American Scientific Research Journal for Engineering, Technology, and Sciences (ASRJETS)
ISSN (Print) 2313-4410, ISSN (Online) 2313-4402
© Global Society of Scientific Research and Researchers
http://asrjetsjournal.org/

PDDM: A Database Design Method for Polyglot Persistence

Cristofer Zdepski[a], Tarcizio Alexandre Bini[b*], Simone Nasser Matos[c]

[a,b,c]*Federal University of Technology – Parana (UTFPR), Address Doutor Washington Subtil Chueire,330-Jardim Carvalho, Ponta Grossa, 84017-220, Parana, Brazil*

[a]*Email: cristoferz@gmail.com*

[b]*Email: tarcizio@utfpr.edu.br*

[c]*Email: snasser@utfpr.edu.br*

Abstract

Databases by Web 2.0 has revealed the limitations of the relational model related to scalability. This led to the emergence of NoSQL databases, with data storage models other than relational ones. These databases propose solutions to such limitations through horizontal scalability and partially compromise data consistency. The combination of multiple data models, called polyglot persistence, extends these solutions by providing resources for the implementation of complex systems that have components with distinct requirements that would not be possible by the use of only one data model in a satisfactory way. However, there are no consolidated methods for the NoSQL database design and neither methods for design systems that apply the polyglot persistence. This work proposes a database design method applied to systems that use polyglot persistence, combining different data models. This method can be applied to the relational model and aggregate-oriented NoSQL data models. The method defines a set of sub-steps based on the existing concepts of database design. The goal is to define a formal process to assist in defining the data models to be used and to transform the conceptual design into a logical design. The method application is demonstrated in some test cases, in order to show its results and applicability for later execution of the physical design of these databases.

Keywords: NoSQL; Database Design; Polyglot Persistence; Logical Level Design.

----

* Corresponding author.

1. Introduction

Databases have been applied to the context of organizational systems since the mid-1970s, when they gradually began to replace file-based storage systems. At the same time, recognition of the importance of data has grown as an integral part of enterprise resources [1]. Over the years, many DBMS have emerged and, due to its great adaptability to various scenarios and data consistency, the relational model dominated the market becoming the most widely applied [2]. The popularization of Web 2.0, driven by the growth of interactive platforms such as social networks, blogs, and content managers, has brought much larger scales to the petabyte scale, with daily terabyte scale growths [3]. This growth evidenced weaknesses of the relational model due to its limitations with high access volumes or large amounts of data, caused by its difficulty of distribution on multiple servers, called horizontal scaling [4]. Another limitation of the relational model is the flexibility of the data. Many Web 2.0 applications work with unstructured or semi-structured data, such as content managers and social media [5]. The relational model works with structures defined in the form of rows and columns, which imposes restrictions on data format and makes it difficult to use in applications with unstructured data. To address the limitations of scalability and flexibility, new data models called NoSQL have emerged. The term NoSQL is used to represent "non-relational" databases that implement other data storage and access formats [4]. Many DBMSs NoSQL database class has come up with different data models, including key-value, document-oriented, column-family, and graph-oriented models, seeking the ability to address some of the limitations of the relational model. While providing the ability to address some limitations of the relational model, NoSQL databases are not the unified and definitive proposal for data storage. The NoSQL databases use simpler storage structures, which allows for greater data flexibility and scalability, but have other limitations when compared to the relational model. They typically lack the ability to construct complex queries such as joins or aggregates, and virtually all impose limitations on data consistency. As with the relational model, a single data model may not be sufficient to meet the needs of a complex system. To solve these limitations, authors such as [6,7] suggest the application of multiple storage models in the systems implementation, aiming to use the best characteristics of each one, approach that receives the polyglot persistence name. Polyglot persistence provides more troubleshooting capabilities for complex systems with essentially distinct parts, such as session control, access history, and social search [7]. These different parts of the same system often have different requirements, and applying a single data model involves adaptations to data structures that can lead to not-optimized solutions. In large-scale systems, this lack of optimization becomes even more evident as it compromises the viability of the system. Although polyglot persistence solves certain problems of a complex system, it also brings new challenges [8]. The differences between data models NoSQL when compared to the relational model bring difficulties in the data integration process. Transforming data between models is not always a simple process because data structures are different. Polyglot persistence adds to the development of systems a complexity that needs to be justified by the gains provided by the diversity of data models. The new multi-model data scenario provided by polyglot persistence does not yet have a consolidated database design method. While the relational model has several studies on the subject, NoSQL models are still poorly explored in this area, as is polyglot persistence [9]. Aiming to use several data models in a single project, the idea was created to create a database design method applied to polyglot persistence, incorporating the NoSQL models and the relational model, bringing a unified view. The proposed method, named PDDM (Polyglot Database Design Method), is a continuation of the work

started by [10] and was developed from the essential steps of database design: Conceptual, Logical and Physical Design. Sub-processes were created within the logical project. As a result, these sub-processes and their application to test cases will be presented, providing the ability to design a system that uses some of the NoSQL data models and the relational model in an integrated manner. This paper is organized as follows. Section 2 presents works related to the subject of NoSQL database design and polyglot persistence. Sections 3 and 4 present the PDDM sub-processes and demonstrates the application for some use cases. Finally, Section 5 presents the conclusions about the method, use case applications and presents some proposals for future works.

2. Related Works

In order to find available methods in the literature related to NoSQL database design and polyglot persistence, a systematic mapping was performed. This mapping aimed to find: a) the main authors of the area, b) which are the most common application cases of NoSQL databases in the literature, c) which methods exists related to NoSQL database design and d) which of these methods deal with or could be extended to use polyglot persistence. To perform this mapping was used the *MethodiOrdinatio*, described by [11] and which allows the classification of papers related to the subject based on 3 factors and allows the definition of a relevance factor that is used for the classification of them. Based on this systematic mapping [12], the main authors and methods used for NoSQL database design were identified, and it was concluded that there is no uniformity in the strategies used for this purpose. The proposals differ widely in tools and methods used for database design, even for the same database. In addition, most of the methodologies found are restricted to just one data model, and often to just one software. Among the found methods, [13] presents NoAM, a methodology that proposes a form of intermediate data representation, regardless of the database to be used. Such a strategy applies only to aggregate-based models and allows the same model to apply to key-value, document-oriented, and column-family databases. According to the authors, the intermediate model could be used for the creation of any of the pointed data models in any software, simply converting the intermediate model into the specific model of each software. The strategy, however, does not consider the particularities of each model and its primary objectives, but only considers a standardized structure to be used for all. In [14] proposes a set of tabular structure conversion rules for a relational model for HBase implementations, a column family model database, and a set of rules for Hive conversion, an SQL implementation based on Hadoop. The article only presents the application in HBase, a software that implements the column family model, and Hive being all data stored in tabular form, suggesting that the strategy only applies to tabular structure models such as the column family and relational. Another proposal, called NoSE, proposes an approach capable of identifying the application of good practices in the development of NoSQL databases without coding them [15]. The proposal suggests that the automated design process produces more efficient database schemas than would be produced in a rule-based approach, a process commonly applied in this context. The proposal is limited solely to use in the column family data model and more specifically for use with Cassandra, but suggests that it could also be adapted for use with HBase. Still, it would not be adaptable to other models, such as document-oriented, key-value or graphs. Another concern pointed out by the authors is the absence of a uniform access strategy to the various existing databases. A proposal is presented by [16], called SoS (Save our Systems). This strategy also addresses the logical modeling process required for data models to be compatible with the so-called “meta-layer”, which would be an intermediate step between the application and the databases to be used. Subsequently, Reference [17] presents a

SoS-based programming model to allow homogeneous handling of different NoSQL databases, specifically Redis, MongoDB and HBase. The models presented are based on the concept of aggregates but the authors do not present any restrictions on the use of other data models. Based on the concept of "schema less", Reference [18] suggests a methodology for documenting schema variants to list data storage rules. This methodology is not a database design methodology, but only documentation of an already implemented database. It can be seen that there are few strategies that can fit all data models, just as many of them are not exactly strategies for database design, but strategies for documenting already defined databases. Most of the selected papers propose methodologies focused only on aggregate-based models, and some even only for one of these models. The SoS methodology proposed by [16] mentions that its application to both aggregate and relational models is possible, but it is more of a development methodology than a database design methodology. It establishes what they call the "meta-layer", responsible for the abstractions needed to use different software or even different data models transparently to the rest of the application. However, this does not establish a method for database design, only a more flexible form of use that allows for easier transition between software and data models. The NoAM methodology, proposed by [13], proposes a methodology for database design by creating a layer for the definition of aggregates and future conversion to the specific models to be used. Through this methodology it is possible to use the same logical model and apply it to databases of the key-value model, document-oriented or column family. It is a methodology that allows the use of various data models, but only aggregate-oriented ones. It also does not establish a way to establish the boundaries between models in the same application. In [6] demonstrates an implementation model using polyglot persistence for an e-commerce system, but does not establish a methodology for database design. Its focus is mainly on presenting the challenges of deploying an application that requires multiple databases.

3. Proposal

As can be observed in the related works, some limitations for the use with polyglot persistence. Some are applied to specific data models, others were an approach to systems development rather than a design method, and none have the ability to draw boundaries between different data models in the same system. This section introduces the PDDM, which is used for database design on systems that require multiple data models. The method was developed based on the database design steps. During the definition of the method were identified the limitations of the database design steps related to the use of multiple data models and which processes would be necessary to use the NoSQL models. At the moment, the aggregate-oriented NoSQL data models and the relational model were addressed, and the graph data model is excluded for future extension of the method. The PDDM is based on the stages of database design that are the accepted of several authors including [19,20] and [21], they are: a) Conceptual Design, b) Logical Design and c) Physical Design. To allow the definition and use of multiple data models, some sub-steps have been added, as can be seen in Figure 1.

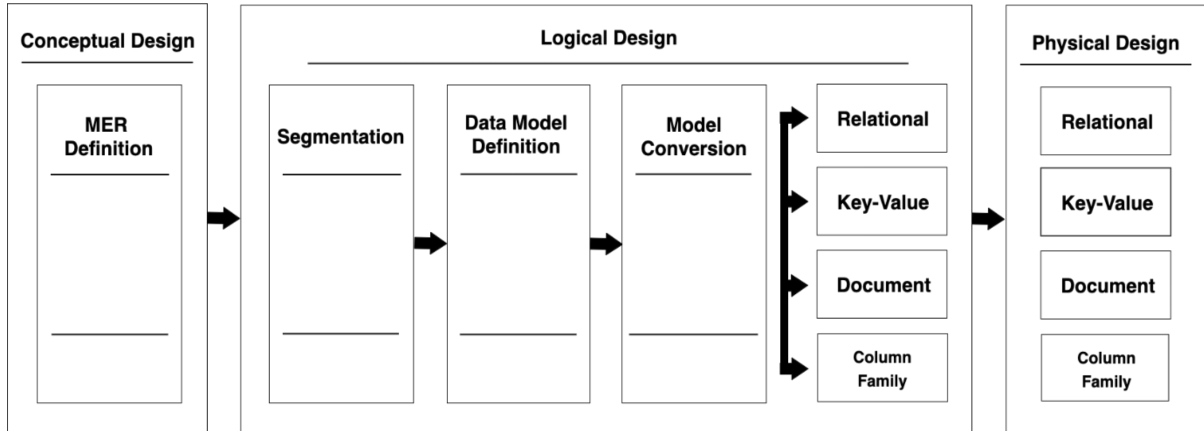

Figure 1: Method Overview.

The PDDM application, as well as other database design methods, aims at transforming defined requirements for the database, from conceptual design to its physical implementation. The main difference from existing methods is the ability to transform conceptual design into multiple data models by defining the boundaries of each. The following sections describes the PDDM steps for designing polyglot-persistent databases. For a better view of these steps will be used the model initially proposed by [10], shown in Figure 2, which brings a simplified e-commerce model.

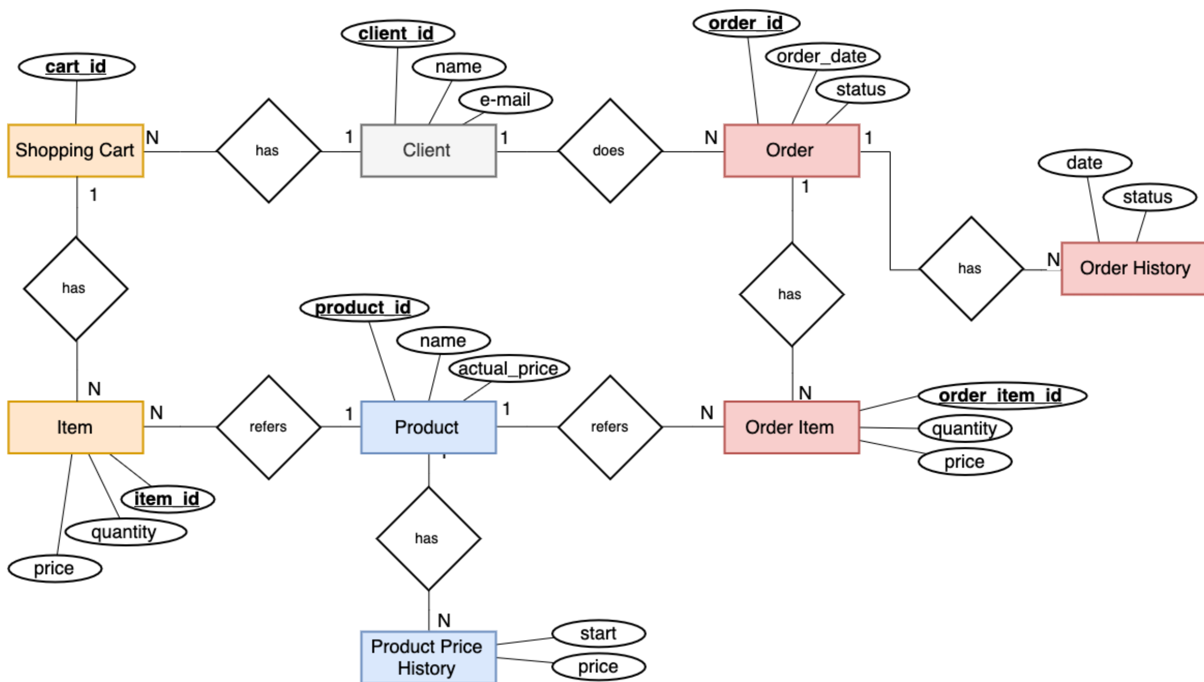

Figure 2: Example Conceptual Design.

3.1. Segmentation

As part of the logical design definition includes the conversion of the logical design into structures compatible

with the data model to be used, it is necessary that at this point the data models to be used are defined. Since the method was created for design in systems that can use polyglot persistence, it is necessary to define which are the cutoffs between ERD (Entity-Relationship Diagram) entities for the data models to be used. The first step to allow the delimitation of the various data models to be used is the identification of the points in which the system can be segmented without compromising its operation. To achieve this consistently, the system is divided from its functionalities, which must be provided by the input model. This ensures that they are structurally concise and can be physically and logically isolated from each other. These divisions are called Segmentation Units, and each of them represents a complete and independent feature of the system. Having the functionalities previously defined, for the definition of segmentation units there are 2 basic rules: 1) The units must be completely functional from the system point of view, that is, they establish functionalities that can be used in isolation. 2) They must be independent of each other and do not require entities from other units to be used. These rules are necessary to ensure that each unit can be treated as an independent system and can be designed with different data models and stored in different data structures. For a better understanding of the segmentation process, it can be divided into 2 steps: 1) Identification of functionalities and definition of segmentation units and 2) Creation of replicas for isolation of units. The identification of the functionalities and definition of the segmentation units consists in the division of the original ERD based on the previously described functionalities in order to obtain multiple ERD, which aim the application in different data models. Examples of features are a Shopping Cart in an e-commerce system, a Product Search Engine, and an Accounts Receivable Management system. Figure 3 shows an example of what entity segmentation would look like between units in the example model.

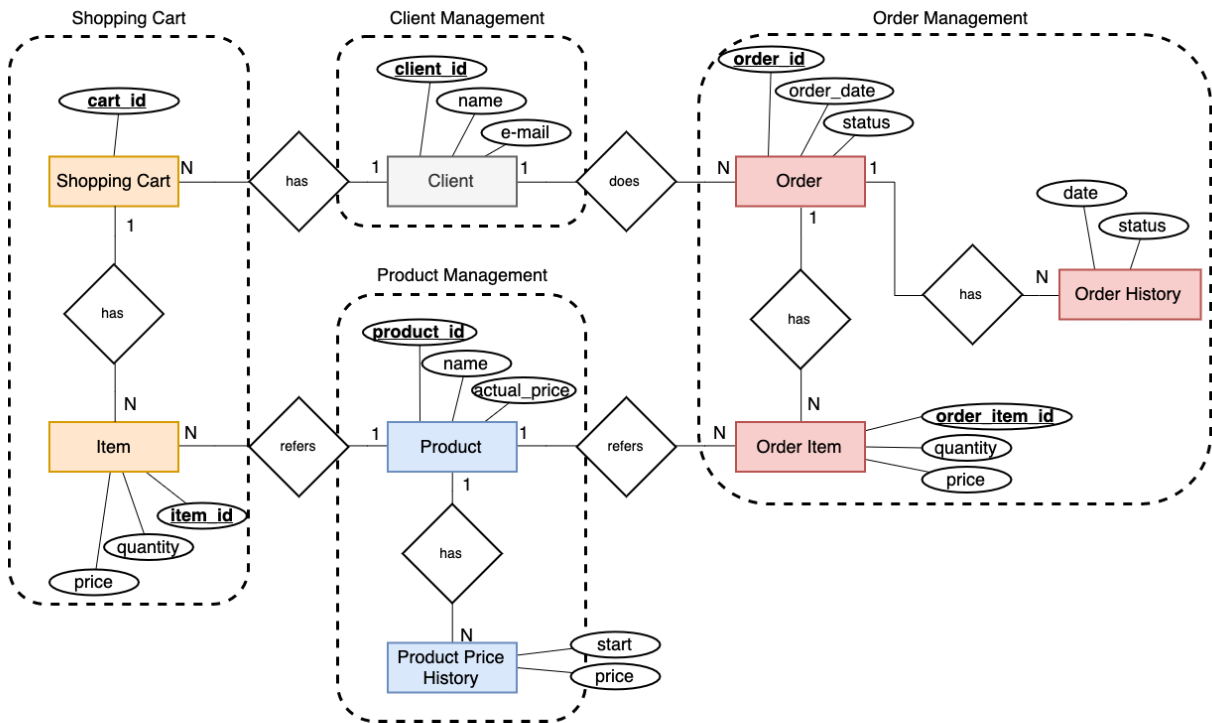

Figure 3: Feature Segmentation Example.

However, only the separation of entities into distinct models creates incomplete models from the functional

point of view. Analyzing the model of Figure 3 and considering the units as logically isolated structures, it can be observed that the entity *Item* of the unit *Shopping Cart* has dependence on the entity *Product*, because of its 1-to-N relationship. To fix this inconsistency, entity replicas are created. Creating replicas for unit isolation consists of creating replicas of entities that are directly involved in the operation of one unit but belong to another, violating the principle of unit independence. The *Product* example shown by Figure 3 refers to an entity that is directly involved in the operation of other units, the *Shopping Cart* and *Orders*. It is important to highlight that the segmentation units aim to allow the system segmentation in different data models and, consequently, using multiple databases. With segmentation done as shown in Figure 3, the application needs to query data from other units. This means access to different databases, with different data models in one operation, for system operation. These queries cause performance losses because they require multiple database requests, and bring the complexity of integrating data from different models into a single operation. In order to eliminate this access to other units, the solution is to replicate the data from external entities necessary for the operation of the unit, forming replicas of the entities. This way, all information required for the operation is stored in the same database, which facilitates implementation at the application level and makes the units independent each other. Data replication between units is defined by the type of relationship between units. Relationships where unit-side cardinality is *N* represent the need to create replicas. Figure 4 represents the *Product* and *Customer* entities, applying replica creation. Note that the entity *Product* has been replicated in both units *Shopping Cart* and *Order Tracking*, due to the cardinality N in these units, while the entities *Item* and *Order Item* did not need to be replicated in drive *Product Management*.

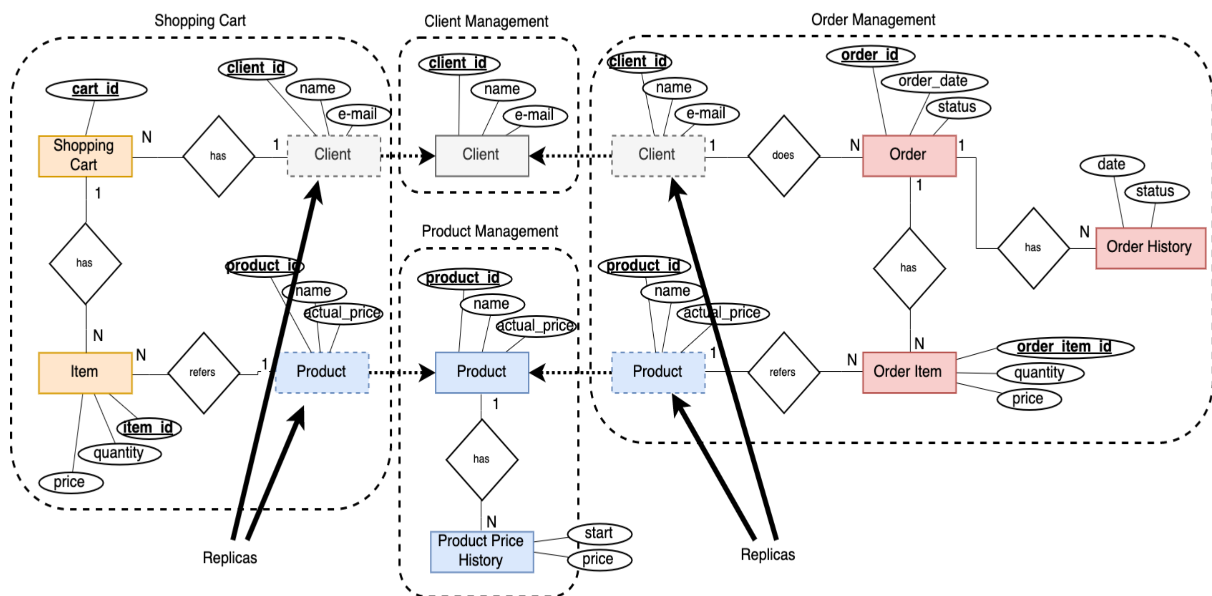

Figure 4: Replica Creation for the Model.

3.2. Data Model Definition

Having defined the boundaries between system functionality and ensuring that each unit can operate independently, it is possible to define which data models are best suited for each segmentation unit. This is a complex task because it requires prior knowledge of data models, their best applications, limitations and their

suitability to the requirements of each specific system functionality. At the moment, the PDDM does not propose a method for this definition, making this definition dependent on the knowledge and experience of the designer. Information such as functional requirements, query functions to be used in databases and some peculiarities of NoSQL DBMS can help in defining these models.

3.3. Data Model Logical Design

Once the data models for the segmentation units are defined, it is possible to go to the essential definition of the logical database design, which is to convert the conceptual model into a structure compatible with the data model to be used [20]. This process can be performed for each segmentation unit independently, since they are independent of each other as seen in Section 3.1. From this point on, the project becomes independent for each data model, because the design of one of them will not directly affect the operation of the others. Aiming at the execution of the logical project, it was necessary to define a representation model that was compatible with the data structures to be used in the aggregate-based databases. Figure 5 shows a metamodel with data structures and their relationships with other entities, attributes and data formats to be used.

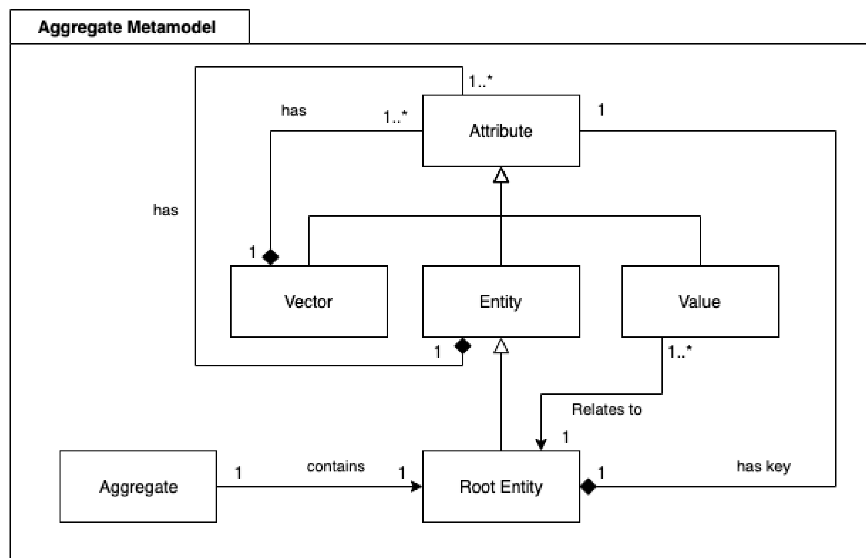

Figure 5: Metamodel for Aggregate Model Definition.

Based on Figure 5, it was possible to determine 2 possible types of relationships between entities:

- Aggregation: Where the entity is referenced by another entity, expressed by the "Relates to" relationship in the metamodel.
- Composition: Where the entity is embedded in another entity, represented by the entity-type attribute, that is, sub-entity of an entity.

Having defined the two possible relationship types for the model, it is possible to establish two types of entities and their function within the model. The first of these are the entities considered to be the root of the aggregates. They establish the name of the aggregates and their primary key becomes the primary key of the aggregates to

be created by the model. All entities in the model that do not participate in the composition of another entity or, in other words, that do not compose another entity, are part of that type. The other entity type is composed of excluded entities of the first type, that is, entities that are part of the composition of other aggregates. As described in the composition relationship, these entities are embedded in related entities and therefore only make up the aggregate structure to which they are related. Your primary keys become irrelevant outside the context of the aggregate as they will no longer be externally accessible, but must be preserved to allow identification of the substructure. Figure 6 shows an example of using the aggregates model to represent 2 aggregates: *A* that has an embedded entity *A1*, represented by the composition relationship, and *B*, which has a relationship with *A*, represented by the aggregation relationship.

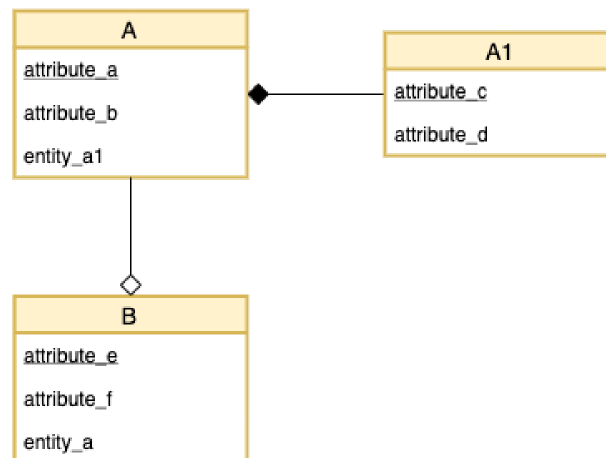

Figure 6: Metamodel Utilization Example.

3.4. Data Model Conversion

With the defined aggregate representation model, a set of 7 rules was established to allow converting the ERD to the aggregate model, maintaining their definitions. They are:

- R1: Applies to all entities in the source model. In this rule, every ERD entity becomes an aggregate model entity, preserving its attributes. It is suggested to keep the attribute names of the original entity to facilitate the understanding of the generated model. Primary keys are also maintained for the entities, which will be used as the identification key for the aggregate. For multi-valued attributes, vector attributes are created. The entities generated by applying rule R1 are the basis for applying the other PDDM rules.
- R2: Handles entities with *1* to *1* relationship. According to this rule, *1* to *1* relationship generates a mutual relationship between both entities with cardinality *1* by creating an attribute on each entity for the relationship. The R2 rule also applies to *1* to *1* self-relationship cases, following the same principle as the relationship between two entities.
- R3: Applies to *1* to *N* relationships. According to this rule, a *1* to *N* relationship between entities transforms into a *1* .. * cardinality relationship, creating a vector-type attribute for storage, and a cardinality *1* on the *N* to *1* relationship, creating an attribute for it.

- R4: Applies to *N* to *N* relationships, but only to relationships without attributes. This distinction between relationships with and without attributes is necessary to simplify *N* to *N* relationships without attributes. In this conversion, both entities create a *1..** cardinality relationship with each other by creating a vector-type attribute for the relationship on each entity.
- R5: It deals with *N* to *N* relationships that have attributes. Since the use of list type attributes, as used in R4, does not allow the addition of attributes, it is necessary to create an associative entity, which is then responsible for maintaining the attribute. This associative entity is then created in both directions of the relationship, containing all of its attributes, to allow both entities access to the attribute.
- R6: It deals with a *1* to *1* relationship with weak entities. Applying the rule is similar to rule R2 with the difference in the type of relationship created, which in this case is a composition. Thus, during the construction of the aggregate, the weak entity becomes embedded in the related entity.
- R7: It deals with *1* to *N* relationships with weak entities. It is also a rule similar to rule R3, with the difference in the type of relationship created, which in this case is a composition. Thus, the strong entity has a list of related weak entities embedded.

With this set of 7 rules, it's possible convert an ERD that follows the Peter Chen model [22], preserving its original characteristics and allowing it to be applied to aggregate-based databases.

4. Application

In order to demonstrate the practical application of PDDM, some test cases were chosen in different fields of knowledge and from different authors. It was possible to demonstrate the applicability of the method in several scenarios. Among the test cases, the entity-relationship model proposed in [19] was used in this paper to demonstrate the method application because it is well known in the literature. This model deals with a university control system. In addition, the credit control of students, teachers and disciplines offered, controlling the classrooms used and class schedules. The model presents several characteristics of conceptual modeling, such as *1* to *N* and *N* to *N* relationships, and self-relationships which allow us to explore the application of the PDDM method in more detail.

Figure 7 demonstrates the schema proposed by [19]. It was added the functionality definition, required for the PDDM application, through the colors of each entity. In this case the blue color represents the Course Management, and the yellow color the Discipline Management. As described earlier, the first step in applying PDDM is to segment the model based on its definition of functionality. This process begins by separating the entities of each functionality into a specific ERD for each segmentation unit. Next, the entities that need the creation of replicas are identified by checking the relationships with cardinality *N* that are in the boundary between the functionalities. In the case of Course Management, there are three unit boundary relationships with the following entities: Teacher, Student, and Section. However, all these relationships have cardinality *1* on the entity side of Course Management, which rules out the need to create replicas. The result of segmentation is only entities belonging to functionality, as shown in Figure 8.

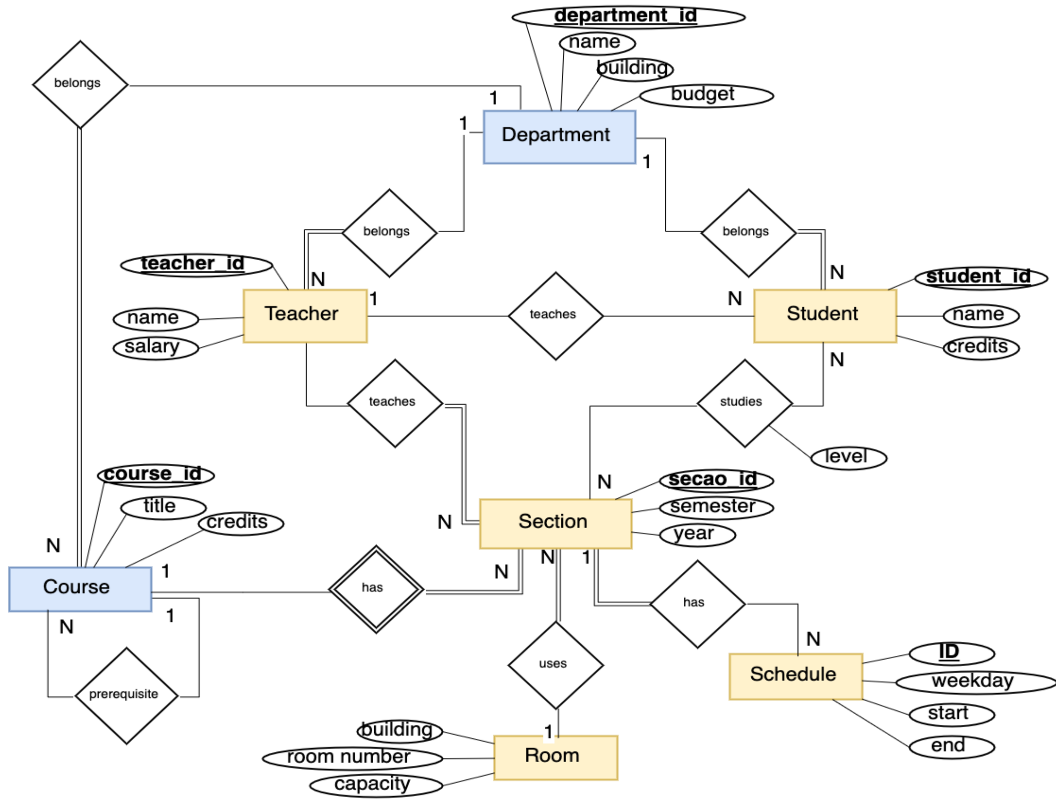

Figure 7: ERD for the test case.

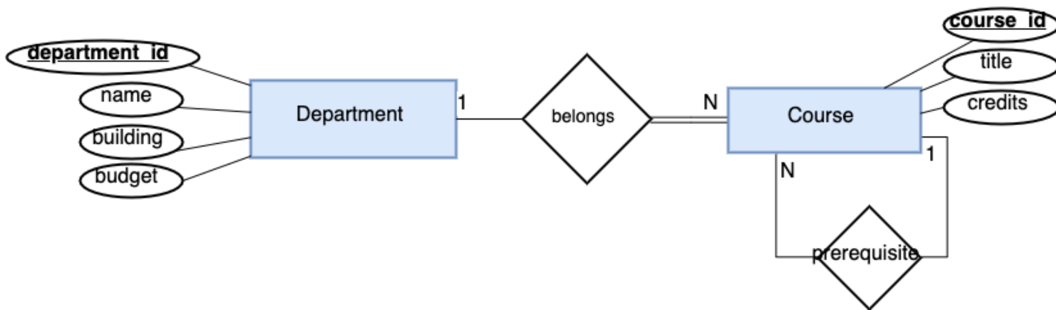

Figure 8: Segmented ERD for the test case.

With properly separated models, it is possible to perform the data model definition step. This process currently requires the designer's knowledge of the capabilities of each data model and its most recommended applications. Just to illustrate, we will use the column family data model.

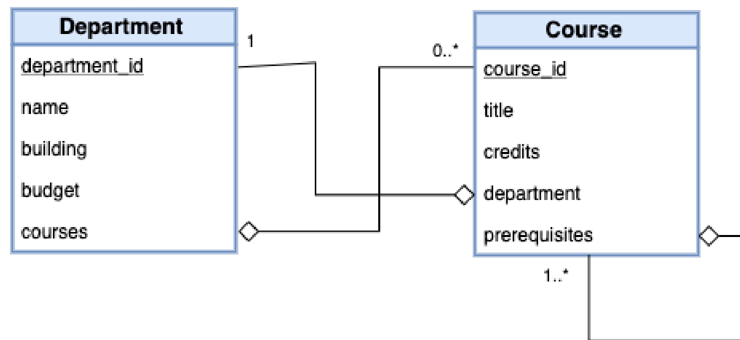

Figure 9: Application of PDDM for Course Management.

Then the transformation step can be performed, where the model transformation rules will be applied. The result of this transformation can be seen in Figure 9. Of the transformation rules, two have been applied:

- R1: Applies to all entities. Through it were created the entities Department and Course, preserving the primary key and other attributes of MER;
- R3: Applies to *1* to *N* relationships that can be seen in Figure 8, between the Department and Course entities and in the Course entity self-relationship. In this step relationships are created between previously created entities and attributes *courses*, *department* and *prerequisites*.

At this point we have the complete aggregate model for Course Management, where two aggregates can be identified: Department and Course, composed of their respective entities. These two aggregates can then be used to represent data in a column family database, in this case the Cassandra [23] as shown in Figure 10.

```
cqlsh:courses> create table department (department_id int primary key, name varchar, building varchar,
budget float, courses list<int>);
cqlsh:courses> create table course (course_id int primary key, title varchar, credits int, department int,
prerequisites list<int>);
```

Figure 10: Application of Aggregate Model to Cassandra.

The same segmentation process was applied to the Course Management functionality. As with Course Management, functionality entities were placed in a separate ERD and their relationships with external entities were analyzed. In this unit, the three existing relationships with the Department and Course entities have *N* cardinality in the Teacher, Student, and Section entities, which dictates that replicas are required for the unit. The replicas are then created in the model maintaining their original coloration for easy identification of the source unit, but with dashed lines to identify that they are replicas. In creating replicas, the designer may choose to eliminate entity attributes that he deems unnecessary for the unit, except for its primary key that must be retained. To demonstrate this possibility, it was decided to disregard the building and budget attributes of the Department entity, as they are not necessary for Discipline Management. Figure 11 presents the resulting ERD from the application of the segmentation process, where it is possible to observe the two created replicas.

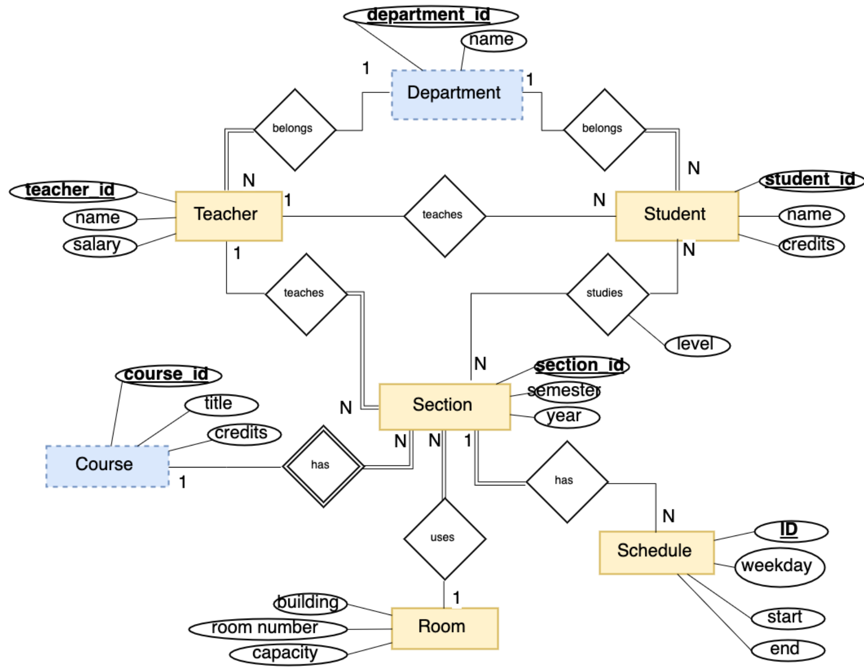

Figure 11: Segmented ERD for Discipline Management.

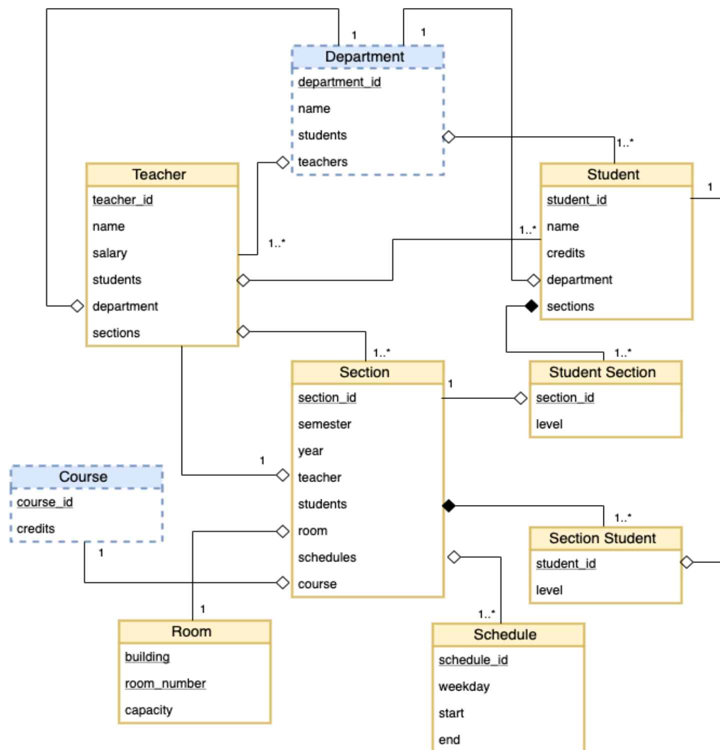

Figure 12: PDDM applied to Discipline Management.

For demonstration in another aggregate data model, we chose to use the document-oriented data model. While applying the conversion rules, it is clear that the Discipline Control functionality has more entities and relationships and a greater number of conversion rules apply, as can be seen in Figure 12. They are:

- R1: Recreate all entities with their keys and attributes;
- R3: Applies to all *1* to *N* relationships in aggregation form, creating attributes for relationships in entities. Examples of this relationship are the relationships between the *Department* and *Professor* entities, and between *Student* and Department;
- R5: Applies to *N* to *N* relationships with attributes. This case can be seen between the *Student* and *Section* entities. In this case, 2 entities were created, *Student Section* and *Student Section*, representing the two senses of the relationship, which is necessary to allow access to related entities in both directions. These associative entities are created as composition of the source entity, have the attributes of the relationship, and the field of association with the opposite entity.

With the complete aggregate model for Course Management you can identify its aggregates. This unit has 2 composition relationships, which causes not all of its entities to become aggregates, such as the *Section Student* and *Student Section* entities. In this case they will be part of the constitution of the *Section* and *Student* aggregates, respectively. We therefore have the following aggregates defined: *Department*, *Course*, *Teacher*, *Student*, *Section*, *Room* and *Schedule*. To exemplify the execution of the physical project, it was decided to use MongoDB [24] as DBMS, as it is widely used and represents the document-oriented data model. The *collections* framework was used to define aggregates in MongoDB, creating one for each aggregate. Insertion into MongoDB was done with sample data, which is shown in Figure 13.

```
db.department.insert({"_id":1,"name":"DAINF","students":[21],"teachers":[11]});
db.teacher.insert({"_id":11,"name":"John","salary":1600,"students":[21],"department":1,"sections":[31]});
db.student.insert({"_id":21,"name":"Cristofer","credits":18,"department":1,
"sections":[{"section_id":31,"level":1}]});
db.section.insert({"_id":31,"name":"Research Methodology","semester":3,"year":2019, "teacher":11,
"students":[{"students":21,"level":1}],"room":41,"schedules":[51,52]"course":61});
db.course.insert({"_id":61,"name":"Computer Engineering","credits":30});
db.room.insert({"_id":41,"building":"Block L","room_number":"L304","capacity":40});
db.schedule.insert({"_id":51,"week_day":"Monday","start":"9:50","end":"11:30"});
db.schedule.insert({"_id":52,"week_day":"Thursday","start":"14:30","end":"16:10"});
```

Figure 13: Usage of Data Model on MongoDB.

Through the results obtained by the application of PDDM in the proposed test cases it was possible to realize that the method is applicable to different domains. The test cases were selected for the application of the method seeking to bring domain diversity, different authors and presenting specific peculiarities that allowed the application of all conversion and segmentation rules.

5. Conclusion

The increasing amount of data generated and manipulated by current applications has demanded the creation of new databases that use different forms of data access and storage, in order to allow the solution of problems previously not possible with the widely used databases. relational data because of its limitations in working with this large volume of data. This led to the emergence of NoSQL data models bringing, among other features,

capabilities for processing large amounts of data, allowing for easy scalability and data distribution. However, none of the NoSQL templates created is the ultimate solution to all current software needs. In the development of complex software, the combination of several models is necessary to provide the functionality of different data models to meet the different requirements that may arise in the application domain. To this end, the concept of polyglot persistence emerged, which deals with the combination of several data models in order to combine the various characteristics necessary for the development of these applications. However, unlike relational database design, the design of applications that use polyglot persistence does not yet have consolidated methodologies and its development uses the knowledge of the designers, examples and best practice sets provided by the developers of the NoSQL databases. Seeking more information about the currently existing methodologies for the design of these databases, a systematic mapping was performed. This mapping sought methods capable of designing NoSQL databases, even partially, bringing the trends and limitations of these methodologies and their applicability to the design of applications that use polyglot persistence. From this mapping was possible to notice the absence of design methodologies focused on polyglot persistence because, although some methodologies work with more than one data model, none provides methods of separation of data models within same project, working with only one storage model per application. Given the need for a method capable of designing for polyglot persistent applications and the increasing use of NoSQL databases, this paper presents the PDDM database design method. The method aims to enable the design of databases that use, in combination, multiple data models, such as NoSQL and relational models, but only aggregate-oriented NoSQL models are addressed in this paper. The method was based on the database project development steps and defined a set of sub-steps considered necessary for model segmentation, identification of the most appropriate data models to be employed and the conversion from conceptual to logical design. For the representation of aggregate data structures, a metamodel was created with the necessary characteristics for the data structures to be stored in the databases. From this model, a set of conversion rules was defined in order to transform the MER used as input by the method into an aggregate model, keeping its original definitions. These rules cover the conversion of MER entities and the types of relationships available. To prove the applicability of the method, test cases were used to demonstrate its use in different domains, created by different authors and with different data sets. Models have also been selected to allow all conversion rules to be applied, as some types of relationships or entities, such as weak entities, are less common in the available models. Although PDDM does not cover conversion to physical design, some possible applications of the method to the physical design of the most common database design for each of the aggregate-oriented NoSQL data models have been demonstrated. During the development of the PDDM method, some points were identified as the basis for future work, as they could not be covered in this work, both because they are different domains of knowledge, and because they broaden the scope of research significantly. One of the method's improvements would be to define a methodology for identifying MER functionality used as input to PDDM. Of the models used in the test cases, only the e-commerce example has this identification, and yet it was a definition made without an established method, which can lead to definition errors and flaws in the resulting model. A methodological definition of these functionalities should involve requirements engineering and was therefore discarded at this point in the PDDM definition. Converting the MER to the aggregate data model, although it is a simple process with rules in place, becomes more laborious as the schema to be converted increases and is susceptible to errors during execution. To avoid such effort and the possible errors generated, the implementation of a tool for the

automation of the conversion process can bring great advantages to the process and even make it possible to apply it in large schemes, as is usually the case with commercial applications. As a last suggestion of future works, PDDM currently covers only aggregate-based NoSQL data models, but its definition does not prevent other models from being incorporated. These include graph-based and other hybrid database applications such as OrientDB [25] and other databases without a clearly established model such as ElasticSearch [26].

References

[1]. T.M. Connolly, C.E. Begg. Database Systems: A Practical Approach to Design, Implementation and Management. USA: Pearson Education Limited, 2015, pp. 1440.

[2]. P. Atzeni. "Data Modelling in the NoSQL world" in Proceedings of the 17th International Conference on Computer Systems and Technologies, 2016, pp. 1-4.

[3]. A. Corbellini, C. Mateos, A. Zunino, D. Godoy, S. Schiaffino. "Persisting big data: The NoSQL landscape". Information Systems, vol. 63, pp. 1-23, Jan. 2017.

[4]. R. Cattell. "Scalable SQL and NoSQL data stores". ACM SIGMOD Record, vol. 39, pp. 12, 2011.

[5]. J. Bhogal, I. Choksi. "Handling Big Data Using NoSQL" in Proceedings IEEE 29th International Conference on Advanced Information Networking and Applications Workshops, 2015, pp. 393-398.

[6]. K. Srivastava, N. Shekokar. "A Polyglot Persistence approach for E-Commerce business model" in Proceedings International Conference on Information Science - ICIS 2016, 2016, pp. 7-11.

[7]. P. J. Sadalage, M. Fowler. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional, 2012.

[8]. R. De Virgilio, A. Maccioni, R. Torlone. "Model-driven design of graph databases" in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 8824. Springer International Publishing Switzerland, 2014, pp. 172-185.

[9]. C. de Lima, R.S. Mello. "Um Estudo sobre Modelagem Lógica para Bancos de Dados NoSQL". Departamento de Informática e Estatística – Universidade Federal de Santa Catarina, pp. 11-20, 2015.

[10]. C. Zdepski, T. A. Bini, S. N. Matos. "An Approach for Modeling Polyglot Persistence" in Proceedings International Conference on Enterprise Information Systems-ICEIS, 2018, pp. 120-126.

[11]. R. N. Pagani, J. a. L. Kovaleski, L. M. Resende. "Methodi ordinatio: A proposed methodology to select and rank relevant scientific papers encompassing the impact factor, number of citation, and year of publication". Scientometrics, vol. 105, pp. 2109-2135, 2015.

[12]. C. Zdepski, T. A. Bini, S. N. Matos. "New Perspectives for NoSQL Database Design: A Systematic Review". American Scientific Research Journal for Engineering, Technology, and Sciences (ASRJETS), vol. 68, pp. 50-62, May. 2020.

[13]. F. Bugiotti, L. Cabibbo, P. Atzeni and R. Torlone. "Database Design for NoSQL Systems" in Conceptual Modeling, vol. 8824. Springer International Publishing, 2014, pp. 223-231.

[14]. M. Y. Santos and C. Costa. "Data models in NoSQL databases for big data contexts" in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 9714. LNCS, 2016, pp. 475-485.

[15]. M. J. Mior, K. Salem, A. Aboulnaga, and R. Liu. "NoSE: Schema design for NoSQL applications". IEEE Transactions on Knowledge and Data Engineering, vol. 29, pp. 2275-2289, 2017.

[16]. P. Atzeni, F. Bugiotti, L. Rossi. "Uniform access to non-relational database systems: The SOS platform" in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 7328. LNCS, 2012, pp. 160-174.

[17]. P. Atzeni, F. Bugiotti, L. Rossi. "Uniform access to NoSQL systems". Information Systems, vol. 43, pp. 117-133, Jul. 2014.

[18]. E. Gallinucci, M. Golfarelli, and S. Rizzi. "Schema profiling of document-oriented databases". Information Systems, vol. 75, pp. 13-25, Jun. 2018.

[19]. A. Silberschatz, H.F. Korth, S. Sudarshan. Database system concepts. New York: McGraw-Hill, 2010, pp. 1376.

[20]. P. Rob, C. Coronel. Database Systems: Design, Implementation, and Management. Boston, MA: Course Technology Press, 2007, pp. 720.

[21]. R. Ramakrishnan, J. Gehrke. Database management systems. New York: McGraw-Hill Education, 2003, pp. 1065.

[22]. P.P.S. Chen. "The Entity-Relationship Model-Toward a Unified View of Data". ACM Transactions on Database Systems, vol. 1, pp. 9-36, Mar. 1976.

[23]. Apache Software Foundation. "Apache Cassandra". Internet: http://cassandra.apache.org/, [Mar. 04, 2019].

[24]. MongoDB. "MongoDB". Internet: https://www.mongodb.com/, [Fev. 04, 2019].

[25]. OrientDB An SAP Company. "OrientDB". Internet: https://orientdb.com/, [Sep. 12, 2019].

[26]. Elasticsearch B.V. "Elasticsearch". Internet: https://www.elastic.co/pt/, [Sep. 12, 2019].