A Big Data Modeling Methodology for NoSQL Document Databases

Gerardo ROSSEL, Andrea MANNA
Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación. Buenos Aires, Argentina
grossel@dc.uba.ar, amanna@dc.uba.ar

In recent years, there has been an increasing interest in the field of non-relational databases. However, far too little attention has been paid to design methodology. Key-value datastores are an important component of a class of non-relational technologies that are grouped under the name of NoSQL databases. The aim of this paper is to propose a design methodology for this type of database that allows overcoming the limitations of the traditional techniques. The proposed methodology leads to a clean design that also allows for better data management and consistency
Keywords: *NoSQL, Document Databases, Conceptual Modeling, Data Modeling, NoSQL Database developing.*

1 Introduction

The need for analysis, processing, and storage of large amounts of data has led to what is now called Big Data. The rise of Big Data has had strong impact on data storage technology. The challenges in this regard include: the need to scale horizontally, have access to different data sources, data with no scheme or structure, etc. These demands, coupled with the need for global reach and permanent availability, gave ground to a family of databases, with no reference in the relational model, known as NoSQL or "Not Only SQL".

The NoSQL databases can be classified by the way they store and retrieve the information [1][2]:

- Key-Value databases.
- Document databases.
- Column Families databases.
- Graph Databases.

The development of conceptual modeling and general design methodology associated with the construction of NoSQL databases is at an early stage [SS17]. of data modeling is to highlight in [3]: "*Data modelling has an impact on querying performance, consistency, usability, software debugging and maintainability, and many other aspects"*

There are previous works on development methodologies we can cite, like the BigData Apache Cassandra methodology, proposed by Artem Chebotko [4][13]. It uses the Entity Relationship Diagram as a conceptual model, but it is oriented to a specific engine, Apache Cassandra. Thus, it is not generic and does not adapt to a design of other NoSQL Databases. Another proposal using a conceptual model for the design of NoSQL is described in [5]. It suggests the use of the various NoSQL databases common features to obtain a general methodology, in which an abstract data model called NOAM is used for conceptual data modeling. Such data model is intended to serve all types of NoSQL databases using a general notation.

Recently, an attempt to generate a universal modeling methodology adapted to both relational and non-relational database management systems was also presented, on the grounds of overcoming the constraints that the entity relationship model has, according to the author [6].

The use of conceptual modeling is also proposed in [7], although the background is not sufficiently studied, such as our work on interrelation of documents and the relationship between them and the conceptual model [8]. They use UML as a tool for the realization of the conceptual model and simple rules to transform it into a

logical model using UML stereotypes.
These efforts show that traditional methodologies and techniques of data modeling are insufficient for new generations of non-relational databases.
It is therefore necessary to develop modeling techniques that adapt to these new ways of storing information. In this sense, this paper will provide the tools to solve these limitations for document database design. As indicated in [14] the methodology should allow: "*describe the data-model precisely*"
The rest of the paper is organized as follows: Section 2 outlines the definition of document database; Section 3 describes the main elements of the methodology and phases of document database develop; Section 4 presents the logical design using the document interaction diagram or DID by extending our previous work: moving from logical to physical model using *JsonSCHEMA* is presented in section 5 and finally Section 6 presents conclusions and future work.

2 Document Databases

The proposed methodology is oriented to the design of databases based on documents. A document is a collection of field name and value pairs. The values can be a simple atomic value or a complex structure such as lists of values, another document or lists of child documents.
NoSQL documents are generally referred to as schema-less, which seems to suggest that it is not necessary to make a model before the development starts. The fact that the structure of the data does not need to be defined in advance has many advantages for prototyping or exploratory development, but as data expands and the applications make use of them, the necessity to have them organized in some way arises. In that sense it is more appropriate to say that they are agnostic with respect to the internal structure of the data. It is, therefore, necessary to make a design of the data organization.
to as *schema-less*, which seems to suggest that it is not necessary to make a model before the development starts. The fact that the structure of the data does not need to be defined has a priori many advantages for prototyping or exploratory development, but as data expands and the applications make use of them, the necessity to have them organized in some way arises. In that sense it is more appropriate to say that they are *agnostic* with respect to the internal structure of the data. It is, therefore, necessary to make a design of the data organization.

3 Methodology

The proposed design methodology has as its starting point the conceptual model, that can be considered as a high-level description of data requirements. Conceptual modeling is usually performed using some form of entity-relationship diagram ([9]) for conceptual class diagram in UML. Conceptual modeling is intended to describe the semantics of software applications.
In traditional relational database design methodologies, conceptual modeling gave way to a logical design that was later transformed into a physical design. It operates by transforming models from higher levels of abstraction to a model that maps directly into the structures of the database.
Phases of proposed NoSQL document database develop consists of high or conceptual level (conceptual model and access patterns), logical level (types of documents, interrelations and specifications), and physical design in steps like phases of traditional relational database.
In the high-level phase, a conceptual data model is developed in a similar way to the design of relational databases. In the current era, with the emergence of Big Data, the need for conceptual modelling is even more important than before.
As a tool of specification and communication with the other phases, the entity relationship diagram is used (ERD) [9]. In this phase, it is also necessary to

specify the query patterns that have been obtained in the analysis requirements. Query patterns can be specified in natural language or in a more formal language like ERQL [10].

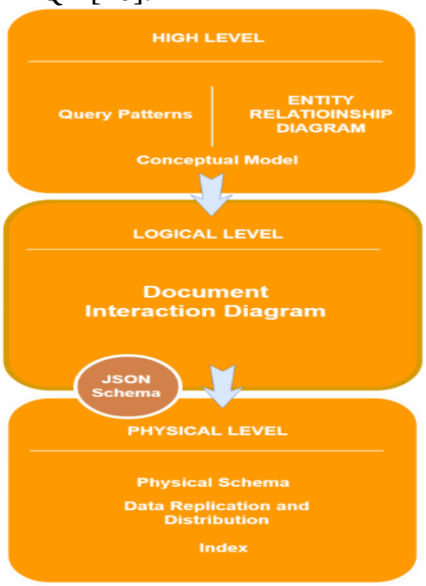

Fig. 1. Phases

The Logical Level is the heart of the proposed methodology, in which the types of documents and their interrelationships are established. To represent the logical design, we use a new type of diagram that extends the ERD and that we call document interrelation diagram (DID)[8]. Each type of document is later specified using *JSONSchema*.

There are two ways of relating documents: referencing or embedding. The ability to embed documents allows the designer to store related data as a simple document.

In this way, what is called impedance mismatch can be solved (that is, the difference between the structures of data in memory and the way in which they are stored) [2]. The decision whether to embed or reference is a design decision that is guided by query patterns.

The last phase of our methodology is the analysis and optimization of a logical model to produce a physical data model. In this phase, topics such as index creation, sharding, data distribution, and adapting the data types to the software of the database are considered. The utilization of JSONSchemes is essential in this regard.

4 Logical Design

The more important task in this phase is the development of the document interrelation diagram. The DID represents the logical model for a document-based database that captures the classes or types of documents, their structure and interrelation. The documents can be grouped into different classes. Each database uses its own terminology as collections in MongoDB or tables in RethinkDB. we use classes or document types as terminology to indicate a group of documents with similar characteristics.

In the DID each entity of the ERD corresponds to a class or document type, unless it is specifically indicated that this entity will have an independent existence as document type.

In order to exemplify, the entity relationship diagram of Fig.2 will be used. This ERD represents, in a simplified way, the conceptual model of a database that stores orders, products and customers.

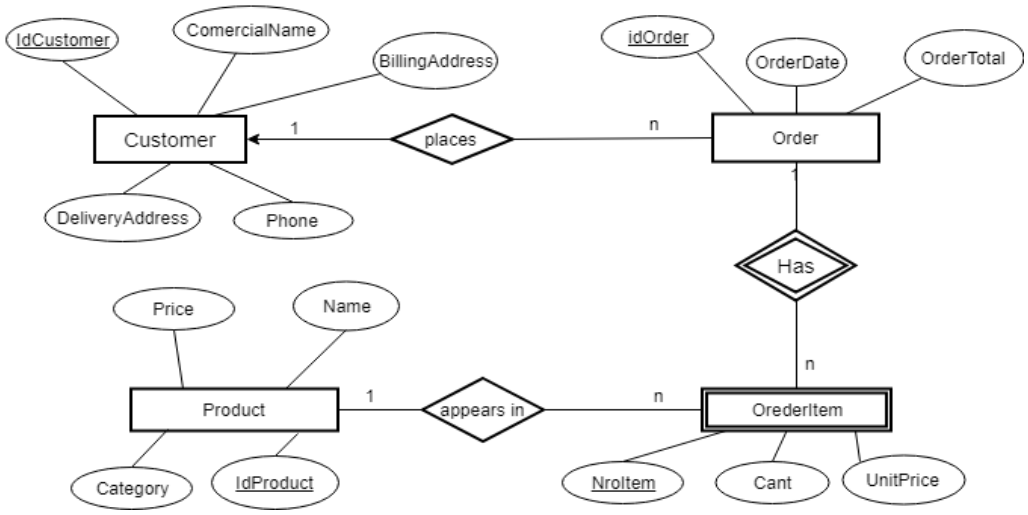

Fig. 2. ERD

The entities *Customer*, *Order* and *Product* becomes three document types. *OrderItem* is a weak entity so it is a special case.

To complete the document interaction diagram, it is necessary to decide how the interrelationships will be solved. For this it is necessary to consider the query patterns.

Let's start with the relationship "places". Many design decisions are possible:

- Reference from both sides
- Embed on both sides
- Reference from Order and embed from Customer (or vice versa)
- Embed partially from one side and reference from the other.
- Embed partially from both sides
- Embed total / partial or reference from one side and do nothing from the other

Fig. 3 shows how the reference of both sides is specified while Fig. 4 does the same with embedding of both sides. The arrow indicates reference and curly brackets indicates embedding [8].

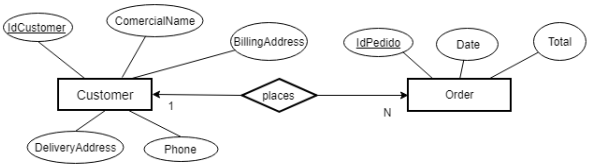

Fig. 3. DID: reference

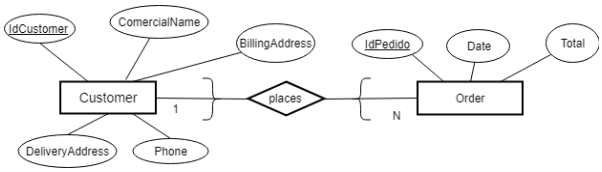

Fig. 4. DID: embedding

Embedding simplifies access by minimizing the number of times it should be read from persistent storage. The goal is to keep data that is frequently used together in one document. Although it might be better for a document not to incorporate all the information of the document with which it is interrelated, but only the necessary information that arises from the query patterns.

Suppose that the query patterns indicate that a common way of access to the data is the printing of the order for which the customer's commercial name and shipping address are needed, in addition to all the associated order items. Also suppose that you want to get the dates of the orders made and the total amounts of the same. If the interrelation is solved using only references, the applications are being forced to make several roundtrips to server for to obtain the necessary data. In these cases, a partial embedding can be a better solution.

Fig.5 shows how partially embed is represented. It is necessary to indicate which
fields of the other entity that will be embedded.

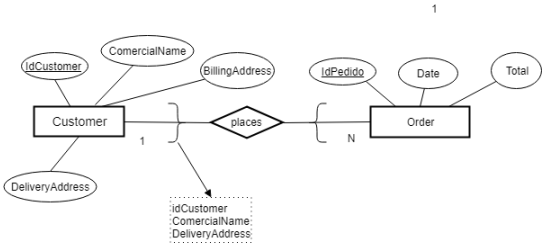

Fig 5 Embed Partially

Weak entities generally form an aggregate with the strong entity that determines them. It is the case of *ItemOrder* and *Order* in which *Order* can be considered as an aggregate or "a collection of related objects that we wish to treat as a unit" [1].
The simplest way to deal with this is to embed the weak entity in the type of document generated by the strong entity. It is also necessary to indicate that the weak entity will only have an embedded existence, which is done by placing a cross on it as shown in Fig. 6
The cross over any entity indicates that it is not generating a type of document that will be stored independently.

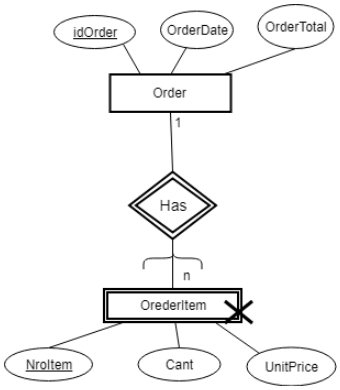

Fig. 6. DID: week entity

Although *ItemOrder* entity does not generate a document type, it has an interrelation with the *Product* entity that must be resolved in the logical model. The product information needed in the *ItemOrder* will depend on the domain over which the model was made and what are the access patterns. In this case, it can be assumed that only the name of the product is needed, for which we partially embed the name of the product in the item. When embedding the *ItemOrder* in Order it is embedded with everything it contains including references and embedded fields of other types of documents, in this case the name of the product. The final diagram is as in Fig. 7.

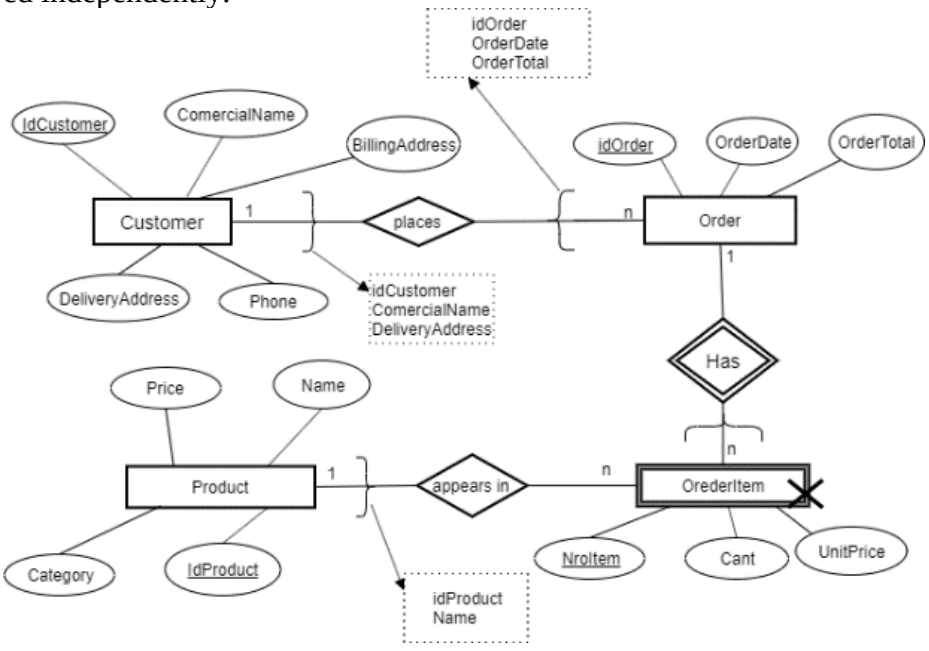

Fig. 7. DID

In some cases, it is not enough with the types of documents generated from the ERD to resolve all interrelationships. Assume the case of a database that must save user access to different modules and that a large number of daily accesses are made by each user. The most important query is to know on a given date which modules a user accessed. The ERD in Fig.8 is the conceptual data model.

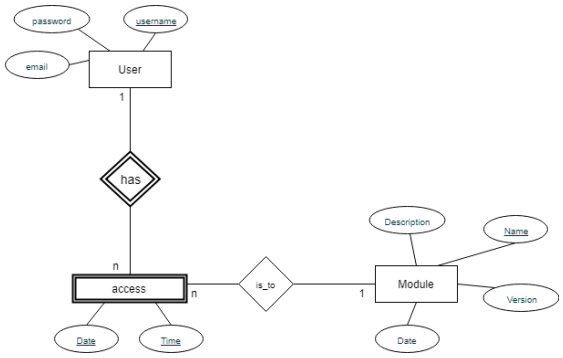

Fig. 8. ERD Users and Modules

How to resolve the interrelation between *User* and *Access*? At first glance it seems to be a case like that of the previous order and item. But there are two important differences that change the design decision:

1. The immutability or not of the data: In the previous case, once the order has been sent to the client, the items can no longer be modified. However, in this new domain accesses are added frequently.
2. The volume of data: The items in an order have a limited amount of data. On the other hand, user accesses grow permanently and frequently.

In a document-based database the document is the unit of access, changes in their sizes may generate the need to reorganize the physical space where they are stored, if this is done very often there may be a degradation of performance.

The query patterns in the example indicate that, in general, accesses for a given date are consulted, so it would be a good design decision to divide the accesses by date. Also, once the date is finished, the accesses of the same are immutable. To have a document by date it is necessary to create an auxiliary document type. Fig. 9 shows how that document is specified.

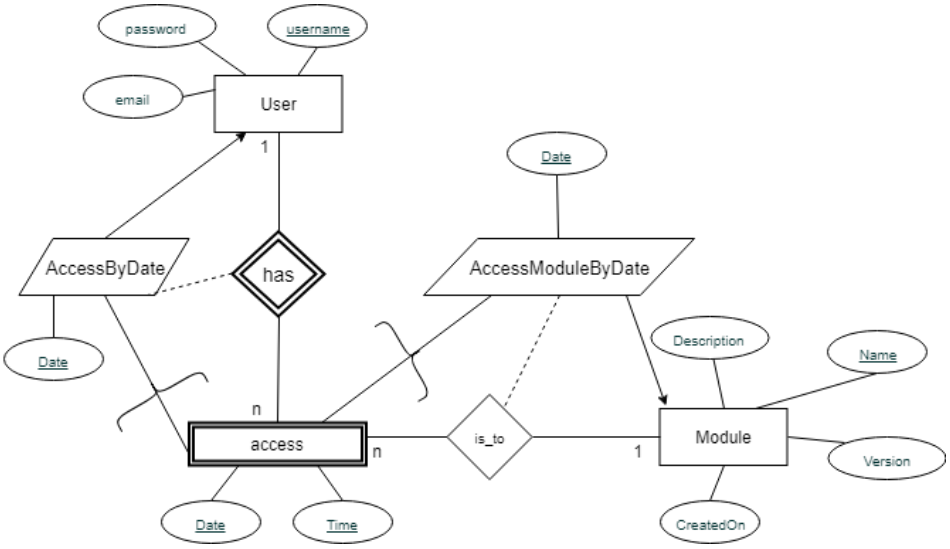

Fig. 9. DID: Partition

The new document that does not correspond to any entity of the ERD is drawn as a parallelogram with two inclined sides. It is also necessary to indicate which interrelation that document is representing that is achieved with a dotted line from the interrelation to the symbol of the intermediate document.

The auxiliary document has on one hand a reference to the user and on the other it embeds the accesses. The key will be the date and user id. We must explicitly mark as

a key the *Date* taken from the accesses to indicate that it is the partition key and therefore there is a single date per document, the user identifier does not need to indicate it since the arrow indicates reference to the key of the user and also the cardinality of the user-measurement relationship indicates that the measurements are of a single user. It is not necessary to keep the measurements as an independent document, so the cross is placed on that entity.

The extended entity relationship diagram also supports hierarchies between entities.

The hierarchies in the ERD can be with full or partial coverage, with overlapping or without overlapping. The possibility that documents of the same type have different schemes facilitates the design. We can generate a single type of document corresponding to the super-entity that also has the attributes of the sub-entities. For this, it is enough to indicate that the sub-entities do not generate a type of document as seen in the Fig. 10.

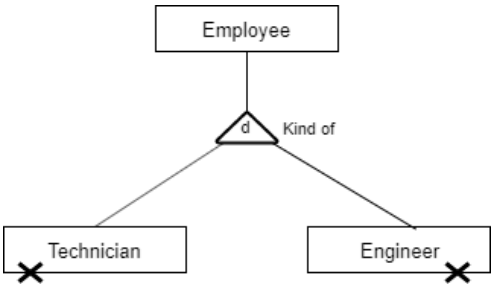

Fig. 10. DID Herarchies

Depending on the pattern of consultations, other decisions may be made:

- Mark the super-entity as not generating a document type and then generate one for each sub-entity. This is possible if the hierarchy has no overlap.
- Specify that both super-entity and sub-entities generate one document type each. Indicating which attributes would be placed in super-entity.

Another type of relationship that is necessary to model is ternary relationship.

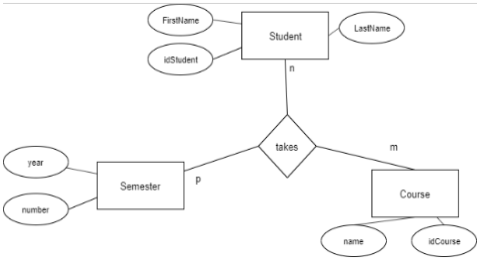

Fig. 11. ERD: ternary relationship

Suppose a ternary relationship between *Student*, *Semester* and *Course* entities. The cardinality in this case is n:m:p, for a student and a semester there are many courses he takes, a semester and a course has many students enrolled, for a course and a student can be many semesters where he takes it. The DER of Fig. 11 shows this relationship.

The most complex part is deciding how to model the relationship takes. The decision on how to model will, as always, depend on the query patterns. The basic case is to generate a type of document that simply contains the information of the relationship with the identifiers of each of the entities involved. To do this, an auxiliary document is drawn with the name of the new document type and a dotted line that binds it to the entity as seen in Fig.12.

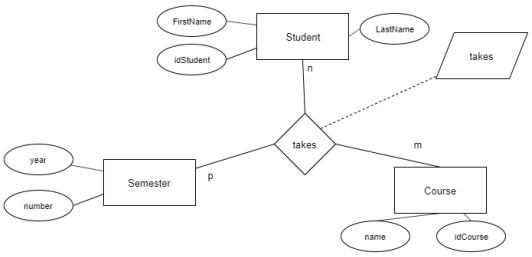

Fig. 12. DID: ternary relationship

That is the simplest model, but suppose that a very common query is to know which students are enrolled in a course in a semester, in fact you want to know first name, last name of them for a given course and semester. While the previous model

allows you to answer this query, you might decide to have a document type that stores the complete information to optimize access to it.

The semantics of this diagram (Fig. 13) are that only key attributes are added or that allow you to group data from another entity from those participating entities in the interrelationship that are not related by any link to the new document type.

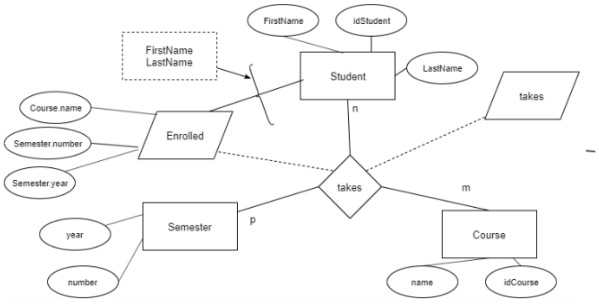

Fig. 13. DID: complex ternary relationship

One case to consider is when it becomes necessary to group multiple instances of an entity, by one or more attributes, into a single document. To exemplify let's assume a part of a DER where users and their searches are modeled.

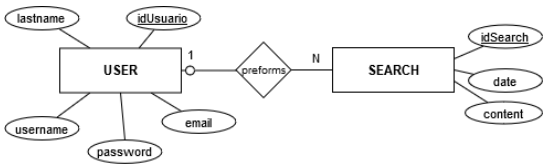

Fig. 14. DER: User/ search

The relationship between user and search can be modeled in various ways, either by embedding searches in the user or by referencing. The relationship could also be resolved by partitioning by user and date in the same way as shown in Figure 9 for user and access. Let's say that a very frequent query is to know the searches performed on a given date. The solution of partitioning by user and date is not efficient for this because access should be made for each user who has a search on that date. In this case, the ideal is to have a single document with all the searches for a date. This would involve grouping by the date attribute, i.e. generating a document for each date that has all searches. An auxiliary document should be created to save all searches with the date as key. The notation is similar to that seen before, although in this case the auxiliary document refers only to the entity on which it is being grouped.

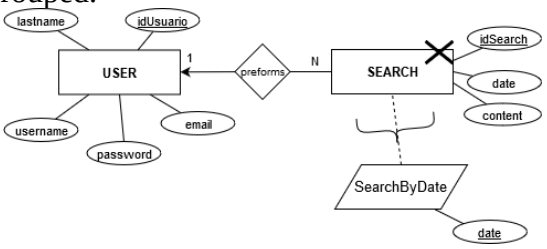

Fig. 15. DID: User/Search

Figure 15 shows the corresponding DID. Note that the reference from *Search* to *User* is important, because marking the entity as not generating a type of document would lose the relationship.

It is also possible to generate an intermediate document to resolve the relationship between *User* and *Search.* There would be data redundancy in favor of access speed. The complete DID is shown in Figure 16.

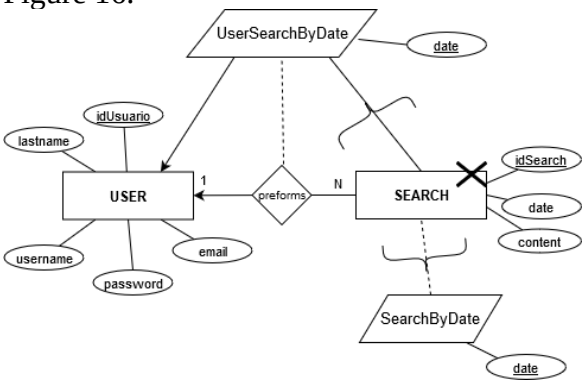

Fig. 16. DID: Complete User-Search

5 From logical to physical level

Upon completion of the development model interrelationship of documents, which is equivalent to logic design relational database, it continues with the physical design.

The physical design implies making decisions about specific aspects of implementation such as: data distribution, index generation, use of engine facilities of the selected database, etc.

Many document databases support indexes.

Index creation must be based on query patterns. It's about doing a trade-off so you don't have a few indexes that could lead to poor read performance, but not so many that affect the write performance.

The use of *JSONSchema* for a more detailed specification of each type of document facilitates decision making process and implementation. A *JSONSchema* is a JSON document which describes the structure of another document.

The steps to follow are as follows.

1. For each document type in the DID:
    a. Define the appropriate data types for each attribute
    b. Write the specification using *JSONSchema*.
2. For each query analyze the ease of documents to respond to it. Ideally a single access should be enough for the most used queries.

From the DID each type of document is mapped to a *JSONSchema* which allows to specify in detail the structure of each document. For example, the document type *AccessByDate* in Fig. 9 is mapped to the the following scheme:

```
{"title":"AccessByDate",
 "type":"object",
 "properties":{
            "userId":{"type":"integer"},
            "date":{"type":"string","format":"date"},
            "accesses":{ "type":"array",
            "items":{ "type":"object",
                        "properties":{
                                    "modulename":{"type":"string"},
                                    "timestamp":{"type":"string",
                                    "format":"date-time"}
                                    }
                        }
                }
    }
```

From the DID in Figure 16 JSONSchema will be generated for each of the following document types:

User: With the attributes in the diagram, specifying the type of each.

UserSearchByDate: having the *userid* and *date* as keys and a vector with that user's searches on that date.

SearchByDate: The key is the *date* and has a vector with the searches and in each the corresponding *userid*.

No other document types are generated. By indicating that an attribute is key we are claiming that it is unique and that it identifies each document, even though the database always generates an identifier attribute.

The flexibility of the *JSONSechema* to establish optional properties makes it an ideal tool for specifying document types of variable structure. In the case of hierarchies this facility is extremely useful because you can specify conditions for which an attribute exists or not. Looking at *JSONSchemas* it is possible to realize that in some case it is convenient to reserve space the same in such a way that the document does not resize it during its lifetime. If the document grows larger than the size allocated for it, the document may be moved to another location with the consequent input/output cost [12].

Some document-based databases have tools to validate if a document complies with a *JSONSchema.*

6 Conclusions

A methodology that allows obtaining a detailed design from a conceptual model has been presented. This work extends and completes previous work on document modeling in the design process.

The proposal presented allows flexibility to establish detailed design decisions. There is not currently, to the best of our knowledge, complete methodology such as that presented for document-based databases that have the same level of flexibility and specification capability.

The presented methodology was used successfully in several developments using different database engines. In future work we plan to report in detail the cases of success in the use of this methodology.

References

[1] Adam Flowler, “The State of NoSQL”, 1st edition, 2016

[2] Pramod J. Sadalage, Martin Fowler, “NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence”, *Addison-Wesley*

[3] Gómez, P., Casallas, R., & Roncancio, C. (2016). “Data schema does matter,

even in NoSQL systems!" 2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS), 1-6.

[4] Artem Chebotko, Andrey Kashlev, Shiyong Lu, "A Big Data Modeling Methodology for Apache Cassandra", IEEE International Congress on Big Data (BigData'15), pp. 238-245, New York, USA, 2015.

[5] Francesca Bugiotti, Luca Cabibbo, Paolo Atzeni, Riccardo Torlone. "Database Design for NoSQL Systems". *International Conference on Conceptual Modeling*, pp. 223 - 231 Atlanta, USA, Oct 2014.

[6] Ted Hills, "NoSQL and SQL Data Modeling", *Basking Ridge, NJ: Technics Publications*, 2016

[7] Shin, K & Hwang, C & Jung, H. (2017). "NoSQL database design using UML conceptual data model based on peter chen's framework". International Journal of Applied Engineering Research. 12. 632-636

[8] Gerardo Rossel, Andrea Manna, "Diseño de Bases de Datos Basadas en Documento: Modelo de Interrelación de Documentos" *XIII Workshop Bases de Datos y Minería de Datos. Congreso Argentino de Ciencias de la Computación* CACIC 2016 San Luis Argentina..

[9] Peter P. S. Chen, "The entity-relationship model: toward a unified view of data", *Proceedings of the 1st International Conference on Very Large Data Bases*, ACM, New York, NY, USA, 1975.

[10] M. Lawley and R. W. Topor, "A query language for EER schemas," in *Proceedings of the 5th Australasian Database Conference*, 1994, pp.292–304.

[11] Storey, Veda & Song, Il-Yeol. (2017). Big data technologies and management: What conceptual modeling can do. Data & Knowledge Engineering. 10.1016/j.datak.2017.01.001.

[12] Dan Sullivan. 2015. NoSQL for Mere Mortals (1st. ed.). Addison-Wesley Professional

[13] Jeff Carpenter & Eben Hewitt. (2020). Cassandra: The Definitive Guide (3st. ed.). O'Reilly Media, Inc.

[14] Pivert, Olivier. NoSQL Data Models: Trends and Challenges. 2018. Wiley-ISTE

Gerardo ROSSEL graduated as Ms. Sc. in Computer Science from the Faculty of Exact and Natural Sciences of the University of Buenos Aires. He has a Doctor's degree from the National University of Tres de Febrero. At present, he is an assistant lecturer at Computer Department of FCEyN UBA. He has more than 20 years of experience in software industry and is Chief Scientist of UpperSoft software company. Her specific area of competences is in Databases, NoSQL, Data Science, Machine Learning, Patterns and Software Architectures, Epistemology and Philosophy of Computer Science. He is co-author of book "*Algoritmos, Objetos y Estructuras de Datos".* He has published several papers in national and international conferences and journals. He was a member of the International Program Committee of several international conferences.

Andrea MANNA, graduated from the Faculty of Exact and Natural Sciences of the University of Buenos Aires in 2000. She got the title of Ms. Sc. of Computer Science. At present, she is assistant lecturer in the Faculty of Exact and Natural Sciences of the University of Buenos Aires. She has been working in the software industry since 1995. She is Chief Software Architect of UpperSoft Sofware Company and work in software development for more than twenty years. She is co-author of book "*Algoritmos, Objetos y Estructuras de Datos".*