LEANDRO DUARTE PULGATTI

DATA MIGRATION BETWEEN DIFFERENT DATA MODELS OF NOSQL DATABASES

Dissertation presented as partial requirement for the Master degree . Graduate Program in Informatics, Sector of Exact Sciences, Universidade Federal do Paraná.
Supervisor: Marcos Didonet Del Fabro

CURITIBA

2017

Resumo

Desde sua origem, as bases de dados Nosql têm alcançado um uso generalizado. Devido à falta de padrões de desenvolvimento nesta nova tecnologia emergem grandes desafios. Existem modelos de dados , linguagens de acesso e frameworks heterogêneos, o que torna a migração de dados ainda mais complexa. A maior parte das soluções disponíveis hoje se concentra em fornecer uma representação abstrata e genérica para todos os modelos de dados. Essas soluções se concentram em adaptadores para acessar homogeneamente os dados, mas não para implementar especificamente transformações entre eles. Essas abordagens muitas vezes precisam de um framework para acessar os dados, o que pode impedir de usá-los em alguns cenários. Entre estes desafios, a migração de dados entre as várias soluções revelou-se particularmente difícil. Esta dissertação propõe a criação de um metamodelo e uma série de regras capazes de auxiliar na tarefa de migração de dados. Os dados podem ser convertidos para vários formatos desejados através de um estado intermediário. Para validar a solução foram realizados vários testes com diversos sistemas e utilizando dados reais disponíveis.

Palavras Chave: NoSql Databases. Metamodelo. Migração de Dados.

Abstract

Since its origin the NoSql Database have achieved widespread use. Due to the lack of standards for development in this new technology great challenges emerges. Among these challenges, the data migration between the various solutions has proved particularly difficult. There are heterogeneous datamodels, access languages and frameworks available, which makes data migration even more complex. Most part of the solutions available today focus on providing an abstract and generic representation for all data models. These solutions focus in design adapters to homogeneously access the data, but not to specifically implement transformations between them. These approaches often need a framework to access the data, which may prevent from using them in some scenarios. This dissertation proposes the creation of a metamodel and a series of rules capable of assisting in the data migration task. The data can be converted to various desired formats through an intermediate state. To validate the solution several tests were performed with different systems and using real data available.

Key-words: NoSql Databases. Metamodel. Data Migration.

P981d Pulgatti, Leandro Duarte
Data migration between different data models of NOSQL Databases / Leandro Duarte Pulgatti. – Curitiba, 2017.
79 f. : il. color. ; 30 cm.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2017.

Orientador: Marcos Didonet Del Fabro.

1. NoSql Databases. 2. Metamodelo. 3. Migração de dados. I. Universidade Federal do Paraná. II. Didonet Del Fabro, Marcos. III. Título.

CDD: 005.13

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
Setor CIÊNCIAS EXATAS
Programa de Pós-Graduação INFORMÁTICA

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **LEANDRO DUARTE PULGATTI** intitulada: **Data Migration Between Different Data Models of NoSql Databases**, após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua aprovação.

Curitiba, 17 de Fevereiro de 2017.

MARCOS DIDONET DEL FABRO
Presidente da Banca Examinadora (UFPR)

JORGE AUGUSTO MEIRA
Avaliador Externo (SNT)

EDUARDO CUNHA DE ALMEIDA
Avaliador Interno (UFPR)

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

Agradecimentos

Gostaria de agradecer a todos os que de algum modo contribuíram para a realização deste trabalho de mestrado de uma forma direta ou indireta.

Mas alguns agradecimentos são especias: Aos meus pais Odinês Duarte e Leonel José Pulgatti, pela vida e pela confiança que sempre tiveram em mim.

A Andreia Zanella, que se provou companheira nas horas difíceis e que muito me apoiou quando eu precisava e aconselhou quando pedi.

A minha filha, Giulia Simão Pulgatti, por ela tento sempre ser alguém melhor todos os dias.

Ao meu orientador Marcos Didonet, pois sem seu conhecimento e sem sua incrível paciência comigo e meus atrasos nunca teria sido possível terminar este trabalho.

Ao Luciano dos Santos Fier, que tenho como um irmão, e só de conversar já me alegra o dia e limpa os pensamentos, sem contar as correções e dicas que tanto me auxiliaram.

Ao Paulo Ricardo Lisboa de Almeida, que me apoiou em várias áreas e estava sempre disposto a auxiliar quando precisei.

A todos os meus amigos e colegas do Dinf que, de algum modo, me ajudaram a concluir a realização deste trabalho.

A todos vocês o meu mais sincero obrigado...

"Pensar com competência é tão esforçado quanto ter músculos...
exige empenho, exige dedicação...
porque não tem por que não entender, o resto é só preguiça e covardia."

Clóvis de Barros Filho

List of Figures

List of Tables

Listings

List of abbreviations and acronyms

ACID Atomicity, Consistency, Isolation, Durability

API Application Programming Interface

BASE Basically Available, Soft state, Eventually consistent

BSON Binary JSON

CAP Consistency, Availability, and Partition tolerance

CRUD create, retrieve, update, delete

DBMS Database Management System

GUI Graphical User Interface

JSON Java Script Object Notation

MDE Model-driven engineering

NoSQL Not Only SQL

RDBMS Relational Database Management System

REST Representational State Transfer

SQL Structured Query Language

XML Extensible Markup Language

YAML YAML Ain't Markup Language

Contents

1 Introduction

Changes are constant in a Database environment during a project life cycle. The rapid response to these changes is priority for the developers [3]. The rigidness imposed by RDBMS (Relational Database Management System) in which the schemas[1] are mandatory tends to hinder the changes [4]. In this scenario, the absence of schema in NoSql Databases is pointed as an advantage by the developers. This benefit comes when a rapid response to the modifications, to meet the changes in the system requirements, is necessary [5].

During the design process in the RDBMS, much effort is put in the data project. The optimal administration of this effort is only possible after the requirements are sufficiently stable. In her work [6] states that the designer has many aspects to think during the project phase. However, it is far less bureaucratic to carry out changes in schemaless databases, such NoSql Databases [4]. In many cases, different versions can coexist within the same entities, with the application being responsible for dealing with these scenarios [7]. This advantage, nevertheless, comes with a price. As the development rarely have the luxury of waiting for the requirements to stabilize, the development will lead to data migration. In this context, particular attention must be paid to the data migration needs

Nowadays, changes in the scenario, notably requirements, are challenging. Quick decisions and the corresponding lack of standardization noted in [5] leads to data migration. The greater the change in scope the greater the need to migrate to another DBMS with different characteristics. After the system is up and running the changes on requirements can lead to a data migration too. For example the vendor lock-in problem in the Cloud databases that are dealt in [8]. The significant differences among the features of different NoSql Database mean that there is no single system which would be the most suitable for every need [9].

Dealing with the points mentioned above and with other complex architectural decisions is challenging, especially in the beginning. It is near to impossible to hit an accurate first decision without an extensive knowledge of the more than 255 [10] NoSql databases in the market. The proper data migration technique is necessary to avoid the project to freeze or tie the developer to suboptimal solutions.

The ability to migrate the data between different types of NoSql Database to allow greater flexibility to systems and is the focus of this dissertation.

[1] In the database systems, a schema can be defined as a structure, or a blueprint, which defines how the database are constructed, this definition is in a formal language which is supported for a DBMS. From this point of view, the schema is a definition of how the data will be grouped and connected, this defining in a elementary way the kind of model one specific DBMS are able to understand and manipulate.

The process of data migration has been well studied. The data migration between RDBMS and NoSql Database has been discussed in the works of [11, 8, 12, 13, 14, 15, 16, 17, 18]. Different approaches and purposes guide these works making difficult to find a common point between them.

Specifically for the data migration using NoSql Databases the work done by [19, 20] are representative. These works are designed for data interchange between columnar databases. In a different approach [21, 22, 23] proposes a creation of an API (Application Programming Interface) intended to provide access to the data in a standardized way. These works act as a wrapper forcing the creation of "plug-ins" for each new DBMS used. For the migration between databases within the same category [24] proposed the creation of a domain-specific language to transform one version of the data to the new one.

In this dissertation, the notion of different data models used by NoSql and described in [5] are used to be able to achieve a data migration. The migration is between different databases with data models that may or may not differ.

1.1 Problem Definition

As noted by [25] one of the biggest problems in the data representantion an is precisely the lack of standardization which exists today.

These lack of standards occurred due to the fast search for new properties and capabilities in the NoSql world. And without any formal rules to guide the process, a plethora of solutions for the same problem has emerged [26]. The choice of a NoSql Databases must be made taking into account the application characteristics and needs, not only the initial ones, but also the future ones, most of them unpredicted.

All these factors lead to the need to migrate data and the necessity to change on NoSql Database to other.

In this dissertation, we address the problem of data migration between different NoSql Databases.

One or more of the following factors contribute to the data migration necessity:

- Different data stores are required to be used at the same time
- Technology decoupling is a requirement
- Some part or the entire data must be migrated
- Changes in scope lead to the use of different databases

Today there is no an approach that focuses on the necessary transformations. The works of Bugiotti [27, 28] shows the possible models that a data register can take on. This work uses these models to propose the necessary transformations to perform data migrations.

There are two main methodologies identified that are being used to address the data migration problem:

Focusing on the representation The method in this area create formats for generic representation of the data [5]. This method concentrate in the data accesses through wrappers using get and put methods [21, 29, 23]. Thus,the focus of this methods are the data recovery,not the translation between the different data sources. The problem is that in many cases the translation is required.

Focusing on the translation These works write translations between specific NoSql Database [20]. These translations include a limited number of systems, being generally between two distinct databases. Similar approaches exist specifically intended to migrate data from an RDBMS and NoSql Database. The absence of a generic format makes difficult to implement new translations.

The heterogeneity of NoSql models which exists today makes almost impossible to map each case or each possible combination. The proposed solution will make use of points which are common to all NoSql Database, such as the similarity of the models and the presence of basic operations which are common to all of them.

1.2 Objectives

The purpose of this dissertation is the creation of an translation between NoSql Databases based on generic data representation and modeling. Thus enabling the data translation aiming to make easier the migration between different data sources.

This work deals with the data migration problem taking advantage of these two approaches described in the previous section. The solution presented in this work perform the data translation between NoSql Database using a generic representation. The combination of both techniques gives the advantage of being generic and able to understand a wide range of NoSql Database. The use of this technique makes it easier to accommodate new models or systems.

The mappings or tools available nowadays, that are able to migrate data between the different NoSql Databases types are focused on specific models or in a specific NoSql Databases. If a system destined to migrate the data is not found the necessary changes in the format of the data to accommodate in the destination NoSql must be coded individually.

In this scenario, if an application needs to execute the migration of the data from one NoSql to another, all the work ends up being manual. In large databases, this restriction ends up restricting or making this process impossible.

All of the projects found that involve the creation of specific modules for each DBMS, or are targeted for databases within the same category. These approaches lead to a lack of genericity and impose difficulties in extensibility or adaptability. At the best of my efforts, no one was found which implement data migration among NoSql's of different categories without the need of creation of new and specialized code.

A generic data representation format intended for data interchange will be used to assist in the data migration work. The use of a model and an intermediary data format are common in many solutions. However, the existing works do not cover all models by restricting the migration possibilities. The template created is specifically designed to migrate data regardless of schemas. This template cover the models specified in the work done by [5].

1.3 Contributions

This work presents the development of transformations and models which includes a creation of algorithms and specialized models which are independent of the specific data models involved. In this way, they must have a visibility of the general structural features of the different models.

As the goal is to perform the data migration, is required that data be manipulated in some sense, specifically with respect to the model use to represent the data. It is assumed in this work which the basic operations to access the database are available and are capable of being performed in any of the DBMS. So even with different names or approaches, if only these basic structures are present,it will be possible to achieve the data migration between the various systems. As the final result, all the data stored in the original system must be transferred to the target system, without loss of information, or wrong conversions which will eventually disfigure the original data.

The development of a model which is capable of understanding different NoSql models is not a trivial task. Some activities which were performed to accomplish this task:

- A revision of the state-of-the-art to learn the different NoSql Database which exists, mainly the differences in reference to their models. This task was particularly challenging due to a great variety of NoSql which exists today.
- Build an model able to understand the different data models used by the various NoSql Database.

- Build an translation which transform the different data models among them.
- Develop different tests to be sure which the model translation and consequent data migration can be performed as expected.

1.4 Document Structure

This dissertation is organized into five chapters. Chapter 2 contains an analysis of the state-of-the-art about NoSql documentation, including the various features, models and possible classifications which can be applied. The proposed metamodel is described in Chapter 3 altogether with the system requirements and the necessary workflow. In Chapter 4 the data migration steps are discussed altogether with the experiments executed in this work. Finally in the chapter 5 the general conclusions of the dissertation and the future work are discussed.

2 State-of-the-Art

In this chapter the state-of-the-art of NoSql Database is presented. In the 2.1 section a brief reflection abut the NoSql motivation. Section 2.2 talks about the common features of most NoSql Database. Section 2.3 discuss about the different classification possibilities, and the section 2.4 explains the differences between the data models, as well as the difference inside the own models. Then in the section 2.5 the data migration process is explained.

2.1 NoSql Data Models

As defined in [10], all databases which are non-relational, distributed and horizontally scalable, can be defined as a NoSql system. These attributes are intended to ensure efficient and convenient access to large amounts of persistent data.

RDBMS integrates a large package with various kinds of features to give a lot of functionalities intended to cover most of the application needs. For the traditional uses is convenient to have all those features in one place [30]. With the evolution of the applications and the increase of the Web applications, developers start to notice that their data does not adapt properly to the relational model. Mainly because the main operations needed over this data are quite simple [29].

For some applications, the relational model and its main features start to become very complex and unnecessarily heavy [29]. Developers perceived which they are not useful and leave some of those features out are more practical and logical [25] [21]. A new group of databases systems called NoSql has emerged for some kinds of problems which the use of an RDBMS is not the best choice [31].

The first use of the NoSql expression was credited to Carlo Strozzi. In 1998 he used the name to designate his new created database, which is a relational one, but intentionally don't use the SQL language for searches [32]. The term NoSql starts to be used in a larger number of data management systems which intentionally abandoned the support for ACID transactions (Atomicity, Consistency, Isolation, Durability). Although currently not all NoSql Database have abandoned support for ACID. In this scenario, the term "SQL" refers to an RDBMS, and all kind of data management system that doesn't rely on the relational model started to be called NoSql Database. The term NoSql is more often interpreted as "Not Only SQL".

2.2 Common features of NoSql Databases

The main feature of any database is the ability to store and retrieve data in the long term, called persistence. This characteristic is also important in NoSql type applications. For some NoSql systems, flat files are often sufficient as a storage mechanism, rather than sophisticated and specialized structures. The way where the data is archived and manipulated are subject to a big variation from one software to other. Whereas, some features are common in the majority of the NoSql Databases. Although these characteristics are important because they help to define the data model that can be used, for this work, only the format how the data is designed and its structures are used.

Schemaless Data Structures Almost all NoSql implementations store and manage their data without an explicit schema [33]. It means that during the phase of definition and creation of the data structure little fixed rules exists. The data structures can be done with little or no care about internal structures. It is assumed that is always possible to evolve this structures over time, and adding new attributes whenever necessary. This characteristic allows a great flexibility during the development phase. As a consequence, the responsibility to understand and create a logical connection between the records becomes the application responsibility.

Is important to note that the claim which NoSql Database are completely schemaless is controversial [5]. What is not considered in this statement is the fact which the data layout must be defined by the application. The RDBMS do not interpret the data, but the data has to conform to the definition created into the database. In NoSql Database the application must define the data format. If a schema mismatch occurs this error will be throw by the application instead of the RDBMS. Thus, even without an explicit database schema, the design of the data must be taken into consideration when changes in the data structure are made.

Replication As described in [34] replication is based on a copy of the data which are stored, at least in two, or preferably on several nodes of a network. This method is used to improve the system reliability and helps to minimize data loss in case of a failure of individual nodes or even an entire cluster. Another gain of this technique frequently cited is the fact which replication can improve read operations. In contrast, a significant drawback is a fact which is not simple to maintain the consistency of the replicas.

Sharding Some nodes of the system can contain a piece of the original data, called shard, where they can perform the required operations in the shard which the node owns [35]. The main objective of this method is to attain horizontal scalability of the system. The main drawback however, is the fact which the operations which need

data from several objects, potentially involves a considerable number of nodes, which will require an intensive data traffic over the network. Another problem is related to the number of nodes. Each time which a new node is added, the probability of faults increases as well. The way to counter this side effect is to implement more redundancy, in other words, more replication. Administrative tasks like control the distribution of data over the nodes, perform workload balancing, and others can contribute to the increase of the complexity of the solution.

CAP theorem Differently to the traditional, the consistency model on which NoSql Database are based describes the physical consistency of the state of the data on different nodes[36, 37]. Eric Brewer was credited to develop the conjecture which in distributed systems, there is a fundamental trade-off between consistency, availability and tolerance partition[38]. The ACID theory assumes the consistency as a logical attribute in the sense of data integrity defined in the design phase. This assumption was formalized in 2002 by Seth Gilbert and Nancy Lynch and then started to be called CAP theorem. The three points on which this theorem is based are [30]:

Consistency Consistency refer to atomicity and isolation. This means which all processes which are running concurrently, may be able to view the same version of the data.

Availability This means which when requested, even the system and the data are fully available. In other words, this means which each request eventually will receive a response. In this case the performance of the response is not an issue.

Partition tolerance To achive this point, the system must be able to work properly, even in the case of parts, of some of the components of the system fails. It is assumed which the communication between the servers it is not 100% available during 100% of the time.

As noted by [39] almost all DBMS provides basic operations if not directly by their API does use attached programs. And beyond these core services, each NoSql can offer more capabilities such a more powerful language or native data access using proprietary API's.

All the studied system offers, at least, simple operations to access key-value pairs [21], like:

- set(key, value) this function can be utilized to add or modify an key-value pair
- get(key) is used to retrieve, from the database, the value associated with a specific key

- delete(key) deletes from the database the key and the value associated with which key.

Because these basic functionalities are fundamental for any database system they are present in all NoSql studied, also in some cases, these operations have other names.

2.3 Lack of criteria for classification

Most of the products need to identify themselves as a NoSql Database [10], mainly because, differently the well studied and known relational databases, does not exist a general rule to be applied. This lead to a wide variety of architectures and designs, each of them focused in to solve one particular point or add one desirable characteristic. Even the way where data storage is archived is a subject of variation among implementations, making it difficult to create a single standard for classifying the databases. However, four major sets of characteristics tend to be employed when a classification is proposed. In some cases, more than one of these characteristics may be combined in an attempt to achieve a better representation. The first three classification methods will be briefly described, and the most used, on which this work is based, will be presented in more detail.

BASE principle The acronym BASE (Basically Available, Soft State, Eventual consistency) was created specifically to contrast to the ACID paradigm. The BASE principle [40], which is followed by most of the NoSql Database, implies which data consistency must be guaranteed by the developer, and should not be handled by the database. BASE comes from the paradigm where data is distributed, and synchronization of data is too complicated, and a significant loss of performance is not supported. The importance of understanding how the BASE principle is implemented derives from the fact which the necessary basic operations, CRUD, are directly affected. Thus as the consistency is less relaxed and the greater the need to ensure replication, the basic operations turn out to be affected [41]. At the same time implement a system which manages heterogeneous databases with different implementations of the BASE principle [42] demonstrates to be challenging, and some research is still in place.

Different APIs for data access An Application Program Interfaces (API) serves as a middle-layer between the database structures and the application. The APIs aim to encapsulate the database layout by providing high-level access to data, and thereby isolating the application from data layout. This ends up creating a huge dependency between the application layer and its persistence layer, since a change in any one of them implies, inevitably, in changing the other. Almost all of the databases provide basic CRUD (Create, Read, Update, Delete) operations, even if implemented in

different ways, and likewise, they provide access to a standard data exchange format such as XML (Extensible Markup Language), JSON (Javascript Option Notation) or BSON (Binary JSON).

In general, can be said which each product implements their low-level query language, or in some cases they simply don't implement any querying method, which leads to a complete lack of standardized interfaces.

Different ways of applying the CAP theorem As described in 2.2 only two points of the CAP theorem can be entirely implemented simultaneously. So the system can be classified based on which of the properties of the CAP theorem they implement [43], and divided into three categories:

- CA systems: Consistent and highly available, therefore not partition-tolerant.
- CP systems: Consistent and partition-tolerant, therefore not highly available.
- AP systems: Highly available and partition-tolerant, therefore not consistent.

2.4 Data Models

The most used classification method is based on the data model implemented. This categorization uses the way where data elements are organized and how this data elements relate to one another. A data model explicitly determines the structure of data. Based in this criteria four main categories are cited:

- Key-Value stores
- Document databases
- Wide-Column (or Column-Family) Stores
- Graph Databases

The kind of data which are stored in a NoSql Database is a set of objects[1].

All of the DBMSs listed in this study are capable to store scalar values, like numbers and strings, and generally they can handle BLOBs (Binary Large Objects) to. Some of them also provide a way to store more complex data like nested or referenced values [5].

As most of the NoSql Database are schemaless, this implies in which even the objects which are part of the same collections, do not necessarily have the same structure. And is

[1] An *object* is analogous to an object in the modern programming languages, but without the associated set of operations which can be applied to it. Only their attributes.

quite common which during the evolving time of the system, and with the modifications needed, this objects, although express the same concept, have different structures.

All the examples used in this section are based in a representation of the data from one person. The person data that is managed are their first name, last name, age and phone number. The phone number is composed of a type and a number. This example are extracted from [44].

It is assumed that each object is composed by a unique identifier and a complex value. Also it is assumed that these values can nest the value of other objects; for example, one person friend can be linked to an other person register and their values.

A simple example of a registry of one person object can be seen in the figure 1.

Figure 1 – Person data Object representation.

2.4.1 JSON

According to [45] JSON was first available in 2001 and it is an acronym for JavaScript Object Notation. In this standard format the data are expressed in a pure text readable for humans. JSON was intended to ease data interchange between programming languages in a simple but structured way. Inspired by the object literals of JavaScript it shares a subset of ECMAScript[2] textual representations. To be able to reach this compatibility JSON offers the representation of numbers as a text, and texts are sequences of Unicode code points or a mere sequence of digits. Each language or application must be able to make sense of this representation assigning a type (integer, float, decimal..) and the correspondent capacities. This is done to guarantee the interchange of the data even if different applications deal with data types in a diverse manner. JSON provides a simple notation for expressing collections of name/value pairs[46] and this collection can be used to represent a wide variety of structures like record, struct, dict, map, hash or objects. Also provides support for ordered lists of values, which can be called arrays, vectors or lists. The representation of more complex data structures can be archived nesting objects and/or arrays. However, JSON isn't the best choose to represent data in binary form. It Is expected which the definitions and rules of the JSON standard will not change very

[2] www.ecmascript.org

often ensuring great stability to the notation. Each JSON document is an object enclosed by curly braces {} and each name or key are separated by a colon ":" and the sequential or subsequent values are separated by a comma ",".

Another important feature for this work, is that JSON documents have for each atomic event an associated event [44]. These events are then labeled according to the fallowing list.

- END_ARRAY — End of a JSON array.
- END_OBJECT — End of a JSON object.
- KEY_NAME — Name in a name/value pair of a JSON object.
- START_ARRAY — Start of a JSON array.
- START_OBJECT — Start of a JSON object.
- VALUE_FALSE — false value in a JSON array or object.
- VALUE_NULL — null value in a JSON array or object.
- VALUE_NUMBER — Number value in a JSON array or object.
- VALUE_STRING — String value in a JSON array or object.
- VALUE_TRUE — true value in a JSON array or object.

The main aspects of the JSON definition are used for most of the NoSql Databases. Document stores are a particular case because they can store their data directly as a JSON object. All the studied NoSql Databases are able to export their data in the JSON format programmatically. It makes the utilization of this data format a natural choice for the research.

An example of a JSON object can be seen in listing 2.1.

```
{"Person":
  {"firstName": "John", "lastName": "Smith", "age": 25,
     "phoneNumber": [
        { "type": "home", "number": "212 555-1234" },
        { "type": "fax", "number": "646 555-4567" }
                    ]
  }
  {"firstName": "Sophia","lastName": "Owen", "age": 22,
        "phoneNumber": [
                { "type": "home", "number": "212 555-4321" },
                { "type": "fax", "number": "646 555-9876" }
```

```
                    ]
  }
}
```

Listing 2.1 – JSON Complex Object Example

2.4.2 Key-Value Stores

In key-value stores each value is identified by a unique key, which is, the system stores the values and an index (the key) to find them, based on a programmer-defined key [47]. Formally is a collection defined by $(\mathcal{C}_k v = (k1, v1), ..., (kn, vn))$.

These databases are schemaless collections of key-value pairs, and each key-value pair is a single record in the database. These keys are used to perform the basic operations, such as insert, remove and recover the associated values. The values associated with these keys can be virtually anything, from simple untyped elements, more simple structures like strings and integers, or even more complex ones, as structured objects, relying only on the DBMS characteristics as capabilities [31]. This means which the definition of what keys and values are subject to huge variations, as well as the operations which these systems offer to access their groups of key-value pairs. All the systems in this section store sets of attribute-value pairs, but they use different data structures to achieve those goals. According to [2] the main strategies for representing data in key-vales stores are:

- key-value per object - *kvpo* (See Table 1)
- key-value per field - *kvpf* (See Tables 2 3 4 5)

The key-value per object - *kvpo* strategy is based on the fact which for a single key exists only one object associated. This key has a collection name and one identifier of the object. The value is a serialization of the whole complex value of the object. A second strategy uses multiple key-value pairs for each object. Specifically, it adopts a key-value pair for each top-level field of the combined value of the object (key-value per field, kvpf). The key is composed of the collection name, the object identifier, and the name of the top-level field.

Table 1 – Key-value per object [2]

Key	*Value*
Person:John	{"firstName": "John", "lastName": "Smith", ...}

As shown in Table 1, an single key-value pair, with the key formed by Person:John and the value formed by {"firstName": "John", "lastName": "Smith", ...} can be represented

in a simple form using the *kvpo* strategy. In this case, the value field is a simple serialization[3] of the entire value of the object.

Is possible to use multiple key-value pairs for each object, Table 2. In this kind of operation, a key-value pair is created for each top-level field of the value of the object. The key is formed of the collection name, the object identifier, and the name of the top-level field. The value is the original value of the field. Using the example presented in 2.1, the representation will use four different key-value pairs, one for each of its declared fields.

Table 2 – Key-value per field [2]

Key	*Value*
Person:John/firstName	John
Person:John/lastName	Smith
Person:John/age	25
Person:John/phoneNumber	...

Some implementations use a hash to represent the various fields of an object. Using a single hash for a whole object, and a field-value pair for each of its top-level fields, as shown in Table 3

Table 3 – Key-hash per field [2]

Key	*Value*
Person:John	firstName:John lastName:Smith age:25 phoneNumber:...

NoSql Database can composes their key with a combination of a major key and a minor key, Table 4. In this case, the major key must be a non-empty sequence of plain strings. The minor key is a possibly empty sequence of plain strings. Each element of a key is called component. The symbol / separates components, and the symbol - is used to separates the major key from the minor key.

Table 4 – Key-value per field [2]

Key	*Value*
Person/John/-/firstName	John
Person/John/-/lastName	Smith
Person/John/-/age	25
Person/John/-/phoneNumber	...

The last model utilizes multiple key-value pairs for each object, using atomic values only. In this case, a key-value pair is used for each atomic value contained in the value

[3] *serialization* in the context of this document, is the process of translating data structures, an object for example, into a text format which can be stored and later reconstructed

of the object. The key is composed of several fields like the collection name, the object identifier, plus the sequence of all the fields which is necessary to go to locate one particular atomic value. Table 5.

Table 5 – Key-value per atomic value [2]

Key	*Value*
Person/John/-/firstName	John
Person/John/-/lastName	Smith
Person/John/-/age	25
Person/John/-/phoneNumber/0/type	home
Person/John/-/phoneNumber/0/number	212 555-1234
Person/John/-/phoneNumber/1/type	fax
Person/John/-/phoneNumber/1/number	646 555-4567

2.4.3 Columnar Databases

RDBMS keep the records of the tables, preferably, continuously on the disk or if possible in the same block. What is intended to be fast for writing operations, and fast for reading entire sets of records. As a drawback, when are necessary to read only one or few columns, but many records, this structure tend to be far less efficient. Columnar Databases, also called extensible record store for some authors [48, 49], is a datastores organized around columns, tables and rows. So column databases are designed to optimize the reading of columns, or groups of columns. The adopters of this technology states which the columnar databases are most attractive for OLAP operations. In this sense, contrary to the structured relational databases which uses tables with columns and rows with uniformized fields for each record, column-oriented databases contain one extendable column of related data[30] and can be defined as a table which is sparsely populated [1]. These databases are predominantly schema-less, and each row can have its set of columns. To distribute data, rows and columns can be used to shard data over the nodes. The main inspiration for most column-oriented datastores is Google's Bigtable, what makes these databases appropriate for applications which access a well known and restricted subset of columns for each request.

2.4.3.1 Columnar Models

Robinson et al.[1] creates a visual representation which explains the four methods which the columnar databases typically utilizes to group your models, Figure 2.

According to [2] the mode which these databases implement their structures can be defined in the following way: Each object is stored in a distinct row, having two columns: id for the object identifier and value for a serialization of the whole complex value of the object or the many collections of other columns.

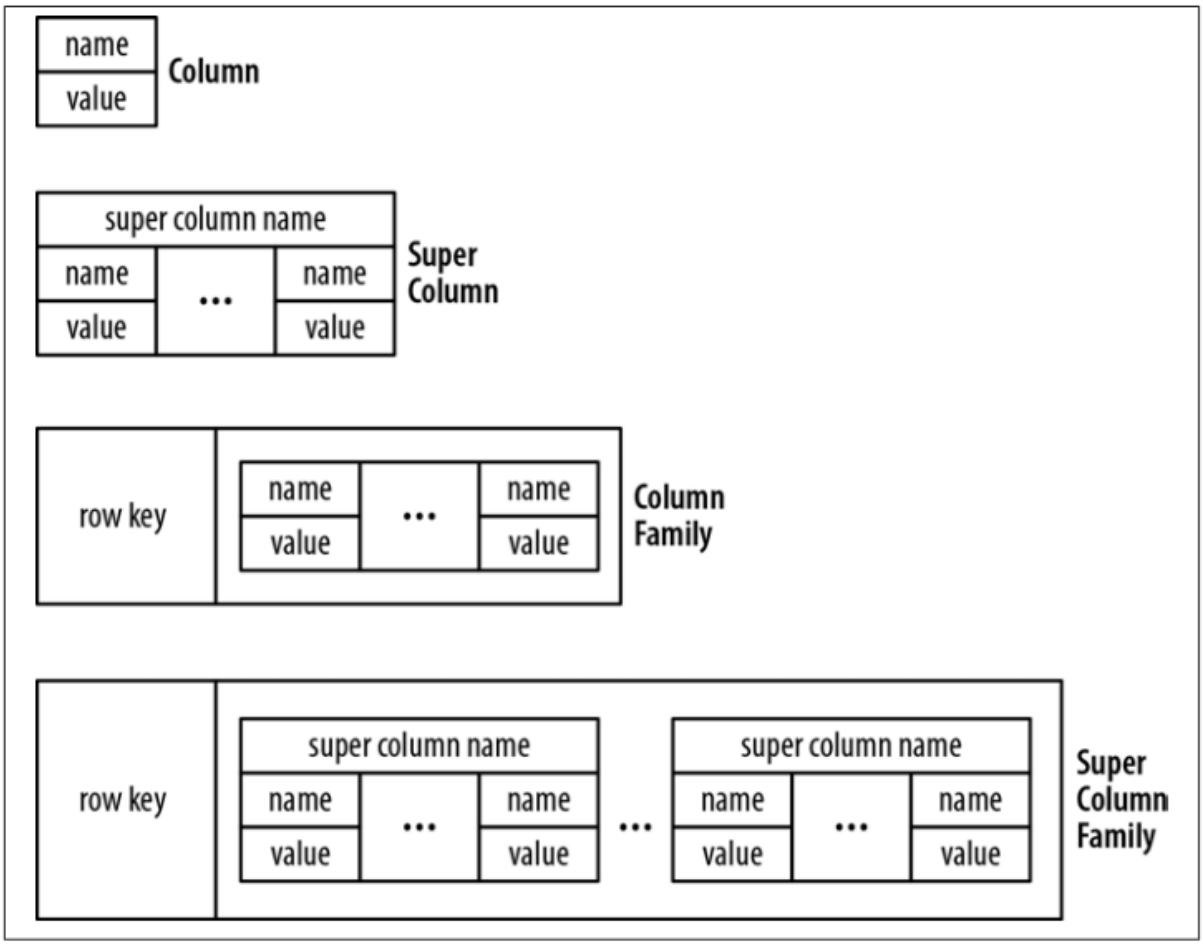

Figure 2 – Representation of the four building blocks of column family storage [1].

A *Column* extend the key-value model by organizing keyed records as a collection of columns, where a column is a key-value pair. The key becomes the column name, and the value can be an arbitrary data type such as a JSON document. In these models, pairs of key/value[4] could be grouped into lines.

A *Super Column* is a collection and may contain records which other columns, so each column is a group of other columns, and these groups are stored and manipulated based on a "Super Column" name, which can be defined as a Key part, and the columns group itself determine the value.

In a *Column Family* approach, columns are stored in rows, and each row key, the "Column Family" contains a group of correlated columns only

The last model is known as *Super Column Family* are composed of groups of "Super Columns", so one Key is used to reference multiple "Super Columns".

From the storage system point of view, [2] states which can be implemented a row per object (rpo Table7) data representation strategy, another possible approach is the use

[4] Note which value, in this case, isn't the entire row, only one or a group of columns

o a single cell for each object (cpo Table8) model.

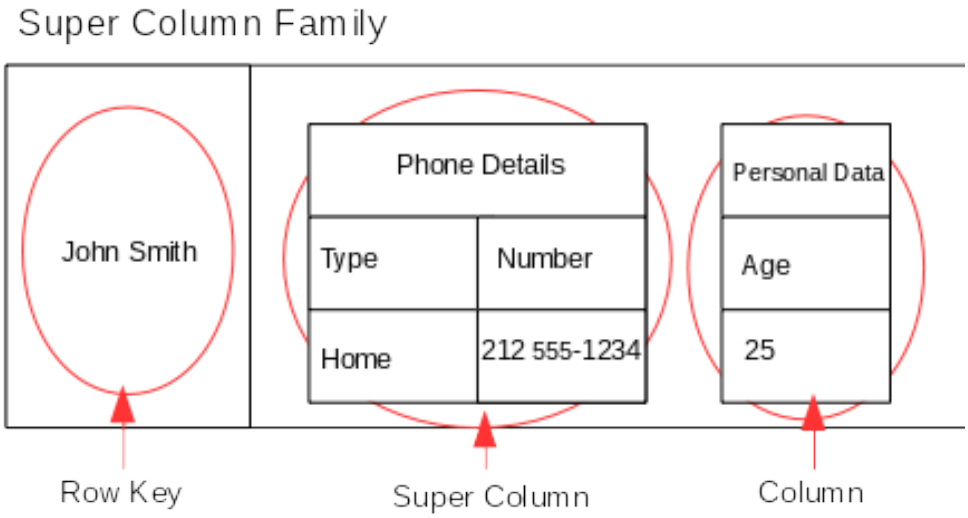

Figure 3 – Example of a Super Column Family data representation.

2.4.4 Document Store (DS)

The document store models are designed to manipulate and persist a wide diversity of complex values [50], which can comprise scalar values, lists, and other documents in a nested format. These documents are organized into collections of objects, nominally a group of documents. They can be encoded using formats already established equal Javascript Object Notation (JSON) or Extensible Markup Language (XML). This Documents must have keys and secondary indexes which can be built on non-key fields. The idea is to format the document to be self-descriptive. Each document can have their characteristics, allowing that in the same collection different formats of the same document can coexist[30].

Like Key-Value stores 2.4.2, some variation exists in the format in which the data is stored[2], though belonging to the same model and be capable of storing the same documents, the three main variations are described in the tables 6, 7 and 8

Table 6 – Document per object [2]

Collection	*document id*	*document*
Person	John	{" id":"John", "firstName":"John", "lastName":"Smith", ...}

Table 7 – Item per object [2]

table	*_id firstName lastName age phoneNumber*
Person	John John Smith 25 { ... }

Table 8 – Cell per object [2]

table	*id* *value*
Person	John {"firstName":"John", "lastName":"Smith", ...}

2.4.5 Graph Model

Graph databases organize their data in structures which are connected in some way, more commonly some form of a directed graph. The way which the Graph databases use to represent their data as nodes, edges and properties. Is important to note which the properties are key/values pairs. Nodes can represent entities, and the edges are the connection of two nodes and represent the relationship, and the properties are the data itself [51].

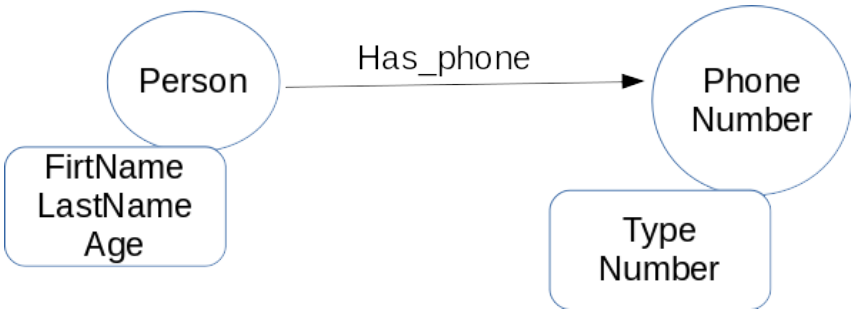

Figure 4 – Graph Representation.

They are best suited to deal with problems involving graph traversals and sub-graph matching. One deficiency is the fact which horizontal scaling are not efficient. Graph databases are built for use with transactional (OLTP) systems [1] and commonly offer ACID transactions providing a more high consistency than the average NoSql Database. In Graph Databases relations can occur in any direction, and the nodes are linked with more than one other node [47]. Is best fitted when the relationships between nodes are the key.

2.5 Data Migration

Data migration can be defined as the process of transfer data from one database, called source, to other database called target. Another scenario is the migration of the data from one version to another version in the same database. In both cases some typical steps which are be performed are:

1. Connect to the source database
2. Connect to the destination database
3. Read the data from the source

4. Transform the data to the destination format/model
5. Perform the insertion of data into the destination
6. Control and treat any errors during process execution

It is during the step number 4, which a manual translation and the mapping of the model are required and must be performed. Primarily because the business rules which are implicit coded and enforced into the model should be understood. Then to be translated to a physical database schema. It is at this point that this work act, creating a process that facilitates the transformation of the data.

Even in a high standardized system, like RDBMS, and well-documented database this process are prone to errors. Because the translator must know about business rules and the underlying physical target model, which is a complex task. This scenario becomes even more complex in NoSql Databases, due to their schemaless nature. These databases hinder the understanding of business rules, and each object can contain different structures. There is no guide or framework which can assist in model comprehension

Another way to understand the model is to be able to abstract the data model by recognizing common concepts in the data model of various NoSql solutions, to define a more general metamodel and map more easily, or ideally automatically, from the source to the target model. Thus the most challenging part of the process can be automated, reducing errors and minimizing the need for the understanding of the business model to perform the migration.

As seen previously on Section 2.4, the same data can be expressed in different designs and styles. In this scenario, to be possible to perform the data migration among the various NoSql Database the best approach is one which can be generic and reproducible to all models. Thus maintaining a greater independence of the data in relation to the model being able to cope with the peculiarities and differences present in each analyzed models [52]. So what is sought is to create an application which can work with the widest range of possible models, reducing the need to create custom solutions which server only for a couple of models at a time.

A more detailed discussion about how this task will be implemented is performed in Chapter 3.

2.5.1 Data migration between NoSql Database and NoSql Database

In contrast to the data migration between RDBMS and NoSql Database, the data migration between NoSql Database and NoSql Database is a understudied area. All the projects studied in the Section 2.5.2 can be categorized into three major areas of research:

Common Data Representation The first strategy is based on the creation of an interface which unifies the way in which data is retrieved and managed in the databases, building a representation which is intended to be common to all database managers. Two points which arise in this approach are the fact which this language created most be adopted for all the database managers to be effective. The second thing is the fact that for each new database which can be inserted in this model, a construction of a piece of software to adapt or translate the language used in the DBMS to the new language is needed.

Point-to-point translator The second strategy is the construction of a translator which are responsible for performing the database-to-database mapping. To map the data from a source to a destination is necessary to build a peace of software which knows the specificities of each database involved. The main advantage of this method is that all the details are taken into account during the migration process, being, generally, more accurate and subject to performance enhancement. The drawback is the fact which if a new database needs to be supported, at least two new translators need to be written.

Metamodel The third strategy is based on the construction of a metamodel able to understand the differences between the different database managers. The works studied which uses this method end up creating the metamodel to perform the migration between DBMS's which are the same data model, just taking care of the details like indexing, or between specific systems, from one program to another only.

In this work the third approach will be used, however with a broader view, analyzing the details which are common to all models, and creating a metamodel which covers these common functionalities. Further discussion about this method is carried out in Section 3

2.5.2 Related Work

Three major approaches are studied in related works. The first deals with the migration between RDBMS and NoSql. The second one deals with the migration between different NoSql Databases proposing the creation of a common language or a common interface. The third deals with the problem with the creation of a metamodel capable of dealing with several models.

2.5.3 Data migration between RDBMS and NoSql Database

Most of the system which intends to migrate data between RDBMS and NoSql Database follow the same pattern, read the RDBMS metadata, with specific code blocks for each RDBMS, and then using a predetermined mapping with a deterministic algorithm

mapping each entity to their destination. An example of a possible transformation steps from a RDBMS and a column NoSql (2.4.3) can be seen in Table 9. This method becomes restrictive because every change or new DBMS which want to support new codes must be written specifically for the new process.

Table 9 – Common RDBMS to NoSql migration mapping

RDBMS Object		NoSql Object
Database	$\longrightarrow$	Collection
Table	$\longrightarrow$	Entity
Column	$\longrightarrow$	Attribute
Foreign keys	$\longrightarrow$	Relationships (when available)

The Work work [11] analyze the data structures of the RDBMS and the NoSQL databases and to suggest a Graphical User Interface (GUI) tool which migrates the data from a traditional RDBMS to NoSQL databases, specifically MySQL and CouchDB (a columnar database).

In [13] the authors create one framework called "*NoSQLayer*", this framework was divided into two modules, one for metadata extraction and mapping, and other which perform the migration. The migration process is tested between MySQL and a MongoDB (a Document Store). This work focuses in query executions, more than in a data migration.

[15] makes a comparison between RDBMS and NoSql Databases, based on their scaling model and data model, so the author lists some criteria for choosing a NoSQL, comparing their main functionalities.

[16] focuses their work on the schema transformation, this transformation is carried out among an RDBMS and a columnar NoSql Database, the authors read the RDBMS metadata, searching for Primary Keys and Foreign Keys and then concatenate the columns associated with this keys in a typical Columnar NoSql model.

The work from [18] proposes a schema conversion model for transforming an RDBMS to NoSql to MongoDB (a Document Store), the work focus on a transformation of schemas intended to provide better support for queries with join operations. The method utilized is based on a graph transformation system, which reads and store in the form of graphs the schema definition from both databases, then based on the foreign keys the system can build a document to be utilized in the destination database, which concatenates the sequence of data from a child to a parent.

[53] Sqoop is a tool developed by Apache which aims the transference of bulk data between an RDBMS and Apache Hadoop (a Column store NoSql). The tool was designed to work in both ways so that the data can be transferred from an RDBMS to a Hadoop and vice versa. To accomplish this task, the tool first reads from de source database the metadata describing the schema of the data to be imported, then transform the data in a

Hadoop format using MapReduce to achieve parallel processing and some degree of fault tolerance, after which the data con be exported back to an RDBMS.

In their work [54] uses the XML as a mid-term to convert data, but in relational databases. To archive this the data is "serialized" and then represented in an XML, the data is restructured to adapt to a target model and then finally rewrite in the new format. Based on the combination of predefined functions.

[12] creates a comparison between two data migration methods, the first one is empirical and based on common knowledge, and the second one uses one guideline, created by the authors.

2.5.3.1 Data migration between NoSql Databases

A framework called CDPort is proposed in [23], this framework aims to build a data model and an API which is an attempt to create a standardized way of access for RDBMS and NoSql Databases both of them stored in the cloud environment. In the Data Model created, the objects are modeled in so-called "entities, " and these entities have the following characteristics:

- Entity type: is an attribute used to categorize each stored entity.
- Key: Is the attribute which identifies each entity and must be unique
- Property: A description key/value attribute. Each entity can have multiple of this properties.

The focus of the work is to create an API able to access, with the same commands, different data structures.

In his work [55] proposes a mapping language called xR2RML, intended to convert to an RDF format heterogeneous data formats, extending the work done by [56] for a NoSql Databases. They work mapping and explaining some textual data formats including JSON.

[17] creates a GUI which can connect in a Hbase, a column model, and process SQL query's over this data based on MapReduce techniques, being this way able to process unstructured data. The authors have studied the data models involved, mapping these characteristics. So the complete process can be summarized as follows:

- parse the SQL Query submitted
- do the necessary transformations to translate the SQL logic into Hbase Model
- generate the MapReduce code to process the Query

- wrap back the results to be displayed in the GUI

Despite the work focuses on conversion of queries, the study on the difference of the models also is used to conduct a migration.

[21] try to manage the heterogeneity of the interfaces available to access data in the NoSql world. So they propose a programming interface intended to be common to NoSql Databases and which can be extended to a RDBMS, called Save Our Systems (SOS). The solution is divided into three main components,

- a standard interface
- one meta-layer responsible for storing the form of the data
- specific handlers for each database system.

The meta-layer is composed by three main conceptual elements: Struct, Set and Attribute. The author describes the metamodel as formed by a "construct" each of this constructs are composed of a name and value. The values are composed of basic types. These constructs are grouped into more complex elements, called structures and sets which are combinations of constructs or also other structures and sets. When mapped to Key Value a construct is equivalent to an object; when mapped to document stores each object is translated into a collection. In column store tables are created for each collection, with the constructors serving as columns.

[57] addresses the problem of data migration between the same database but with mutable schema design, to achieve this, they use a Datalog model for reading, writing, and migrating data. To evaluate their work, they used algorithms already tested in both bottom-up and top-down depending on the rules being evaluated. The application was created to guarantee which the migration will remain transparent, even if the data has been migrated eagerly (before the new model release) or lazily. The author states why JSON was not chosen for this work because in this format data was represented in a hierarchical structure with multi-valued entities, which does not meet the needs due to the fact which the work focus on the semantics and the order of data migration

In an article based on his thesis [20] create a system for migrating data between NoSql columnar databases. He creates a client/server application which uses a metamodel designed solely to handle whit columnar databases, taking into account details like indexing. Their metamodel is a representation of all the column families, very similar to what can be seen in Figure 2.

The main propose of the work in [24] is helping in the evolution of the data models in NoSql Databases. They created a tool called KVolve which manages the changes in

data model using a domain-specific language, the changes applied in a lazy way allow the application to keep working during the process. The data transformation from one version to other are done using functions written in Python which is responsible for interpreting the domain language created and apply the changes.

In their work [58] focus in the migration from a Document Store to a Graph Database. The process is done through data standardization and classification, aiming the data consistency and correct mapping. The work also focus in the performance of the migration.

2.5.3.2 Metamodel Strategy

[29] is an extension of the work in [21], with examples of the framework previously proposed running on simplified version of Twitter and the result of the tests. The focus in this paper is in the interface utilization, more than the metamodel description.

Model-Driven Engineering along with Program Transformation are used by [8] to mitigate the PaaS vendor lock-in problem. The work focus is how to migrate applications from one platform to other. They use an ontology called KDM for system knowledge extraction, and after which using some pre-defined patterns they guide the users during the migration. Some external parameters, like price, can be used to choose the new platform.

A series of articles present the NoAM (NoSQL Abstract Model) [5, 28, 59] which is a methodology to database design. These works are based on the observation which the NoSql Databases, independently of the final model used, share some similar features, specifically the capacity to access their data in what was called "data access units", mainly varying their granularity. So NoAM is explicitly an "abstract data model" which represents the data in a series of formats which later can be refined and specialized to each NoSql Database desired.

In work [28] the focus is put in describe a data modeling and a data design methodology to ensure with the data can be represented in the major NoSql Databases models, and this generic model can be refined or redesigned to better accommodate in the chosen NoSql Databases database This work is a direct derivate from [5] when the database design problem are mainly addressed. The work tries to represent the data in an intermediate representation which is then implemented, or more precisely translated, in a target NoSql Databases, so each specific features which need to be applied can be taken into account.

The representation exposed in Section 2.4 is a direct application of the work presented in the NoAM Data Model, which is in short, which data already in this final format.

2.5.4 Literature Summary

As stated in [20] the creation of a metamodel is largely used. These metamodel stores the common characteristics of the different databases which will be migrated and can correctly map these components. Therefore by knowing the metamodel the system can write specific translators for each database supported. These translators via one intermediary model can perform the migration of the data and his model between the databases.

In Table 10 a summary of the literature was shown.

Table 10 – Main properties of some related works

Authors	**RDBMS**	**Key value**	**Column**	**Document**	**Other**[5]
Migration from RDBMS to NoSql Database					
Mughees[11]	Yes	No	No	Yes	No
Gomez et al.[12]	Yes	No	Yes	No	No
Vale & Rocha[13]	Yes	No	No	Yes	No
BĂZĂR et al.[15]	Yes	No	Yes	Yes	No
Lee & Zheng[16]	Yes	No	Yes	No	No
Zhao et al.[18]	Yes	No	No	Yes	No
Apache Software Foundation[53]	Yes	No	Yes	No	No
Papotti & Torlone[54]	Yes	No	Yes	No	No
Migration between NoSql Database and NoSql Database					
Scavuzzo et al.[20]	No	No	Yes	No	No
Alomari et al.[23]	No	Yes	Yes	Yes	Yes
Michel et al.[55]	Yes	No	No	No	Yes
Chung et al.[17]	Yes	No	Yes	No	No
Atzeni et al.[21]	No	Yes	Yes	Yes	Yes
Scherzinger et al.[57]	No	No	No	No	Yes
Saur et al.[24]	No	Yes	Yes	Yes	Yes
Bansel[58]	No	No	No	No	Yes
Metamodel Strategy[6]					
Atzeni et al.[29]	No	Yes	Yes	Yes	Yes
Beslic et al.[8]	Yes	No	No	No	Yes
Bugiotti et al.[5]	No	Yes	Yes	Yes	Yes

[5] In this context other types means all kind of data stores not yet mentioned, like Graph databases.
[6] The work in this dissertation is inserted in this section.

Bugiotti et al.[28]	No	Yes	Yes	Yes	Yes
Atzeni et al.[59]	No	Yes	Yes	Yes	Yes

2.6 Summary

As seen in 2.5.2, the problem of data migration between NoSql Databases is already being studied by different authors. Most of the attention was given in the migration of data between relational databases and NoSql Databases. Which makes sense, keeping in mind, which the NoSql development are recent and born to complement the relational databases weak points. From this perspective data migration from legacy system's to the new platform tends to be prioritized.

With the evolution of this new technology over the time, it was noted the need for migration data between NoSql Databases and some works began to appear in this regard. The vast majority of these works was focused on executing the data migration among databases of the same type. Other works focus on the process of migration of multiple versions of the same schema in the same database. The most promising technique came from the NoAM application [59] with creates a layer which represents the various models where the data can be represented by the most NoSql Databases. In this work, this representation is used to better take advantage of the main features offered by each data model available, in the same sense which traditional modeling try to take advantage of the RDBMS.

What we see in common between the works studied is mainly the difficulty in migrating data between different models of NoSql Databases. As well as the complexity of metadata extraction and manipulation using metamodels. In the Chapter 3 a new proposed methodology will be studied to make migrating scalable without the need to write new codes.

3 Data Migration Structure Design

This chapter describes the proposed work called Data Migration Architecture which is a framework designed to enable data copy between different NoSql Database models. The work uses the JSON data format as an intermediary representation to allow the proposed framework to deal with the NoSql heterogeneity of models presented in 2.4. The only assumption made about the data is that can be expressed in JSON format and the NoSql Database support CRUD operations. An introduction to the metamodel is stated in 3.1.1 and a detailed explanation about the framework starts in the Section 3.1 and how the data is represented in de designed framework are shown in 3.1.3.

3.1 Introduction to Proposed Architecture

The work presented in this dissertation is based on the transformation and rewritten of the data from of any NoSql model to another. The data must be capable to being expressed in a generic, comprehensive and standardized format.

The proposed method is shown in Figure 5

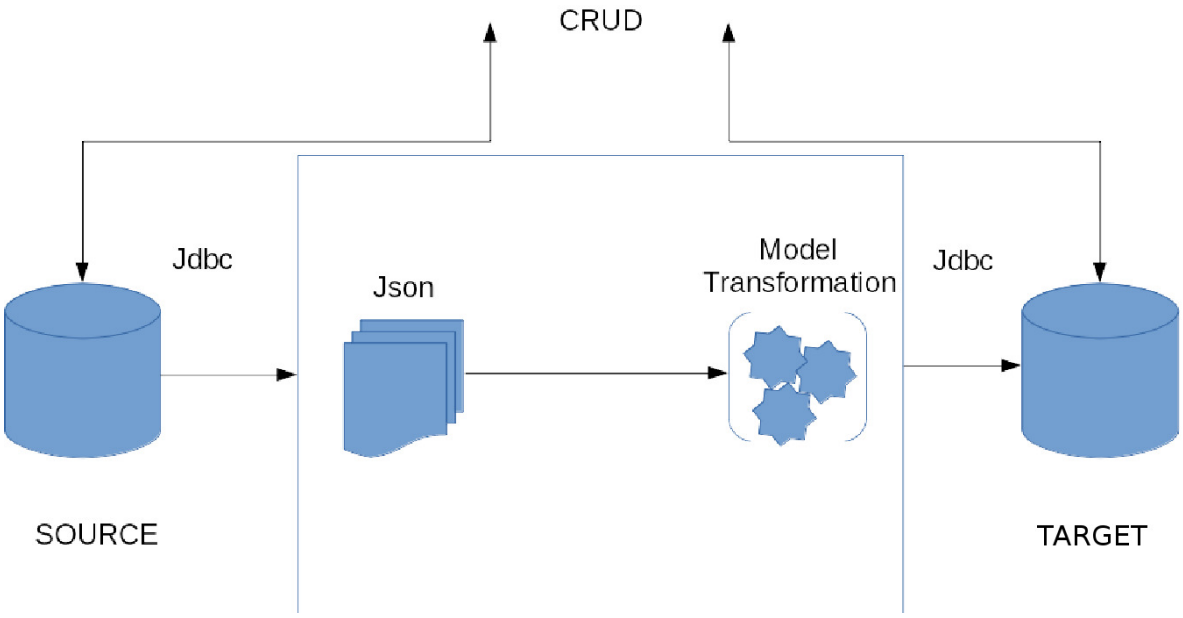

Figure 5 – Proposed architecture.

In this architecture, the Source and the target databases are connected to the migration system via JDBC drivers. Then using the basic data access methods the data are extracted and converted into a JSON object or file. The Read method is used in this step. The system then read the JSON file and then divide them into their basic components.

Each individual component is then analyzed and labeled. Then the system converts the JSON object received into the destination format through the built-in metamodel. Finally using the set methods, the data are inserted into the target database.

The translation process and mapping of the data will be explained in this chapter and consists of three main steps:

- The data is received in JSON format
- Their structure are analyzed to retrieve the desired properties
- A direct translator will translate the data into a requested NoSql model.

In this study the metamodel which was built doesn't focus on the characteristics of a specific database, on the contrary, it uses the common architecture which the NoSql databases shares. Specifically, the different data models described in Section 2.4 are used. Using this technique when a translator for a specific model are constructed they can communicate with all other models which are already created.

The idea is to create an extension of the basic methods, based on CRUD, which can extract the data from the source database and to export the data in JSON format. After the model transformation, the data can be inserted into the target database.

The datasets provided by the architecture are then translated and combined to get the specific data format.

The four NoSql Databases models, or families, used in this research share two basic common characteristics which will be explored:

1. the smallest possible representation of a data structure is the same, a set composed of a key and a value which can be handled and managed individually.
2. there is no need to follow a rigid structure, and each NoSql Databases has a minimum list of components or functions which allow direct access to data hiding their internal structure.

3.1.1 Metamodel

According to [60] a metamodel is a data structure that enables to represent the components of a conceptual model, process, or system. It is used to describe several other models and elements, including how they are organized, related and limited. The use of this concept brings advantages such as more efficient architecture and high scalability and good flexibility. For example, different kinds of schemas and the relationships between them can be easily mapped and understood, allowing in a final stage a transformation between this schemas.

As briefly discussed in 2.5.1, the method which will be studied and implemented in this dissertation is the creation of a metamodel intended to act as a translator between different NoSql models.

With the development of one metamodel, interesting characteristics can be exploited. The more generic the model is, fewer translations are necessary and more NoSql Databases can be embedded in these models. As a consequence the developers do not need to have a deep knowledge of supported databases. Is necessary only to know in a generic way what the model which are utilized by the destination NoSql Database and their basic commands. Thus making easier to carry out tests or create a proof of concept into a new database.

The metamodel created in this work is focused on data migration.

3.1.2 Data representation into the Metamodel

The process of translation the data between the initial JSON structure and a desired final format is done in distinct stages, and each stage has a predefined input and output data model.

A stream of characters is received and the parsing process produces a series of events. For each event (see 2.4.1) discovered into the file the second step produces a representation in the desired format, as shown in section 3.1.3. The process is repeated until an entire object inside a file is processed. The representation of the JSON object processed is exemplified in listing 3.3.

The final process is the transformation of the data structures mapped in the previous steps into a specific NoSql representation. This transformation is based on the information available in the representation created into the metamodel. Other pieces of information like comments and content format are not necessary for the process.

The process is illustrated in the figure 6.

The idea is to represent the data in the basic components that can be extracted from a JSON object like their value, type, indentation and relation with the other data present in the same object. Thus giving a uniform and common representation of each data here called a JSON slice. Using this technique it become simple to write translators using common Get's and Set's operations provided by the framework.

The translation is done by receiving and reading a JSON object and then slicing these object in their minimal logical components, the data itself, and storing along whit the data the characteristics which will be important in the next step of the process.

This characteristic are:

- Level or indentation of the data making possible to understand the hierarchy of the

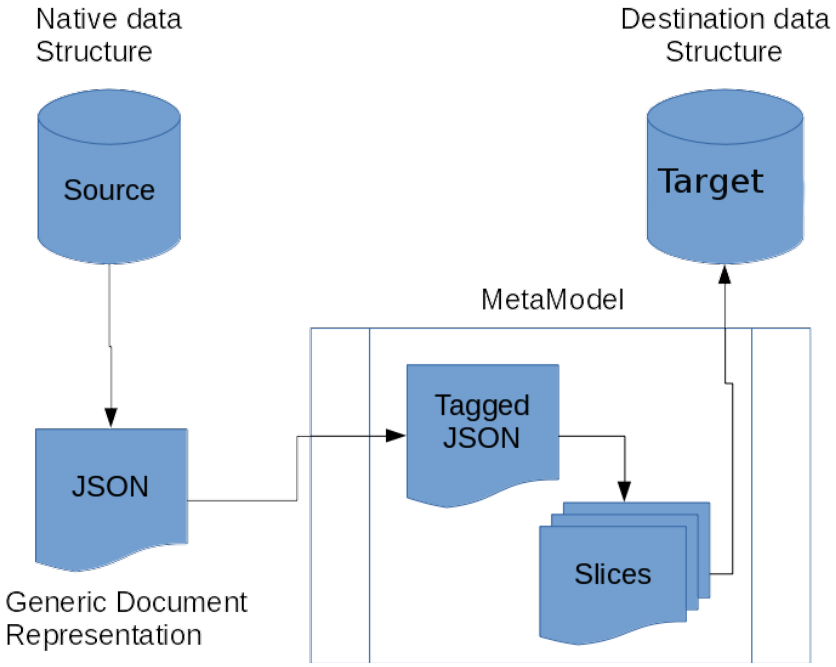

Figure 6 – Process Overview.

data inside the JSON object

This indentation is significant because can be mapped similarly to a relation between the data so is possible to infer an relationship based on this information.

- Label of the data identifies an event inside a slice (as shown in 2.4.1)

  The event means the role of the value within the data structure.

- The father slice

  Inside a JSON an hierarchy can be achieved by nesting other objects or arrays of values. The father slice is the one which is immediately superior into the hierarchy. If the slice is the first obtained, then they will be the object identifier. Is important to know all the hierarchies inside an object to be able to reconstruct the data later.

3.1.3 Intermediate Mapping

Given a JSON file equals to the demonstrated in listing 3.1, these object are first tagged to allow the identification off all this events, this will give a representation of the structure of the document, presented in listing 3.2.

```
{ "Person":
  {"firstName": "John", "lastName": "Smith", "age": 25,
  "phoneNumber": [
        { "type": "home", "number": "212 555-1234" },
        { "type": "fax", "number": "646 555-4567" }
        ]
   }
 }
}
```

Listing 3.1 – Excert of Person Object (JSON notation)

```
{START_OBJECT
"Person"KEY_NAME:
  {START_OBJECT "firstName"KEY_NAME: "John"VALUE_STRING, "
     lastName"KEY_NAME: "Smith"VALUE_STRING, "age"KEY_NAME: 25
     VALUE_NUMBER,
   "phoneNumber"KEY_NAME : [START_ARRAY
      {START_OBJECT "type"KEY_NAME: "home"VALUE_STRING, "number"
         KEY_NAME: "212 555-1234"VALUE_STRING }END_OBJECT,
      {START_OBJECT "type"KEY_NAME: "fax"VALUE_STRING, "number"
         KEY_NAME: "646 555-4567"VALUE_STRING }END_OBJECT
   ]END_ARRAY
  }END_OBJECT
}END_OBJECT
```

Listing 3.2 – Person Object (Tagged)

A Slice is a flatted object with additional properties. A slice to be created must be composed of the following properties:

ObjectId a unique identifier of this slice of the object.

DataValue the data properly.

Level the indentation level of the slice into a object

Label the tag or event associated.

FatherObj the ObjectId of the father's slice.

The unique identifier is created automatically and is a simple numerical sequence added to each new slice created. The tag was discovered in the tagging process, executed in the step earlier. The associations are based on the hierarchy discovered, so each time

which an event like START_ARRAY or START_OBJECT are found, the early level was stored as a father and the level are added to a new one. If an event like END_ARRAY or END_OBJECT are found the inverse process is performed. These processes in Algorithm 1 are a representation of the main steps described.

Algorithm 1 Slicing a JSON object

```
1: while parser ← JSON do
2:     ++ObjectId
3:     Event_i ← JSON.Event
4:     if Event_i is START_ARRAY OR START_OBJECT then
5:         ++Level
6:     else if Event_i is END_OBJECT or END_ARRAT then
7:         - -Level
8:     else
9:         Store Data Value
10:    end if
11: end while
```

To demonstrate more clearly the process let's view an excerpt exemplifying how the property "phoneNumber" will be mapped in listing 3.3.. The example was extracted from JSON listed in 3.1.

```
...

ObjectId  : 8
DataValue : phoneNumber
Level     : 1
Label     : KEY_NAME
FatherObj : 1

ObjectId  : 9
DataValue : null
Level     : 2
Label     : START_ARRAY
FatherObj : 8

ObjectId  : 10
DataValue : null
Level     : 3
Label     : START_OBJECT
FatherObj : 9

...

ObjectId  : 21
DataValue : null
Level     : 3
Label     : END_OBJECT;
FatherObj : 16

ObjectId  : 22
DataValue : null
Level     : 2
Label     : END_ARRAY;
FatherObj : 9

ObjectId  : 23
DataValue : null
Level     : 1
Label     : END_OBJECT;
FatherObj : 1
```

Listing 3.3 – Excerpt of the Slices

A complete view of the JSON object sliced can be seen on Appendix A.

3.2 Data Translation

Because NoSql Database are schemaless, the semantics of the source data is known by the programmer's, not by the DBMS. Since the process to extract data from a source database and represent in the JSON format are done by user programs, this work starts from the premise that the data is already available. In this section, a complete translation from the JSON intermediary format and all the final NoSql models will be presented. The input data of the process always will be in the JSON format. The object listed in 2.1 will be the reference for all the concrete examples showed in the next sections.

After an object was received and completely sliced, see 3.1.3, they can be translated to the final format desired. These formats are described in section 2.4.

The definitions used in this work are described below. These settings are used by all translators created.

- $Class_{name}$ The class name defines the identifier of a class[1] from the object being migrated, meaning that all the objects or arrays that follow belongs to the same kind. In the Document Store model, the class name is called Collections. In the Graph model the class name is the main node.
- Key_{main} For each model for which it is desired to transform the data, a Key_{main} is created based on the rules described in the metamodel created.
- Value Is the data properly indexed by the Key_{main}. The Value is discovered based in the rules described in the metamodel created.

During the processing of the data the first step is to identify the $Class_{name}$ of the objects. This attribute is the first KEY_NAME found inside the slices, this value must by in the level 1 (one) of the slice and be followed by a START_OBJECT.

Algorithm 2 Finding the $Class_{name}$

```
1: if (Event_i = KEY_NAME) and (Level = 1) and (NEXT.Event_i = START_OBJECT) then
2:     Class_name = Slice.value
3: end if
```

Following the rule described the class identified in the example is called "Person" and this values will be used during the remainder of this dissertation.

JsonReWrite exposes the methods listed in table 11, each of then uses the Get's methods exposed by *jsonSlice* to understand ans then rewrite the JSON object in a desired format.

[1] In the context of this work class is used as a noun, thus meaning things regarded as forming a group by reason of common attributes, characteristic or qualities

Table 11 – *JsonReWrite* Methods

Database Type	*Method name*	*reference*
Key Values		
	kvpo()	3.3.1
	kvpf()	3.3.2
	khpf()	3.3.3
	kvpfMM()	3.3.4
	kvpav()	3.3.5
Column Store Databases		
	Column()	3.4.1
	Scolumn()	3.4.2
	Columnfamily()	3.4.3
	SColumnfamily()	3.4.4
Document Store Databases		
	dpo()	3.5.1
	ipo()	3.5.2
	cpo()	3.5.3
Graph Databases		
	graph()	3.6

3.3 Key Value

For the Key Value databases the translations mapped will be shown in a subsection for each different proposed model. These models are explained in section 2.4.2

3.3.1 Key-value per object

The slice are analyzed to construct the Main Key Key_{main} of this object. The Key_{main} is formed by the object $Class_{name}$ plus the first VALUE_STRING found in the JSON

In our example the Key_{main} formed is : "Person:John"

The Value is created using all the other values of the object are concatenated in order, preserving the indentation and the structure.

Then the representation will generate a sequence of key/vales like the one demonstrated in the table 12.

3.3.2 Key-value per field

In the Key-value per field, kvpf, representation the Key_{main} are formed by the object $Class_{name}$ plus the KEY_NAME, and this process occurs for each KEY_NAME found in the JSON object. The value, for each Key_{main} discovered, is the data itself associated at the KEY_NAME. If the data is and Array or other Object all the values are

Table 12 – Key-value per object - kvpo()

	Main Key	Value
Rule	Key_{main}	1: **for all** Slice.value **do** 2: $Value = Value$ + Slice.value 3: **end for**
Output	Person:John	{"firstName":"John", "lastName": "Smith", "age": 25, "phoneNumber": [{ "type": "home", "number": "212 555-1234" }, { "type": "fax","number": "646 555-4567" }]}

concatenated in order until the end of the Array or Object.

The key/value sequence generated by this method is demonstrated in table 13.

Table 13 – Key-value per field - kvpf()

	Key	Value
Rule	Key_{main} + "/" + Slice.KEY_NAME	1: **for all** Slice.KEY_NAME **do** 2: **if** $Value = (Array$ **or** $Object)$ **then** 3: **for all** Slice.value **do** 4: $Value = Value$ + Slice.value 5: **end for** 6: **else** 7: $Value$ = Slice.value 8: **end if** 9: **end for**
Output	Person:John/firstName Person:John/lastName Person:John/age Person:John/phoneNumber	John Smith 25 { "type": "home", "number": "212 555-1234" }, { "type": "fax","number": "646 555-4567" }

3.3.3 Key-hash per field

The Key follow the same rule stated in 3.3.1. The same Key_{main} has several vales, each one constituted by the KEY_NAME found plus the value associated. If the value is an array or other object, the value is the concatenation of all elements of the array or object.

Then the representation will generate a sequence of key/vales like the one demonstrated in the table 14.

Table 14 – Key-hash per field - khpf()

	Key	Value
Rule	Key_{main}	1: **for all** Slice.KEY_NAME **do** 2: **if** $Value$ = ($Array$ **or** $Object$) **then** 3: **for all** Slice.value **do** 4: $Value = Value$ + Slice.value 5: **end for** 6: $Value$ = Slices.KEY_NAME + ":" + $Value$ 7: **end if** 8: $Value$ = Slices.KEY_NAME + ":" + Slice.Value 9: **end for**
Output	Person:John	firstName:John lastName:Smith age:25 phoneNumber:["type": "home", "number": "212 555-1234" , "type": "fax","number": "646 555-4567"]

3.3.4 Key-value per field Major/minor

To format the data into this model the Key is composed by the Key_{main} plus /-/ plus each KEY_NAME found in the object. The values are formed by the KEY_VALUE associated to the KEY_NAME. If the value is an array or other object, the value is the concatenation of all elements of the array or object.

Then the representation will generate a sequence of key/vales like the one shown in the table 15.

Table 15 – Key-value per field Major/minor - kvpfMM()

	Key	
Rule	MainKey + "/-/" + Slices.KEY_NAME	1: **for all** Slice.KEY_NAME **do** 2: **if** $Value$ = ($Array$ **or** $Object$) **then** 3: **for all** Slice.value **do** 4: $Value = Value$ + Slice.value 5: **end for** 6: $Value$ = Slices.KEY_NAME + ":" + $Value$ 7: **end if** 8: $Value$ = Slices.KEY_NAME + ":" + Slice.Value 9: **end for**
Output	Person/John/-/firstName Person/John/-/lastName Person/John/-/age Person/John/-/phoneNumber	John Smith 25 "type": "home", "number": "212 555-1234" , "type": "fax","number": "646 555-4567"

3.3.5 Key-value per atomic value

In this format the values are formed by each of the individual KEY_VALUE found. The Key is composed by the Key_{main} plus /-/ plus all the path until the KEY_NAME before the value. In the case of the value is into an array or another object a sequential number is added in the key o maintain the uniqueness.

Then the representation will generate a sequence of key/vales like the one demonstrated in the table 16.

Table 16 – Key-value per atomic value - kvpav()

	Key	Value
Rule	1: **for all** Slice.KEY_VALUE **do** 2: Key = MainKey + "/-/" + Slices.KEY_NAME 3: **if** $Value = (Array$ **or** $Object)$ **then** 4: **for all** $Slice_i$ **do** 5: $Key = Key$ + "/" + $Slice_i$ + Slice.KEY_NAME 6: **end for** 7: **end if** 8: **end for**	Slices.KEY_NAME.Value
Output	Person/John/-/firstName Person/John/-/lastName Person/John/-/age Person/John/-/phoneNumber/0/type Person/John/-/phoneNumber/0/number Person/John/-/phoneNumber/1/type Person/John/-/phoneNumber/1/number	John Smith 25 home 212 555-1234 fax 646 555-4567

3.4 Column Store Databases

In the column store databases the identification of the columns are fundamental This columns then are be grouped into the families, Super Column, Column Family and Super Column Family. The methods created are explained in the next sections.

3.4.1 Column

The column name will be each individual KEY_NAME found and the values are formed by each of the individual KEY_VALUE found. In case of the value is into an array or other object the columns name will be composed by the KEY_NAME o the father plus the final KEY_NAME found. That way no group is created, and the columns are stored individually.

In the table bellow the representation of the columns and the values are represented.

3.4.2 Super Column

The Super Column model is a variation of the format described in 17. The main process is the same, column name will be each KEY_NAME found, and the values are formed by each of the individual KEY_VALUE found. The difference is in the case of the

Table 17 – Column

	Column	Value
Rule	1: **for all** Slice.KEY_NAME **do** 2: **if** Slice.hasFater = **true then** 3: $Key = Slice_f$.KEY_NAME + "/" + Slice.KEY_NAME 4: **else** 5: Key = Slice.KEY_NAME 6: **end if** 7: **end for**	Slice.KEY_VALUE
Output	firstName lastName age phoneNumber/type phoneNumber/number phoneNumber/type phoneNumber/number	John Smith 25 home 212 555-1234 fax 646 555-4567

value is into an array or another object, so the KEY_NAME of the father slice is used as a Super Column name, with the other KEY_NAMEs serving as a column name.

An example of this representation can be seen in table 18

Table 18 – Super Column

	Super Column	Column	Value
Rule	$Slice_f$.KEY_NAME	KEY_NAME	KEY_VALUE
Output	phoneNumber	firstName	John
		lastName	Smith
		age	25
		type	home
		number	212 555-1234
		type	fax
		number	646 555-4567

3.4.3 Column Family

The Column Family group the columns based in a Row Key. The Row Key is set by the first VALUE_STRING found in the JSON. In this representation, the Row Key indicates that all the values of the columns belong to the same Row Key, much similar to a Row in the relational model. The columns follow the rules of creation of a Super Column 3.4.2

A example of this representation is shown in table 19

Table 19 – Column Family

	Row Key	Super Column	Column	Value
Rule	$Class_{name}$	$Slice_f$.KEY_NAME	KEY_NAME	KEY_VALUE
Output	John	phoneNumber	firstName	John
			lastName	Smith
			age	25
			type	home
			number	212 555-1234
			type	fax
			number	646 555-4567

3.4.4 Super Column Family

The Super Column Family remember a table in a relational model, with a Row Key grouping columns that are correlated. The Row Key is set by the object $Class_{name}$, which plays the role of a table name. The columns follow the rules of creation of a Super Column 3.4.2

Table 20 – Super Column Family

	Column Family	Super Column	Column	Value
Rule	$Class_{name}$	$Slice_f$.KEY_NAME	KEY_NAME	KEY_VALUE
Output	Person	phoneNumber	firstName	John
			lastName	Smith
			age	25
			type	home
			number	212 555-1234
			type	fax
			number	646 555-4567

3.5 Document Store Databases

The Document Stores uses a concept when each data (document) are grouped into Collections that can be compared to a table in a relational model. Along with the collection must be created a key called document id. In the next sections the creation rules of the Collections, Document id and documents for the models are explained.

3.5.1 Document per object

The document per object model format resembles the Key-value per object model, kvpo. The main difference is the fact that Key_{main} is split in two, the class name will act as a collection name and the first VALUE_STRING found will be the "Document id". In our example the collection name discovered is "Person" and the Document id is "John".

The document is created using all the other values of the object are concatenated in order, preserving the indentation and the structure.

Then the representation will generate a sequence of Collection/Document id/document like the one demonstrated in the table 12.

Table 21 – Document per object - dpo()

	Collection	Document id	Value
Rule	Class name	VALUE_STRING.first	1: **for all** Slice.value **do** 2: $Value = Value$ + Slice.value 3: **end for**
Output	Person	John	{"firstName":"John", "lastName": "Smith", "age": 25, "phoneNumber": { "type": "home", "number": "212 555-1234" }, { "type": "fax","number": "646 555-4567" }

3.5.2 Item per object

This model is similar to the Key-value per field 3.3.3, the class name will be the Table name and the data will be constituted by the KEY_NAME found plus the value associated. To distinguish each collection within the same table one ID is generated for each document inside a table, this ID is composed by the first VALUE_STRING. In case of the value is an array or other object, the value is the concatenation of all elements of the array or object.

Then the representation will generate a sequence of Table/Documents/Values like the one demonstrated in the table 22.

Table 22 – item per object - ipo()

	Table	Documents	Value
Rule	Class name	KEY_NAME	1: **for all** Slice.KEY_NAME **do** 2: $Value = Value$ + Slice.value 3: **end for**
Output	Person	_id	John
		firstName	John
		lastName	Smith
		age	25
		phoneNumber	{ "type": "home", "number": "212 555-1234" }, { "type": "fax","number": "646 555-4567" }

3.5.3 Cell per object

The Cell per object model is very similar to the kvpo3.3.1. The table name is formed by the object $Class_{name}$

One ID is create based on the first VALUE_STRING found in the JSON.

The Value is created using all the other values of the object are concatenated in order, preserving the indentation and the structure.

Then the representation will generate a sequence of table/id/value like the one demonstrated in the table 23.

Table 23 – Cell per object - cpo()

	Table	ID	Value
Rule	Class name	VALUE_STRING.First	1: **for all** Slice.value **do** 2: $Value = Value$ + Slice.value 3: **end for**
Output	Person	John	John {"firstName":"John", "lastName": "Smith", "age": 25, "phoneNumber": [{ "type": "home", "number": "212 555-1234" }, { "type": "fax","number": "646 555-4567" }]}

3.6 Graph Databases

The Graph databases use a concept of the main node that represents the anchor data. The other data is represented in nodes called leafs. The relation between the main node and the leafs, and between the leafs itself are important too. In this study, the relations between the main node and the leaf nodes are always created with the name "HAVE"

In the graph model are similar in his construction to the Key-hash per field 3.3.3

The Main Node is formed by the object $Class_{name}$ plus the first VALUE_STRING found, "Person:John". This is the same process used to form the Key_{main}. The leaf nodes are constituted by each KEY_NAME found plus the value associated. In the case of the value is an array or another object, the value is the concatenation of all elements of the array or object.

Then the representation will generate a sequence of Main node/Leaf Nodes/Value like the one demonstrated in the table 24.

Table 24 – Graph - graph()

	Main Node	Leaf Node	Value
Rule	Key_{main}	Slices.KEY_NAME	1: **for all** Slice.KEY_NAME **do** 2: **if** $Value$ = ($Array$ **or** $Object$) **then** 3: **for all** Slice.value **do** 4: $Value = Value$ + Slice.value 5: **end for** 6: $Value$ = Slices.KEY_NAME + ":" + $Value$ 7: **end if** 8: $Value$ = Slices.KEY_NAME + ":" + Slice.Value 9: **end for**
Output	Person:John	firstName lastName age phoneNumber	John Smith 25 { "type": "home", "number": "212 555-1234" }, { "type": "fax","number": "646 555-4567" }

3.7 Summary

In this chapter, the architecture of the data translation was discussed. First, a brief discussion on the types of migration which are supported by the framework is performed, then the architecture was demonstrated, and the JSON format explained, along with the databases used in work. A description of the metamodel used and how the data is extracted and stored before being used. The chapter end with an overview of the process.

4 Evaluation

In this chapter, experiments were performed to verify the validity of the designed NoSql data migration framework.

Several evaluations of the process of the translation of the data represented in an intermediary format, JSON, into one of the models described in 2.4will be reported. To evaluate the migration architecture a dataset from City of Chicago Data Portal [1] is used. Specifically data about "Food Inspections" [2] are used. The dataset describes inspections of restaurants and other food establishments in Chicago from January 1, 2010 to December 1, 2016. The information can be accessed using city API for download the dataset and the statistics. The test data were chosen because they are freely available on the Internet and are in the public domain.

One of the representatives of each data model described in 2.4 were chosen for evaluation. The reasons behind the choice of these databases were the presence of the software in the studies listed in section 2.5.2 allowing the results to be replicated more easily. If there are more than one database to the same model was chosen the software which showed the best documentation. Application Programming Interfaces (APIs) provided by each selected NoSql Databases are used to perform the basic operation.

The chosen software were listed in Table 25:

Table 25 – NoSql Databases choosen

Document Store	Column store	Key Value	Graph Database
mongo DB	Hbase	Oracle Nosql	Neo4j

4.1 Implementation

The implementation was coded in java language version "1.8.0_111" [61]. The metamodel created exposes some methods that manipulate the data already sliced and stored in an ArrayList as demonstrated in 3.3. These methods are basic Sets and Gets for the attributes, and on which translations are built. The translators uses a series of rules that were discussed in 3.2.

Figure 7 shows the UML class diagram of the metamodel proposed. This model is composed of one class called *GenericJsonParser*, this class has the responsibility to manage the translation process and extract the characteristics of the JSON object, building

[1] https://data.cityofchicago.org/
[2] https://data.cityofchicago.org/Health-Human-Services/Food-Inspections/4ijn-s7e5

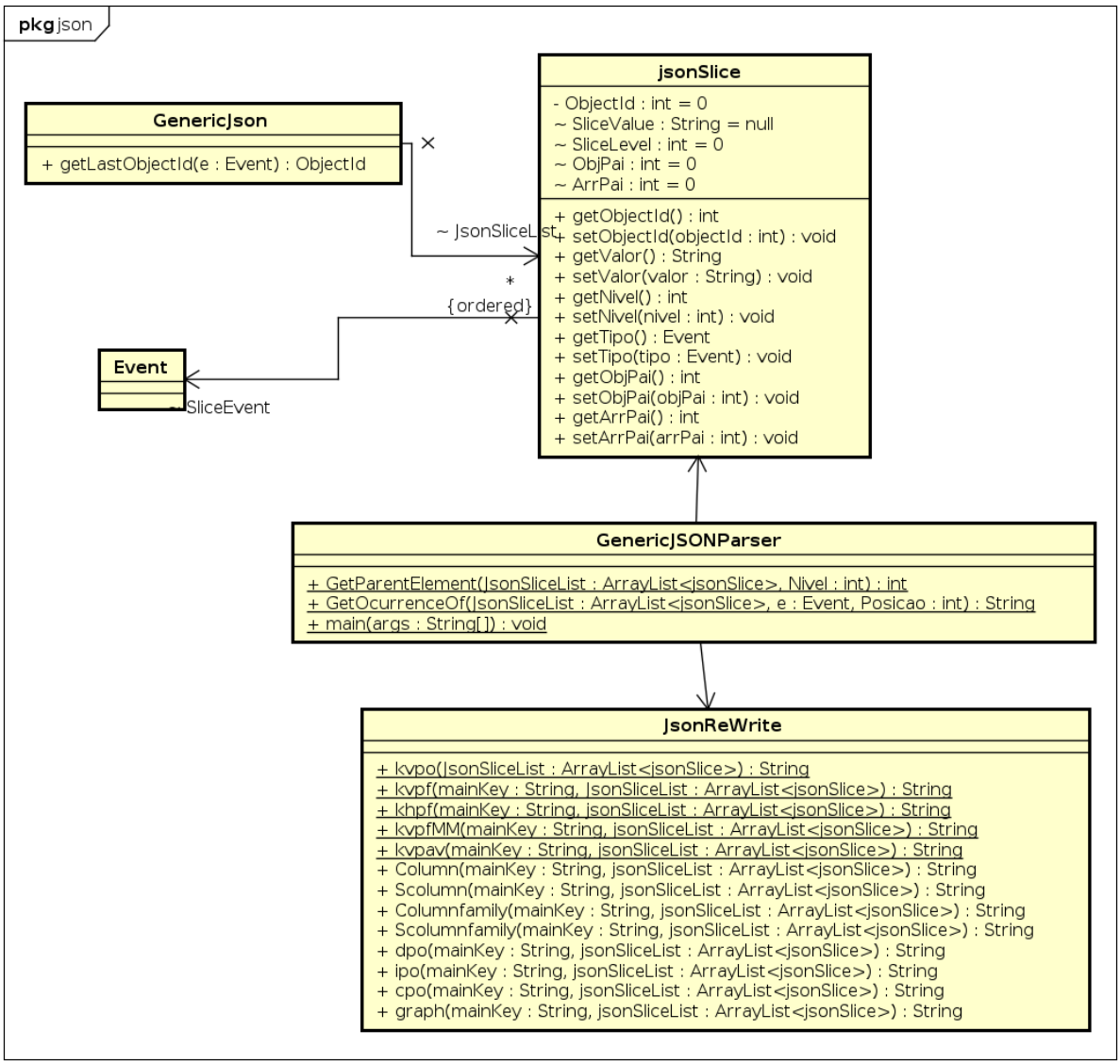

Figure 7 – Metamodel developed showing the main classes and methods.

the list of slices. The slices are represented in the *jsonSlice* class. This class has the basic characteristics of an slice see 3.1.2 together with the get's and set's methods. A class called *JsonReWrite* is used to translate the data into the new format needed. The *Event* class maps the events described in 2.4.1. the *GenericJson* is the representation of a JSON object.

4.2 Case study

In this section, we present a case study involving the methods of data translation. The class of the object is called "Inspections". The file contains a total of 139.535 objects. Each object are composed by 23 distinct fields an 1 array. The array is composed of 5 distinct fields.

As a guideline, it is possible to infer, based on the rules exposed in Subsection 3.3.1, how many Key/Value pairs will be generated by the following formula.

$Output_{Total} = (Fields_{Total} + Array_{Total}) \times Objects_{Total}$

So for example in the scenario tested with a total of 139.535 objects.

Key-value per Object: 1 × 139.535 = 139.535

Key-hash per field: (23 + 1) * 139.535 = 3.348.840

In some cases, the objects inside the array must be added to the total of Fields.

$Output_{Total} = (Fields_{Total} + Array_{Fields}) \times Objects_{Total}$

An example of this rule is:

Super Column Family: (23 + 5) × 139.535 = 3.906.980

A summary can be seen in table 26. A complete view of one JSON with all the fields and array described for one object can be seen on Appendix B.

Table 26 – Characteristics of the original Inspection JSON object generated

	Class	Fields	Array	Total Objects
Original Json	1	23	1[5]	139.535

4.2.1 Key Value translations

Applying the transformations described in 3.3 is possible to note that each representation generates a different amount of Key/Values pairs.

This demonstrates the variety of granularities that each representation is capable of offering. Table 27 shows the characteristics and results of each implementation.

Table 27 – Characteristics of the Inspection JSON object in the Key Value models

Key Value	MainKey	Values	Output Pairs
Key-value per Object	1	1	139.535
Key-value per field	24	24	3.348.840
Key-hash per field	1	24	3.348.840
Key-value per field Major/minor	24	24	3.348.840
Key-value per atomic value	28	28	3.906.980

The tables 28-32 show excerpts of the resulting translations for the Key-Value models.

Table 28 – Excerpt of a Key-value per Object Key/Value pair

	Key-value per Object
MainKey	Inspections:1
Values	{"sid" : 1,"id" : "A657EC01-A6FD-4C58-A907-6B3D82696B19","position" : 1,"created_at" : 1482409144,"created_meta" : "386464","updated_at" : 1482409144,"meta" : "386464", ... }
Output Pairs	139.535

Table 29 – Excerpt of a Key-value per Field Key/Value pair

	Key-value per Field
MainKeys	Inspections:1/id Inspections:1/position Inspections:1/created_at Inspections:1/created_meta ...
Values	"A657EC01-A6FD-4C58-A907-6B3D82696B19" 1 1482409144 "386464" ...
Output Pairs	3.348.840

Table 30 – Excerpt of a Key-hash per field Key/Value pair

	Key-hash per field
MainKeys	Inspections:1 Inspections:1 Inspections:1 Inspections:1 ...
Values	id:"A657EC01-A6FD-4C58-A907-6B3D82696B19" position:1 created_at:1482409144 created_meta: "386464" ...
Output Pairs	3.348.840

Table 31 – Excerpt of a Key-value per field Major/minor Key/Value pair

	Key-value per Field Major/minor
MainKeys	Inspections/1/-/id Inspections/1/-/position Inspections/1/-/created_at Inspections/1/-/created_meta ...
Values	"A657EC01-A6FD-4C58-A907-6B3D82696B19" 1 1482409144 "386464" ...
Output Pairs	3.348.840

Table 32 – Excerpt of a Key-value per atomic value Key/Value pair

	Key-value per per atomic value
MainKeys	Inspections/1/-/id Inspections/1/-/position Inspections/1/-/created_at Inspections/1/-/created_meta ...
Values	"A657EC01-A6FD-4C58-A907-6B3D82696B19" 1 1482409144 "386464" ...
Output Pairs	3.906.980

4.2.2 Column Stores translations

In the Column Stores, the total amount of columns generated is the same. The difference is in the way of the columns are grouped, thus forming the various possible families.

Table 33 summarizes the possibilities.

Table 33 – Characteristics of the Inspection JSON object in the Column Store models

Column Stores	Row Key Family	Super Column	Column	Value	Output Columns
Column	n/a	n/a	28	28	3.906.980
Super Column	n/a	1	28	28	3.906.980
Column Family	1	1	28	28	3.906.980
Super Column Family	1	1	28	28	3.906.980

The tables 34-37 show excerpts of the resulting translations for the Column Store models.

Table 34 – Excerpt of a Column model Column/Value pair

	Column
Columns	id position created_at created_meta ...
Values	"A657EC01-A6FD-4C58-A907-6B3D82696B19" 1 1482409144 "386464" ...
Output Pairs	3.906.980

Table 35 – Excerpt of a Super Column model Column/Value pair

	Super Column
Columns	id position created_at created_meta ...
Values	"A657EC01-A6FD-4C58-A907-6B3D82696B19" 1 1482409144 "386464" ...
Super Column	"location"
Output Pairs	3.906.980

Table 36 – Excerpt of a Column Family Column/Value pair

	Column Family
Columns	id position created_at created_meta ...
Values	"A657EC01-A6FD-4C58-A907-6B3D82696B19" 1 1482409144 "386464" ...
Row Key	"1"
Super Column	"location"
Output Pairs	3.906.980

Table 37 – Excerpt of a Super Column Family Column/Value pair

	Super Column Family
Columns	id position created_at created_meta ...
Values	"A657EC01-A6FD-4C58-A907-6B3D82696B19" 1 1482409144 "386464" ...
Column Family	"Inspections"
Super Column	"location"
Output Pairs	3.906.980

4.2.3 Document Stores translations

In the Document Stores, the process generates a variety of results. Again the choice of the key that will compose the document has a direct consequence in the total of final values that are generated.

In the table 38 a summary of the values generated is shown.

The tables 39-41 show excerpts of the resulting translations for the Document Store models.

Table 38 – Characteristics of the Inspection JSON object in the Document Store models

Document Stores	Collection/Table	Document / ID	Value	Output Values
Document per object	1	1	1	139.535
Item per object	1	24	24	3.348.840
Cell per object	1	1	1	139.535

Table 39 – Excerpt of a Document per object Collection/Document pair

	Document per object
Collection	"Inspections"
Document id	1
Value	{"sid" : 1,"id" : "A657EC01-A6FD-4C58-A907-6B3D82696B19","position" : 1,"created_at" : 1482409144,"created_meta" : "386464","updated_at" : 1482409144,"meta" : "386464", ... }
Output Pairs	139.535

Table 40 – Excerpt of a Item per object Collection/Document pair

	item per object
Table	"Inspections"
Document	_id id position created_at created_meta ...
Value	1 "A657EC01-A6FD-4C58-A907-6B3D82696B19" 1 1482409144 "386464" ...
Output Pairs	3.348.840

Table 41 – Excerpt of a Cell per object table/id/value pair

	item per object
Table	"Inspections"
ID	1
Value	{"sid" : 1,"id" : "A657EC01-A6FD-4C58-A907-6B3D82696B19","position" : 1,"created_at" : 1482409144,"created_meta" : "386464","updated_at" : 1482409144,"meta" : "386464", ... }
Output Pairs	139.535

4.2.4 Graph translation

For the Graph database, the process has generated one main node, the Inspection class, and one leaf node for each field or array in the original file. The values are then inserted into each leaf node.

Table 42 demonstrates the results.

Table 42 – Characteristics of the Inspection JSON object in the Graph model

Graph Databases	Main Node	Leaf Node	Value	Output Values
Graph	1	24	24	3.348.840

The table 43 show excerpts of the resulting translations for the Graph model.

Table 43 – Excerpt of a Graph Main node/Leaf Nodes/Value pair

	Graph
Main node	"Inspections:1"
Leaf Nodes	id position created_at created_meta ...
Value	"A657EC01-A6FD-4C58-A907-6B3D82696B19" 1 1482409144 "386464" ...
Output Pairs	3.348.840

4.3 Summary

The experiments show that it is possible to migrate data between NoSql Database. Until now the works has focused on two main approaches. Some papers focus on data representation but do not perform the data migration. Other works focus on data migration, but as they do not pay attention about the data representation end up migrating the data between specific databases. The work developed in this dissertation unites both approaches, extracting best of each one. Modeling is used to generate generic models, so is possible attain multiple database managers using the same model. The migration can then rewrite the data for each desired manager, based on the model the manager is using.

5 Conclusions

In this chapter, we present the general conclusions of this dissertation. The synopsis of motivations and then the contributions of the approach will be presented. Finally, some subjects for future work are discussed.

5.1 Summary

In Chapter 2 an analysis of state of the art about NoSql and data migration is performed. The fact that a wide number of different NoSql Database exists and no standard among this databases are discussed. These lack of standard lead to a great difficulty in perform data migration among these databases. Some solutions are studied and their most important aspects are analyzed.

Chapter 3 provides a solution to the above problem, by proposing an original migration framework, composed of an intermediate data representation, the JSON that is sliced and represented in a metamodel. This metamodel offers basic functions over each data, allowing then to be recombined in any other format. The use of a metamodel enables data migration between NoSql Database based on the model used by the target Database.

Finally in Chapter 4 the implementation of the solution is explained. The methods ae described and how they are formed is demonstrated. The migration process has translators for 13 (thirteen) different models and as demonstrated in 3.2.

5.2 Contributions

This dissertation presented an original approach intended to permit the data migration between different NoSql Database with different models.

This work integrates two different approaches, the representation and the translation. The representation part uses the work of the [5] to be able to depict the data in the various formats desired. This representation adds to the process an abstraction layer. This layer allows several NoSql Database to be treated as long as it knows its internal data model. Thus starting from an intermediate format, it is possible to represent the data in any of the models used. The intermediate format is then sliced, and along with their metadata information, is read by the translation process.

The translation part uses the models to perform a data migration. As a result, the data can be extracted from a source, represented in an intermediate format and finally converted into a final format understood in the target database. Other works are intended

to migrate from a specific source to a specific target. In contrast this work can migrate the data from several sources to several destinations. The negative point of this approach is the fact which some specifics constructions such as indexes and partitioning of the NoSql Databases are not contemplated.

Two main contributions can be extracted from this work.

Our first contribution is the creation of slicing process with beside a generic metamodel able to understand the several models described in 2.4. This metamodel can be easily modified to understand other types of intermediary representations. Examples that can be explored are YAML and XML.

Our second contribution is the creation of rules able to translate the data from the metamodel created to other models. This rules can be extended if necessary and new rules can be created. Rules intended to data migration to and from a RDBMS can be implemented too.

5.3 Future Work

A list of future works with some considerations are shown below.

Extend the framework Due to the nature of the framework any data that can be expressed in JSON format can be migrated. Other formats like XML, YAML So an extension to work with relational databases although not trivial is possible.

Parallelism The performance of the process is not important in this phase, therefore on an extension that permits the framework to work whit Multiple data inputs Is desirable when the data volume is too large.

Performance One evaluation of the performance of the process and some possible improvements.

GUI Creation of a graphic user interface would improve user experience and can make the process more intuitive. Creation of a generic component for fast development

Appendix

APPENDIX A – Person Object completely mapped

```
ObjectId      : 1
SliceValue    : null
SliceLevel    : 1
SliceEvent    : START_OBJECT
FatherObj     : 0

ObjectId      : 2
SliceValue    : Person
SliceLevel    : 1
SliceEvent    : KEY_NAME
FatherObj     : 1

ObjectId      : 3
SliceValue    : null
SliceLevel    : 2
SliceEvent    : START_OBJECT
FatherObj     : 2

ObjectId      : 4
SliceValue    : firstName
SliceLevel    : 2
SliceEvent    : KEY_NAME
FatherObj     : 3

ObjectId      : 5
SliceValue    : John
SliceLevel    : 2
SliceEvent    : VALUE_STRING
FatherObj     : 3

ObjectId      : 6
SliceValue    : lastName
SliceLevel    : 2
SliceEvent    : KEY_NAME
FatherObj     : 3

ObjectId      : 7
SliceValue    : Smith
SliceLevel    : 2
SliceEvent    : VALUE_STRING
```

```
FatherObj     : 3

ObjectId      : 8
SliceValue    : age
SliceLevel    : 2
SliceEvent    : KEY_NAME
FatherObj     : 3

ObjectId      : 9
SliceValue    : 25
SliceLevel    : 2
SliceEvent    : VALUE_NUMBER
FatherObj     : 3

ObjectId      : 10
SliceValue    : phoneNumber
SliceLevel    : 2
SliceEvent    : KEY_NAME
FatherObj     : 3

ObjectId      : 11
SliceValue    : null
SliceLevel    : 3
SliceEvent    : START_ARRAY
FatherObj     : 10

ObjectId      : 12
SliceValue    : null
SliceLevel    : 4
SliceEvent    : START_OBJECT
FatherObj     : 11

ObjectId      : 13
SliceValue    : type
SliceLevel    : 4
SliceEvent    : KEY_NAME
FatherObj     : 12

ObjectId      : 14
SliceValue    : home
SliceLevel    : 4
SliceEvent    : VALUE_STRING
FatherObj     : 12

ObjectId      : 15
SliceValue    : number
SliceLevel    : 4
```

```
SliceEvent    : KEY_NAME
FatherObj     : 12

ObjectId      : 16
SliceValue    : 212 555-1234
SliceLevel    : 4
SliceEvent    : VALUE_STRING
FatherObj     : 12

ObjectId      : 17
SliceValue    : null
SliceLevel    : 4
SliceEvent    : END_OBJECT
FatherObj     : 12

ObjectId      : 18
SliceValue    : null
SliceLevel    : 4
SliceEvent    : START_OBJECT
FatherObj     : 11

ObjectId      : 19
SliceValue    : type
SliceLevel    : 4
SliceEvent    : KEY_NAME
FatherObj     : 18

ObjectId      : 20
SliceValue    : fax
SliceLevel    : 4
SliceEvent    : VALUE_STRING
FatherObj     : 18

ObjectId      : 21
SliceValue    : number
SliceLevel    : 4
SliceEvent    : KEY_NAME
FatherObj     : 18

ObjectId      : 22
SliceValue    : 646 555-4567
SliceLevel    : 4
SliceEvent    : VALUE_STRING
FatherObj     : 18

ObjectId      : 23
SliceValue    : null
```

```
SliceLevel    : 4
SliceEvent    : END_OBJECT
FatherObj     : 18

ObjectId      : 24
SliceValue    : null
SliceLevel    : 3
SliceEvent    : END_ARRAY
FatherObj     : 11

ObjectId      : 25
SliceValue    : null
SliceLevel    : 2
SliceEvent    : END_OBJECT
FatherObj     : 3

ObjectId      : 26
SliceValue    : null
SliceLevel    : 1
SliceEvent    : END_OBJECT
FatherObj     : 1
```

Listing A.1 – Person Object completely mapped

APPENDIX B – City of Chicago Data Example


```
{ Inspections:
 {
"sid" : 1,
"id" : "A657EC01-A6FD-4C58-A907-6B3D82696B19",
"position" : 1,
"created_at" : 1482409144,
"created_meta" : "386464",
"updated_at" : 1482409144,
"meta" : "386464",
"inspection_id" : null,
"dba_name" : "1978521",
"aka_name" : "ACE PLACE",
"license_" "2496581",
"facility_type" :"Restaurant",
"risk" : "Risk 2 (Medium)",
"address" : "1358 E 75TH ST ",
"city" : "CHICAGO",
"state" : "IL",
"zip" : "60619",
"calendar_date" : "2016-12-21T00:00:00",
"inspection_type" : "License",
"results" : "Pass",
"violations" : "34. FLOORS: CONSTRUCTED PER CODE, CLEANED, GOOD
   REPAIR, COVING INSTALLED, DUST-LESS CLEANING METHODS USED -
   Comments: OBSERVED THE BASEBOARDS FALLING OFF OR MISSING BEHIND
    THE 3 COMPARTMENT SINK, UNDER THE COOKING EQUIPMENT AND AT THE
    FRONT COUNTER. INSTRUCTED TO PROVIDE AND MAINTAIN BASEBOARDS.
   | 35. WALLS, CEILINGS, ATTACHED EQUIPMENT CONSTRUCTED PER CODE:
    GOOD REPAIR, SURFACES CLEAN AND DUST-LESS CLEANING METHODS -
   Comments: OBSERVED HOLES/GAPS IN THE WALLS THROUGHTOUT PREP
   AREA AND INSIDE OF THE RESTROOM. INSTRUCTED TO REPAIR AND
   MAINTAIN ALL WALLS. | 38. VENTILATION: ROOMS AND EQUIPMENT
   VENTED AS REQUIRED: PLUMBING: INSTALLED AND MAINTAINED -
   Comments: OBSERVED NO MOP SINK AT THE FACILITY. INSTRUCTED TO
   PROVIDE AND MAINTAIN A MOP SINK.",
"latitude" : "41.75883291966931",
```

```
25 "longitude" : "-87.59086222278542",
26 "location" : [ "human_address":null, "latitude":"41.7588329196693
     1", "longitude":"-87.59086222278542", "machine_address":null, "
     needs_recoding":false ]
27  }
28 }
```

Listing B.1 – Excert of City of Chicago Data Example

Bibliography

1 ROBINSON, I.; WEBBER, J.; EIFREM, E. *Graph databases*. [S.l.]: " O'Reilly Media, Inc.", 2013.

2 BUGIOTTI, F.; CABIBBO, L. A comparison of data models and apis of nosql datastores. *Dipartamento di Ingegneria della Università di Roma*, 2013.

3 FLORÊNCIO, C.; SANTOS, M. T. P. Evolução dos bancos de dados no ambiente de desenvolvimento de projetos ágeis. *Revista TIS*, v. 4, n. 1, 2016.

4 STÖRL, U. et al. Schemaless nosql data stores-object-nosql mappers to the rescue? In: *BTW*. [S.l.: s.n.], 2015. p. 579–599.

5 BUGIOTTI, F. et al. *A Logical Approach to NoSQL Databases*. 2013.

6 SCHERZINGER, S.; KLETTKE, M.; STÖRL, U. Managing schema evolution in nosql data stores. *arXiv preprint arXiv:1308.0514*, 2013.

7 TIWARI, S. *Professional NoSQL*. [S.l.]: John Wiley & Sons, 2011.

8 BESLIC, A. et al. Towards a solution avoiding vendor lock-in to enable migration between cloud platforms. In: *MDHPCL@ MoDELS*. [S.l.: s.n.], 2013. p. 5–14.

9 HECHT, R.; JABLONSKI, S. Nosql evaluation: A use case oriented survey. IEEE, 2011.

10 DATABASE.ORG nosql. *nosql-database.org*. 2015. Disponível em: <http: //nosql-database.org/>.

11 MUGHEES, M. Data migration from standard sql to nosql. 2014.

12 GOMEZ, A. et al. Experimental validation as support in the migration from sql databases to nosql databases. *CLOUD COMPUTING 2015*, p. 162, 2015.

13 VALE, F.; ROCHA, L. Nosqlayer: a framework for migrating relational datasets to nosql models. *Revista de Iniciação Científica*, v. 14, n. 3.

14 PADHY, R. P.; PATRA, M. R.; SATAPATHY, S. C. Rdbms to nosql: Reviewing some next-generation non-relational databases. *International Journal of Advanced Engineering Science and Technologies*, v. 11, n. 1, p. 15–30, 2011.

15 BĂZĂR, C.; IOSIF, C. S. et al. The transition from rdbms to nosql. a comparative analysis of three popular non-relational solutions: Cassandra, mongodb and couchbase. *Database Systems Journal*, Academy of Economic Studies-Bucharest, Romania, v. 5, n. 2, p. 49–59, 2014.

16 LEE, C.-H.; ZHENG, Y.-L. Automatic sql-to-nosql schema transformation over the mysql and hbase databases. In: IEEE. *Consumer Electronics-Taiwan (ICCE-TW), 2015 IEEE International Conference on*. [S.l.], 2015. p. 426–427.

17 CHUNG, W.-C. et al. Jackhare: a framework for sql to nosql translation using mapreduce. *Automated Software Engineering*, Springer, v. 21, n. 4, p. 489–508, 2014.

18 ZHAO, G. et al. Schema conversion model of sql database to nosql. In: IEEE. *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2014 Ninth International Conference on*. [S.l.], 2014. p. 355–362.

19 SCAVUZZO, M. Interoperable data migration between nosql columnar databases. Italy, 2013.

20 SCAVUZZO, M.; NITTO, E. D.; CERI, S. Interoperable data migration between nosql columnar databases. In: IEEE. *Enterprise Distributed Object Computing Conference Workshops and Demonstrations (EDOCW), 2014 IEEE 18th International*. [S.l.], 2014. p. 154–162.

21 ATZENI, P.; BUGIOTTI, F.; ROSSI, L. Uniform access to non-relational database systems: The sos platform. In: SPRINGER. *Advanced Information Systems Engineering*. [S.l.], 2012. p. 160–174.

22 ATZENI, P.; BUGIOTTI, F.; ROSSI, L. Uniform access to non-relational database systems: The sos platform. In: SPRINGER. *Advanced Information Systems Engineering*. [S.l.], 2012. p. 160–174.

23 ALOMARI, E.; BARNAWI, A.; SAKR, S. Cdport: A portability framework for nosql datastores. *Arabian Journal for Science and Engineering*, Springer, p. 1–23, 2015.

24 SAUR, K.; DUMITRAS, T.; HICKS, M. Evolving nosql databases without downtime.

25 ATZENI, P. et al. The relational model is dead, sql is dead, and i don't feel so good myself. *ACM SIGMOD Record*, ACM, v. 42, n. 2, p. 64–68, 2013.

26 NAYAK, A.; PORIYA, A.; POOJARY, D. Type of nosql databases and its comparison with relational databases. *International Journal of Applied Information Systems*, v. 5, n. 4, p. 16–19, 2013.

27 BUGIOTTI, F.; CABIBBO, L. *An Object-Datastore Mapper Supporting NoSQL Database Design*. 2013.

28 BUGIOTTI, F. et al. Database design for nosql systems. In: SPRINGER. *International Conference on Conceptual Modeling*. [S.l.], 2014. p. 223–231.

29 ATZENI, P.; BUGIOTTI, F.; ROSSI, L. Uniform access to nosql systems. *Information Systems*, Elsevier, v. 43, p. 117–133, 2014.

30 STRAUCH, C.; SITES, U.-L. S.; KRIHA, W. Nosql databases. *Lecture Notes, Stuttgart Media University*, 2011.

31 MONIRUZZAMAN, A.; HOSSAIN, S. A. Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. *arXiv preprint arXiv:1307.0191*, 2013.

32 STROZZI, C. Nosql-a relational database management system. *Lainattu*, v. 5, p. 2014, 1998. Disponível em: <http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page/>.

33 KUZNETSOV, S.; POSKONIN, A. Nosql data management systems. *Programming and Computer Software*, Springer, v. 40, n. 6, p. 323–332, 2014.

34 TAURO, C. J.; PATIL, B. R.; PRASHANTH, K. A comparative analysis of different nosql databases on data model, query model and replication model. In: *Proceedings of the International Conference on Emerging Research in Computing, Information, Communication and Applications ERCICA*. [S.l.: s.n.], 2013.

35 CHANDRA, D. G. Base analysis of nosql database. *Future Generation Computer Systems*, Elsevier, 2015.

36 BROWNE, J. *Brewer's CAP Theorem*. 2009. Disponível em: <http://www.julianbrowne.com/article/viewer/brewers-cap-theorem>.

37 BREWER, E. Cap twelve years later: How the" rules" have changed. *Computer*, IEEE, v. 45, n. 2, p. 23–29, 2012.

38 FRANK, L. et al. The cap theorem versus databases with relaxed acid properties. In: ACM. *Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication*. [S.l.], 2014. p. 78.

39 STÖRL, U. et al. Schemaless nosql data stores–object-nosql mappers to the rescue? In: *Proc. BTW*. [S.l.: s.n.], 2015. v. 15.

40 PRITCHETT, D. Base: An acid alternative. *Queue*, ACM, v. 6, n. 3, p. 48–55, 2008.

41 TRUICA, C.-O. et al. Performance evaluation for crud operations in asynchronously replicated document oriented database. In: IEEE. *Control Systems and Computer Science (CSCS), 2015 20th International Conference on*. [S.l.], 2015. p. 191–196.

42 HANSEN, C. K.; COKLJAT, N. Ensuring base consistency in a crud middleware layer for heterogeneous databases.

43 SAKR, S. Cloud-hosted databases: technologies, challenges and opportunities. *Cluster Computing*, Springer, v. 17, n. 2, p. 487–502, 2014.

44 ORACLE. *Enum JsonParser.Event*. 2014. Disponível em: <https://docs.oracle.com/javame/8.0/api/json/api/com/oracle/json/stream/JsonParser.Event.html>.

45 SMITH, B. *Beginning JSON*. [S.l.]: Apress, 2015.

46 BRAY, T. The javascript object notation (json) data interchange format. 2014.

47 KLEIN, J. et al. *NoSQL Data Store Technologies*. [S.l.], 2014.

48 CATTELL, R. Scalable sql and nosql data stores. *ACM SIGMOD Record*, ACM, v. 39, n. 4, p. 12–27, 2011.

49 SADALAGE, P. J.; FOWLER, M. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. [S.l.]: Pearson Education, 2012.

50 HECHT, R.; JABLONSKI, S. Nosql evaluation: A use case oriented survey. IEEE, 2011.

51 BONDIOMBOUY, C.; VALDURIEZ, P. *Query Processing in Multistore Systems: an overview*. Tese (Doutorado) — INRIA Sophia Antipolis-Méditerranée, 2016.

52 ATZENI, P.; CAPPELLARI, P.; BERNSTEIN, P. A. Model-independent schema and data translation. In: *Advances in Database Technology-EDBT 2006*. [S.l.]: Springer, 2006. p. 368–385.

53 Apache Software Foundation. *sqoop*. 2015. Disponível em: <http://sqoop.apache.org>. Acesso em: 27 oct. 2015.

54 PAPOTTI, P.; TORLONE, R. An approach to heterogeneous data translation based on xml conversion. In: CITESEER. *CAiSE Workshops (1)*. [S.l.], 2004. p. 7–19.

55 MICHEL, F. et al. *xR2RML: Relational and non-relational databases to RDF mapping language*. [S.l.], 2014.

56 CONSORTIUM, W. W. W. et al. R2rml: Rdb to rdf mapping language. World Wide Web Consortium, 2012.

57 SCHERZINGER, S.; STÖRL, U.; KLETTKE, M. A datalog-based protocol for lazy data migration in agile nosql application development. In: ACM. *Proceedings of the 15th Symposium on Database Programming Languages*. [S.l.], 2015. p. 41–44.

58 BANSEL, A. *Cloud based NoSQL Data Migration Framework to achieve data portability*. Tese (Doutorado) — Dublin, National College of Ireland, 2015.

59 ATZENI, P. et al. Data modeling in the nosql world. *Computer Standards & Interfaces*, Elsevier, 2016.

60 FABRO, M. D. D. et al. Amw: A generic model weaver. In: *Proc. of the 1ères Journées sur l'Ingénierie Dirigée par les Modèles*. [S.l.: s.n.], 2005. v. 200, n. 5.

61 ORACLE. *Java Platform Standard Edition 8 Documentation*. 2014. Disponível em: <http://www.oracle.com/technetwork/java/javase/documentation/index.html>.