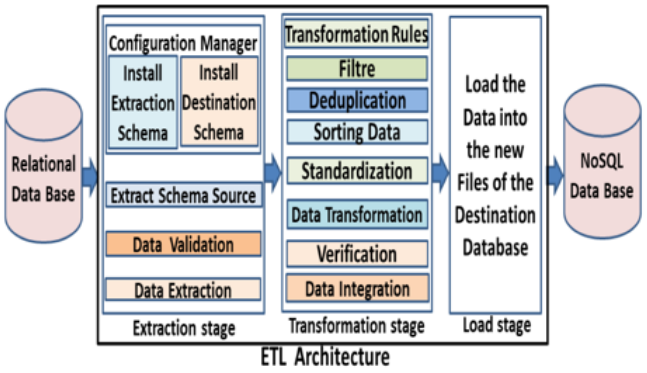

Figure 38: Modeling of the migration ETL adopted by the TMSDRDND Approach

which then determines the type of join. However, the Cartesian product between the JSON files is not technically possible, and if logically possible, will not be the right solution, especially with giant files. Indeed, suppose that each file contains a million objects, then its product will contain 1000 Milliards objects, to be transmitted to apply a restriction operation on all these objects to have as a result, almost half of the square root of this number of rows, which is not possible and is not optimal. In this case, we must think of another solution that offers the same result as joins but in a different way. The join operation forms the primary axis of relational systems to extract and exploit the relationships between tables, but since it uses the Cartesian product between tables, it will be the wrong solution for BigData and BigTable because it exponentially increases the volume of data to be processed. In our contribution, we suggest transforming this operation by a JavaScript program, whose objective is to minimize the collections to be joined using the restriction condition, recursively with the construction of the new virtual collection until 'on reaching the minimal virtual collection that represents the solution of the join. The figure 37 approximates the principle of this transformation: Figure 37 illustrates our way to transform the join operation which is based on a condition of restriction to apply at the first time for one collection, which will subsequently be the restriction condition for the other collection, and so on until all collections are completed to produce the final result. This restriction will be applied to Documents by considering the notation of objects offered by the spirit of JSON. It is a crucial operation for relational systems and the pivotal element for exploiting relations for data derivation, but its mechanism is poorly designed in relational systems and appears as its weakness in handling BigData. MongoDB offers an alternative way of doing this that relies on pipelined queries. Our approach proposes another form that is based on iterative restrictions, and each represents the restriction condition of the other until the last table by forming small data containers so that they are merged to have a single result of joining. This axis can be a way of research either to validate this way or to improve this concept.

4.5. Data migration using an ETL

Data migration is the final step in our approach. This step comes after building the schemas and models of our converted database. Also, this phase concludes the work and at the same time forms a practical test of our approach. This step uses all the transformations carried out before in a well-defined order to create the data container as well as its new functionalities which will add up with the functionalities and powers of the destination system (MongoDB), such as speed, availability, and scalability. To better explain this step, we will propose an architecture for our ETL which defines in detail the extraction phase, then the transformation of the data according to the rules defined in the previous steps, and finally the loading of the processed and transformed data into the final files to import them into MongoDB as the final version of the database. This final version of the database must comply with the specificities of the NoSQL system and precisely MongoDB also must have all the data stored in the source database hosted in the relational system, with the powers of the relational system such as the generation of new data, the restitution of the global information dispatched in several documents and the control of the validity of the entered data.

4.5.1. The overall architecture of our ETL

To use ETL, we need to extract data from a relational system according to its structural source to initiate the process of transforming it according to its structural target. The final transformation can be accompanied by loading the data into the target system according to the principle of multi-threaded parallelism. In Figure 38 we show the ETL architecture used for this data migration. Figure 38 presents the architecture of our ETL by defining its three phases: extraction, transformation, and loading of data. This architecture also presents the different stages of each phase. According to this architecture, it is clear that the extraction is directed by the schema of MongoDB as a destination system to know exactly the information we must extract to transform it to their corresponding according to the specificities of MongoDB, and at the same time this step is driven by the source system schema to know the location of the information to be processed. Also according to this architecture, our system performs a set of verification and validation tests to ensure the database migration result.

4.5.2. The extraction phase

Data extraction starts at:

- Instantiation of the extractor, defining its configuration as the extraction engine, by importing two schemas: the schema of the extraction model, starting with the precision of the source data server, then establishing a connection with it, and then preparing the required at the beginning of this phase: Objects, agents, and interfaces, and finally install the target system schema produced by the structural and semantic component transformation phase to develop additional elements of various diagrams and files of the target system.
- After import, the system loads the diagrams and models of the individual components of the source, as well as the prototypes and references of the components, and begins the verification phase
- We then validate models, data structures, and processing to avoid or minimize errors during the translation
- Finally, we start extracting the data and after validating the model, load them into our objects precisely to put all the data in the right place so as not to lose data and transform all the data successfully.

4.5.3. The transformation phase

During this transformation phase, ETL starts with the following steps:

- Import and load transformation rules and filters created by the first layer of the method.
- The ETL then performs the extraction of the data, copies the extracted data, and puts each piece of information into the corresponding object, which is the beginning of its transformation process.
- Afterward, we need to order these objects and these processes to facilitate the logical order to succeed in the transformation phase.
- This step is followed by a normalization step to avoid semantic errors, especially since the JavaScript language is case-sensitive.
- After that, it's time to start a general transformation.
- This step is followed by validation and integration of the final data in preparation for data loading.

4.5.4. The loading phase

The final step can be to load the data into the target system by importing the data in its final state. This step sets up the latest version of the database in MongoDB, has it up and running, and looks at the various constraints, rules, triggers, and other components that make the database very rich and powerful.

5. Conclusion

Currently, NoSQL databases present powerful technologies to manage BigData, which are implemented in environments handling big masses of data like Google, Yahoo, Twitter, Facebook, and search engines, because they need a power giant in the storage and processing of these huge volumes of data with a large number of users and simultaneous requests. Faced with the limits presented by the old relational systems in the management of BigData, and considering the data accumulated by this old system during decades of exercise in the market, which describes the feedback from the past, the situation of the presence of these institutions, and their competitors, also the future opportunities, then it is legitimate to think about migrating this valuable information from relational systems to a NoSQL system. With this in mind, several approaches and studies are developed by researchers to satisfy this need, but unfortunately, they did not achieve the objective because of the weaknesses in their treatments or because of the lack of their approaches, materialized by the absence of some main components of these databases during its analysis and processing. During this article, we have chosen MongoDB as the destination system for our migration approach, because it brings the philosophy of relational systems closer and offers the possibility of transforming the majority of its components. Also during this work, we proposed our new methodology for the analysis and modeling of the two systems: source and destination of the migration, by presenting the different stages of our methodology. At the end of this step, we managed to present the models, the meta-models of the two systems, the correspondence between their components that we qualify as necessary to start the migration, as well as the components that require a specific translation. This work guided us towards the implementation of the overall architecture of our approach to migration, which we called "TMSDRDND". This completed part defines the framework and the perimeter of our approach and also represents the guide that clarifies our way of implementing this approach and gives it the integral dimension. This architecture shows that our approach is formed by two layers: the first transforms the data structures and data of the semantics of relational systems, while the second layer uses an ETL that we have proposed its architecture and their mechanism, to physically migrate the data from the source system to the destination system respecting the result of the first layer. We decided to make two layers because they are separate and each requires an admin launch. During the first layer, called "TSRSNLayer", we process the two types of data: structural and semantic in two stages which can be executed in parallel to produce two files: the first presents the schemas which define the structure minimum of the migration to the destination system, and the second encompasses programs that approximate the semantics of relational systems in MongoDB. The result of this layer will be the pivot element to start the processing of the second layer called "MDRSNLayer". Based on the first analytical part and the overall architecture of our approach, we proceeded to develop three other parts, each of which presents a set of rules and the way to transform each component that is part of it. During

the development of these three parts, we succeeded in dealing with all the components of the parts: structural, semantic, and the transfer of data via an ETL whose architecture and mechanism has presented below. Knowing that there are components that require the presentation of its mechanism implemented implicitly and automatically in the relational system and not considered by MongoDB, to be able to propose the right transformation such as foreign keys, primary keys, the join operation, triggers, and relationships between tables. This allowed us to migrate a relational database completely with its advantages to MongoDB, which we can reconcile by consistency at all times, a clear data structure that defines the minimum data to be stored for all documents, the data framing through validation conditions, vertical and horizontal document validation, data derivation and extraction power, and storage optimization. In a summary, this approach guarantees the transformation of the advantages of the relational system to MongoDB by keeping the advantages of MongoDB as a manager of BigData, by proposing a new model, which can remove the need for relational databases. This work presents a huge gain in the research community, an axis of scientific investment, and a real revolution in the world of databases. In future work, we present an implementation for the different parts of the layers: "TSRSNLayer" and "MDRSNLayer", to pass to a heuristic definition, which will be used in the elaboration of the agents, which concretizes our intelligent approach.

References

[1] *The digital universe of opportunities*, Available: https://www.emc.com/leadership/digitaluniverse/2014iview/executive-summary.htm.

[2] *Data is eating the world*, Available: https://whatsthebigdata.com/2017/04/18/idc-163-trillion-gigabytesof-data-will-be-created-in-2025.

[3] O. J. Ibidoja, F. P. Shan, J. Sulaiman & M. K. M. Ali, "Robust M-estimators and Machine Learning Algorithms for Improving the Predictive Accuracy of Seaweed Contaminated Big Data", Journal of the Nigerian Society of Physical Sciences **5** (2023) 1137.

[4] M. A. Okono, E. P. Agbo, B. J. Ekah, U. J. Ekah, E. B. Ettah & C. O. Edet, "Statistical analysis and distribution of global solar radiation and temperature over southern Nigeria", Journal of the Nigerian Society of Physical Sciences **4** (2022) 588.

[5] M. V. Sokolova, F. J. Gómez & L. N. Borisoglebskaya,"Migration from an SQL to a hybrid SQL/NoSQL data model", Journal of Management Analytics **7** (2020) 1.

[6] V. F. de Oliveira, M. A. de Oliveira Pessoa, F. Junqueira & P. E. Miyagi, "SQL and NoSQL Databases in the Context of Industry 4.0", Machines **10.1** (2022) 20.

[7] I. Mearaj, P. Maheshwari & M. J. Kaur, "Data conversion from the traditional relational database to MongoDB using XAMPP and NoSQL", In 2018 Fifth HCT Information Technology Trends (ITT) (2018) 94.

[8] M. G. Gopalan, C. Prasanna, Y. S. Krishna, B. Shanthini & A. Arulkumar, "MYSQL to cassandra conversion engine.", In 2017 Third International Conference on Sensing, Signal Processing and Security (ICSSS) (2017) 503.

[9] Y. S. Wijaya & A. A. Arman, "A framework for data migration between different datastore of NoSQL database", In 2018 International Conference on ICT for Smart Society (ICISS) (2018) 1.

[10] J. Kachaoui & A. Belangour, "MQL2SQL: A proposal data transformation algorithm from mongoDB to RDBMS.", International Journal of Advanced Trends in Computer Science and Engineering in progress (2020).

[11] G. B. Solanke & K. Rajeswari, "Migration of relational database to MongoDB and Data Analytics using Naïve Bayes classifier based on Mapreduce approach", In 2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA) (2017) 1.

[12] K. Černjeka, D. Jakšić & V. Jovanovic, "NoSQL document store translation to data vault based EDW", In 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO) (2018) 1197.

[13] B. Namdeo & U. Suman, "Schema design advisor model for RDBMS to NoSQL database migration", International Journal of Information Technology **13** (2021) 277.

[14] S. Ghule & R. Vadali, "Transformation of SQL system to NoSQL system and performing data analytics using SVM", In 2017 International Conference on Trends in Electronics and Informatics (ICEI) (2017) (883).

[15] F. Yassine & M. A. Awad, "Migrating from SQL to NOSQL Database: Practices and Analysis", In 2018 International Conference on Innovations in Information Technology (IIT) (2018) 58.

[16] K. Shiromoto, T. Hochin & H. Nomiya, "Integrated usage between relational DBs and NoSQL DB", In 2019 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD) (2019) 244.

[17] B. Jose & S. Abraham, "Exploring the merits of nosql: A study based on mongodb", In 2017 International Conference on Networks and Advances in Computational Technologies (NetACT) (2017) (266).

[18] D. Sevilla Ruiz, S. F. Morales & J. García Molina,"Inferring versioned schemas from NoSQL databases and its applications", In Conceptual Modeling: 34th International Conference, ER 2015, Stockholm, Sweden, Proceedings **34** (2015) 467.

[19] *MongoDB Manual*, Available: https://www.mongodb.com/docs/manual/reference/method/db.collection.createIndex.