Document Data Modeling: A Conceptual Perspective

David Chaves(✉) and Elzbieta Malinowski

Department of Computer Science and Informatics, University of Costa Rica, San José, Costa Rica
{david.chavescampos, elzbieta.malinowski}@ucr.ac.cr

Abstract. The growing availability of data and the increased popularity of NoSQL databases, that support the idea of managing unstructured or semi-structured data, motivate implementers to skip the phase of a conceptual view of data. However, document data stores belonging to the NoSQL group show a clear tendency of looking for some common feature among documents creating collections. This aspect motivates us to propose a model for the conceptual representation of a document data store based on UML class diagrams and mapping rules for its implementation. We also include a case study using Twitter data and show implementation using three data stores: MongoDB, CouchDB, and ArangoDB.

Keywords: Document data stores · NoSQL databases · Conceptual data modeling

1 Introduction

References to big data mention the common characteristics of being unstructured. This opened the possibility of skipping a conceptual modeling phase. On the other hand, the benefits of using database conceptual models have been acknowledged for decades; however, the conceptual design domain for NoSQL repositories is still at a research stage leading to poorly-designed systems. The modeling does not aim to enforce structure over data, but it helps to understand how data is organized for analysis [1].

Document data stores are the second most popular data model [2] and are similar to a key-value model with a difference of having self-describing, hierarchical, and examinable value. The usual practice in document datastores is to skip the conceptual design taking directly implementation aspects into account. Even though this practice can give positive results for small systems, it becomes more difficult for more complex ones.

In this paper, we propose an extension of UML class diagrams for representing document stores and mapping rules, showing the implementation in the three document stores [2]. In our approach, we look for simplicity and for bridging the gap between academics and practitioners.

This paper is organized as follows: Sect. 2 refers to related works, Sect. 3 introduces our proposal for conceptual modeling. Sect. 4 shows mapping rules for the implementation and Sect. 5 includes a case study using the proposed conceptual

T. Welzer et al. (Eds.): ADBIS 2019, CCIS 1064, pp. 19–27, 2019.
https://doi.org/10.1007/978-3-030-30278-8_3

model and its implementation in document stores; lastly, Sect. 6 gives conclusions and future work.

2 Related Work

Currently, there are few systematic studies on data modeling for NoSQL databases, e.g., [1, 3]. Some works propose particular solutions that can be used for conceptual modeling of NoSQL databases [4, 5]; however, since the complexity of these approaches is high, they can be difficult to infiltrate into real-world applications. On the other hand, several studies refer explicitly to modeling documents in MongoDB using UML notation [6]; other approaches refer to the JSON format to represent the documents for different NoSQL databases [7]. Others develop automatic tools to map from JSON [8] and applying reverse engineering to already deployed systems [9].

In this work, we do not consider performance evaluation; however, this aspect is important and we plan to extend this research since different reports are contradicting. For example, [10] shows a better performance for indexed referenced documents compared to embedded ones, but the results of [11] demonstrate the opposite conclusions.

3 Conceptual Representation of Document Data Stores

Using a conceptual model in a document data store provides the advantage of representing data in a way that helps to understand, access, and analyze it from the beginning of the implementation process. Lacking the model forces the implementers to retain the details of data "structure" considering an implementation level that can be complex in the presence of different document collections.

Conceptual modeling is a product-independent design allowing its creators to focus on user requirements and implement the system, if adequate logical/physical mapping rules are established [12]. The proposed conceptual model uses the UML class diagram in a similar way as the conceptual modeling is done in the relational databases.

3.1 Document with Fields

A document is the main element and presents a set of data in an organized form, even though its structure can differ from other documents. Each document contains fields; one of them is reserved for document id. Since documents can have different fields, we propose to choose as a representative document the one that includes all fields indicating some fields as optional. Figure 1b shows an example where two fields are included in all documents, e.g., *movieId*, and *movieTitle*, and one is optional, e.g., *language*, indicating this by the symbol of "∼" before the name. Other fields group elements in an array, e.g., *genres*; this data type with its cardinalities is indicated in square parenthesis.

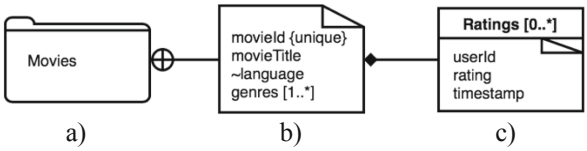


Fig. 1. Document representation: (a) collection, (b) document itself, and (c) an embedded one.

3.2 Document Collection

The collection represents a grouping of similar documents. Compared to relational databases, a document could correspond to a row and a collection of documents to a table. Figure 1a shows a graphical representation for a collection using the UML package. We use the symbol of contention relationship (⊕) to indicate that documents form part of a collection, i.e., its membership [13].

3.3 Embedded Documents

The field in the document can refer to another document forming nested documents. We propose two different UML notations to represent this: a composition relationship and an aggregation relationship. Figure 1c shows a general form for representing the composition relationship with an example of movies and their ratings. This embedded document includes the name, its multiplicity (0..* in Fig. 1c), and the specification of its fields. This kind of relationship is required when a nested document existence depends on its container document, e.g., movie rating is part of a specific movie. Our approach is different from [6] since the last one, additionally, requires class inheritance that increases unnecessarily the complexity of the model.

On the other hand, when a nested document depends on its container but, if necessary for the further extension, they can be converted to standalone documents, we propose to use the aggregation relationship (◊), e.g., the movie storyline is closely related but not strictly dependent to the movie.

3.4 Referenced Documents

When a collection is related to two or more other collections, it is necessary to define a relationship between them to avoid data repetition. To represent this relationship, we propose to use a bi-directional association as it can be seen in Fig. 2 (*Directed by* relationship). This representation includes multiplicity values in the (min, max) form to indicate the number of documents that should participate (min value) in the association and number of documents from one collection that can be associated with documents from another collection (max value).

Figure 2 shows two document collections representing movies and directors. Since a movie can be directed by one or more directors and the director can lead some other movies, the cardinality is many-to-many (indicated by the * symbol). Further, not all movies have a specified director (min value is 0), but all directors have associated at least one movie (min value is 1).

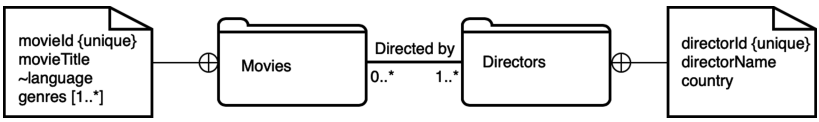


Fig. 2. An example of a referenced collection.

4 Mapping Rules

After outlining the conceptual proposal, we define the following mapping rules using the JSON markup language. It emulates an intermediate stage for the design of the data store, similar to the logical representation in relational databases that allows one to specify relations before their deployment in the particular DBMS. The translation from JSON to the physical level according to the specific system is a straightforward task and may consider other aspects, such as indexing, sharding, replication (if available), among other features. Furthermore, it is possible to automate the mapping process from UML to JSON based on already existing tools, such as crowd [8].

4.1 Document with Fields and Document Collection

To represent a document, we use the JSON specification as shown in Fig. 3 for a document conceptual representation in Fig. 1. Each field is represented in a key-value fashion with the key (field name) between quotes, followed by the associated value. In addition, to represent an array (*genres* in the figure), values separated by commas are included in square parentheses. Additionally, JSON file can include many documents arranged in an array forming a collection.

```
{"id": "321","movieId": "123","movieTitle": "Science", "language": "English",
"genres": ["Comedy", "Fiction"],
"Ratings": [{"userId": "453", "rating": "4","timestamp": "2017-11-09 T 11:20 UTC"},
            {"userId": "784","rating": "5","timestamp": "2016-10-08 T 13:40 UTC"}]}
```

Fig. 3. JSON file representing a document from Fig. 1.

4.2 Embedded Documents

The mapping of embedded documents is based on the commonly-known principles used for object-relational databases [14]. Even though document data stores do not belong to this group, general practice demonstrates the use of this mapping [13]. The following rules are applied considering the multiplicity shown on the conceptual level:

- (0..1) or (1..1): indicates the existence of none or only one embedded document; this document can be represented as such or its fields can be merged with the fields of the main document.
- (0..*) or (1..*): indicates the existence of none, one, or many embedded documents; these nested documents are organized as an array stored in one field of the main document. Each related document is an element of the array.

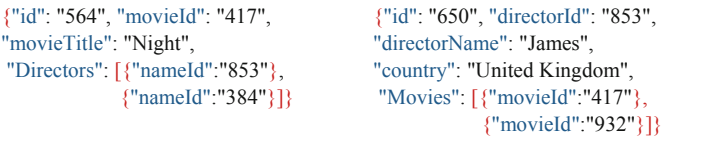
Notice that we do not consider a many-to-many relationship between main and embedded documents since it would indicate that some "external" documents are referencing an embedded document. We consider that if the embedded document must be accessed by other "external" documents, it should be modeled as a collection of documents with the corresponding association relationship.

4.3 Referenced Documents

Mapping of referenced documents, similar to the previous case, is based on known principles from object-relational databases [14] according to the following rules:

- One-to-one cardinality: the document key is included as a field in another document.
- One-to-many cardinality is mapped in two ways: each document on the n-side cardinality stores the key of the document from the one-side cardinality or each document on the one-side cardinality stores an array of keys from the n-side cardinality.
- Many-to-many cardinality is also be mapped in two ways: documents in one or both collections include an array of identifiers of corresponding documents from another collection.

Figure 4 illustrates the case of two documents with a many-to-many cardinality between collection of documents: Fig. 4a represents a movie with an array of corresponding director IDs, while Fig. 4b includes arrays with associated movie IDs.



a) Movie document with several directors.

b) Director document with several movies.

Fig. 4. Implementation in JSON for referenced many-to-many documents.

5 Case Study

After presenting the conceptual model and its mapping rules, we propose a case study to demonstrate the use of the notation and its deployment according to [15] guidelines.

5.1 Case Delimitation

Objective. We evaluate the application of the proposed notation in different open source products for document storing. The main hypothesis is that the implementation is common among systems without incurring to particular additional requirements that could change the proposed conceptual schema.

Related Cases. Many academic works include examples of using document stores as relational implementations (e.g. [3, 13]) or modeling, particularly, in MongoDB [1].

Methodology. Using a qualitative approach, we design a conceptual model for a document data store and implement it in selected data stores. We overview the raw data [16] to define requirements for data store design. Afterward, we develop the schema (Sect. 3) and map it (Sect. 4), showing the implementation differences.

Limitations. This model does not include some possible optimizations for each system, such as indexes and buckets that could be included after the model is deployed.

5.2 Conceptual Representation

Considering Twitter messages, it is possible to identify data referring to users and messages, leading to conceptual schema showed partially in Fig. 5.

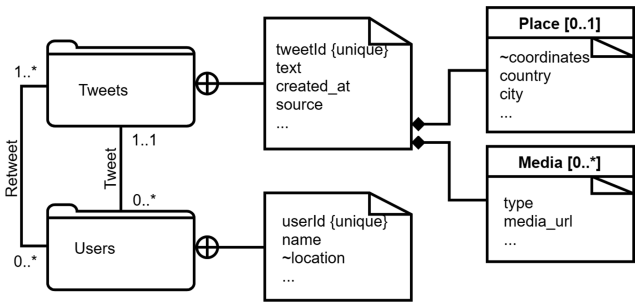


Fig. 5. An extract of a conceptual representation of Twitter data.

As can be seen in Fig. 5, the *Tweets* and *Users* collections include their own fields, e.g., *tweetId*, *text* or *userId*, *name*, among others. Furthermore, the tweet document refers to two optional embedded documents: *Place* that may appear at most one time and *Media* that may consist of several documents representing different media types. In addition, the *Tweets* collection is related to the *Users* collection through two relationships: *Tweet* and *Retweet*. According to shown multiplicity, no all users publish tweets, but every published tweet must have an associated (and only one) user. The *Retweet* association shows the possibility that a message can be republished by many users and these users can republish many messages.

5.3 Applied Mapping for Implementation

MongoDB Implementation. Figure 6 shows an example of a tweet in MongoDB mapped according to rules in Sect. 4. Fields of embedded document *Place* are included in the main document (lines 5–8) with an optional field (coordinates, line 5). The composition *Media* object is embedded as an array (lines 9–10) considering its

multiplicity (only two elements are shown). In addition, *userId* (line 3) represents a one-to-many association relationship *Tweet* between *Tweets* and *Users* collections.

```
1  { "_id":ObjectId("5b21705709429256cd1bc75c"),
2   "tweetId":NumberLong("934592617924276225"),
3   "userId":NumberLong("839337948323713028"),
4   …,
5   "Place_coordinates":{"coordinates":[40.417416, -79.993930],"type":"Point"}
6   "Place_id":"5db3a841345615",
7   "Place_country":"United States",
8   "Place_city":"Pittsburgh, Pennsylvania",
9   "Media":[ { "id":NumberLong("934421567307747328"),"type":"photo", …},
10            { "id":NumberLong("908781363528110080"),"type":"photo", …}]}
```

Fig. 6. Example in MongoDB of embedded (*Place* and *Media*) and referenced (*Users*) documents from Fig. 5.

CouchDB Implementation. This store does not include the concept of collection; therefore, it is necessary to insert the documents in the same database with a field identifying its type, e.g., *Tweets* or *Users*. This adaptation is minimal and does not affect the conceptual model. Figure 7 shows an example of a tweet document in CouchDB with the *Retweet* relationship. This many-to-many cardinality is shown as an array of users (the field *retweetUserId*, line 8). Also, we include the field *type* (line 3) to define its collection.

```
1  {"_id": "339f175188d7c1d587ab41af873b9fcb",
2   "_rev": "1-2a34bf1359537ef283dcc2633a4395fc",
3   "type": "Tweets",
4   "tweetId": "934592567584411649",
5   "userId": 157772888,
6   "text": "RT @mnrothbard: https://t.co/aoJqK0eJz7",
7   "created_at": "Sun Nov 26 01:20:06 +0000 2017",
8   "retweetUserId": [{157772888}, {2367997502}],
9   …}
```

Fig. 7. Example in CouchDB of documents with many-to-many cardinality.

ArangoDB Implementation. ArangoDB supports a user-defined unique *_key* to identify each document. In addition, it includes the *_id* field, which is the combination of the collection name and the document key (Fig. 8, lines 1 and 2). Particularly, both CouchDB and ArangoDB include a revision value (Fig. 7, line 2 and Fig. 8, line 3) in order to support concurrency control, which does not affect the conceptual level design.

```
1  { "_key": "2816",
2    "_id": "tweets/2816",
3    "_rev": "_X-RUUJG-_x",
4    "tweetId":"934592680847327232",
5    "userId": 159104114,
6     …}
```

Fig. 8. Example in ArangoDB with keys combination and revision value support.

6 Conclusions and Future Work

The growing use of NoSQL databases and the increasing amount of data in these repositories make the understanding of their "structure" increasingly difficult. The affirmation that NoSQL databases, in particular, document data stores, manage semi-structured data opens the possibility of skipping the conceptual phase; this phase is important since it helps in understanding the nature of data and relationships existing between different elements. As a consequence, it facilitates the expression of queries to analyze data. Although it is well-known that documents can have different fields, there is a clear tendency to create a document collection in order to group "similar" documents.

In this paper, we propose the use of UML class diagrams to represent document stores on a conceptual level. We also include mapping rules that facilitate the document data stores implementation, showing examples of three data stores. Even though the proposed model and mapping rules can be extended, we expect that the simplicity of this conceptual proposal may be appealing to a wide forum of document data stores implementers.

References

1. Vera, H., Boaventura, W., Holanda, M., Guimarães, V., Hondo, F.: Data modeling for NoSQL document-oriented databases. In: 2nd Annual International Symposium on Information Management and Big Data, Cusco (2015)
2. DB-Engines: DB-Engines Ranking category, May 2019. https://db-engines.com/en/ranking_categories. Accessed 21 May 2019
3. Imam, A., Basri, S., Ahmad, R., Aziz, N., González-Aparicio, M.: New cardinality notations and styles for modeling NoSQL document-store databases. In: IEEE Region 10 Conference (TENCON), Malaysia (2017)
4. Abdelhedi, F., Brahim, A., Atigui, F., Zurfluh, G.: MDA-based approach for NoSQL databases modelling. In: International Conference on Big Data Analytics and Knowledge Discovery, Lyon (2017)
5. Bugiotti, F., Cabibbo, L., Atzeni, P., Torlone, R.: Database design for NoSQL systems. In: Yu, E., Dobbie, G., Jarke, M., Purao, S. (eds.) ER 2014. LNCS, vol. 8824, pp. 223–231. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12206-9_18
6. Poveda, J.: Propuesta de Notación Gráfica para el Modelo Orientado a Documentos de MongoDB. Universidad Distrital Francisco José de Caldas, Bogotá (2013)
7. Zola, W.: 6 Rules of Thumb for MongoDB Schema Design, 29 May 2014. https://bit.ly/2FUb3cp. Accessed 21 May 2019
8. Braun, G., Gimenez, C., Fillottrani, P., Cecchi, L.: Towards Conceptual Modelling Interoperability in a Web Tool for Ontology Engineering in Simposio Argentino de Ontologías y sus Aplicaciones, Córdoba (2017)
9. Hernández, A., Feliciano, S., Sevilla, D., García-Molina, J.: Exploring the visualization of schemas for aggregate-oriented NoSQL databases. In: Proceedings of the ER Forum 2017 and the ER 2017 Demo track, Valencia (2017)
10. Reis, D.G., Gasparoni, F.S., Holanda, M., Victorino, M., Ladeira, M., Ribeiro, E.O.: An evaluation of data model for NoSQL document-based databases. In: Rocha, Á., Adeli, H., Reis, L.P., Costanzo, S. (eds.) WorldCIST'18 2018. AISC, vol. 745, pp. 616–625. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77703-0_61

11. Calvo, K., Durán, J., Quirós, E., Malinowski, E.: MongoDB: alternativas de implementar y consultar documentos. In: IX Congreso Internacional de Computación y Telecomunicaciones, COMTEL, Lima (2017)
12. Gulden, J., Reijers, H.: Toward advanced visualization techniques for conceptual modeling. In: Proceedings of the CAiSE 2015 Forum at the 27th International Conference on Advanced Information Systems Engineering, Stockholm, pp. 33–40 (2015)
13. Lima, C., Santos, R.: A workload-driven logical design approach for NoSQL document databases. In: 17th International Conference on Information Integration and Web-based Applications & Services, Brussels (2015)
14. Dietrich, S., Urban, S.: An Advanced Course in Database Systems: Beyond Relational Databases. Prentice Hall, New Jersey (2004)
15. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. **14**, 131–164 (2009)
16. Scott, J.: Archive Team: The Twitter Stream Grab, 6 December 2012. https://archive.org/details/twitterstream. Accessed 21 May 2019