Data schema does matter, even in NoSQL systems!

Paola Gómez, Rubby Casallas, Claudia Roncancio

Data Schema Does Matter, Even in NoSQL Systems!

Paola Gómez
Université Grenoble Alpes, LIG
France
paola.gomez-barreto@imag.fr

Rubby Casallas
Universidad de los Andes
Bogotá, Colombia
rcasalla@uniandes.edu.co

Claudia Roncancio
Université Grenoble Alpes, LIG
France
Claudia.Roncancio@imag.fr

***Abstract*—A Schema-less NoSQL system refers to solutions where users do not declare a database schema and, in fact, its management is moved to the application code. This paper presents a study that allows us to evaluate, to some extent, the data structuring impact. The decision of how to structure data in semi-structured databases has an enormous impact on data size, query performance and readability of the code, which influences software debugging and maintainability. This paper presents an experiment performed using MongoDB along with several alternatives of data structuring and a set of queries having increasing complexity. This paper introduces an analysis regarding the findings of such an experiment.**

I. Introduction

The data management landscape has become extremely rich and complex. The large variety of requirements in current information systems, has led to the emergence of many heterogeneous data management solutions[1]. Their strengths are varied. Among the most outstanding are data modeling flexibility, scalability and high performance in cost effective ways. It is becoming usual, that an organisation manages structured, semi-structured and non structured data: Relational as well as NoSQL systems are used [1]. NoSQL systems refer to a variety of solutions that use non-relational data-models and "schema-free data bases" [2]. NoSQL systems are commonly classified into column oriented, key-value, graphs and document oriented. There is no standard data model and few formalization efforts. Such models combine concepts used in the past by non-first normal form relational models, complex values and object oriented systems [3]. Concerning transactional support and consistency enforcement, NoSQL solutions are also heterogeneous. They rank from full support to almost nothing which implies delegating the responsibility to the application layer. Moreover, the absence of powerful declarative query languages in several NoSQL systems, increases the responsibility of the developers.

Data modelling has an impact on querying performance, consistency, usability, software debugging and maintainability, and many other aspects [4]. What would be *good* data structuring in NoSQL systems? What is the price to pay by developers and users for flexible data structuring? Is the impact of data structuring on querying complexity high? There are of course no definitive and complete answers to these questions.

This paper presents the analysis of the impact of data structuring regarding *data size* and, for representative queries, *performance*. It compares structuring alternatives through an experiment by using different schemes, which vary mainly their embedded structure, and many queries with different access patterns and increasing complexity. During the experiment we also analyse the impact of using indexes in the collections. Furthermore, we discuss the results, of which some are intuitively expected and others point out unexpected aspects. We performed the experiment and analysis using MongoDB, a document oriented database, which is a popular open source system to store semi-structured data. It is flexible, as it allows many data structuring alternatives, and it does not require an explicit data base schema definition nor are integrity constraints enforced.

The rest of the paper is organized as follows. Section II provides a background on MongoDB. In Section III, we define different representative "document schemes" for the same data. They are used in the experience presented in Section IV. We performed a systematic evaluation that considered usual access patterns with increasing complexity. In Section V, we discuss the results and how the experiences confirm or not the expected impact of the document scheme choices. Related works are briefly described in Section VI. Our conclusions and research perspectives are presented in Section VII.

II. Document DB background

In this paper, we focus on the use of the MongoDB document db to store semi-structured data. Its "data model" is based on JSON[2]. Therefore, the supported data types are also largely used by other systems [5]. In Section II-A, we provide an overview of Mongo's data model.

Concerning system aspects, MongoDB supports BSON serialization, indexes, map/reduce operations, master-slaves replication and data sharding. These features contribute to provide horizontal scalability and high availability. The analysis of all these features is out the scope of this paper. We will mainly focus on query access patterns and the impact of the data structures on their performances. In Section II-B, we introduce Mongo's query capabilities.

[1]Great map on https://451research.com/state-of-the-database-landscape

[2]https://www.mongodb.com/json-and-bson

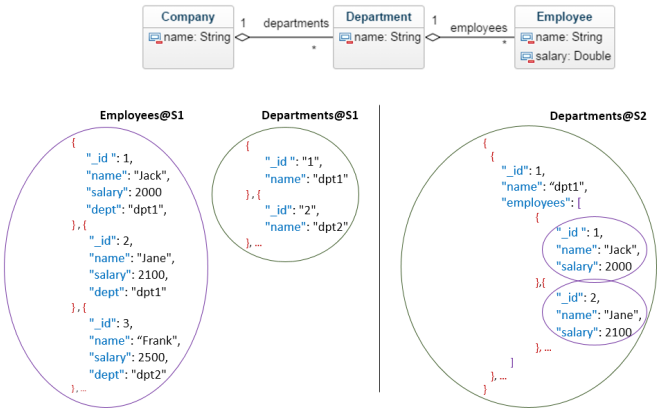


Fig. 1: Embedded and Referenced documents

A. Data structuring capabilities

MongoDB does not support an explicit data base schema that has to be created in advance. It provides data modelling flexibility, as users can create collections including BSON documents that have the same or different structures. A document is a structure composed by a set of `field:value` pairs. Any document has the `_id` identifier, which value is either automatically assigned by the system or explicitly given by the user, such as in our example. The type system includes atomic types (string, int, double, boolean), documents and arrays of atomic values or documents. Integrity constraints are not supported.

Note that Mongo's type system supports two ways to relate documents, *embedding* or *referencing* . The first one allows one or many documents can be embedded by several other documents. The second one refers documents using one or several fields. Figure 1 proposes two choices to represent the one-to-many relationship between employees and departments. Employees could be "completely" embedded in the `Departments@S2` collection, or each employee belongs to `Employees@S1` can refers the correspondent department using the field `dept`.

This type system opens the door to many "modeling" possibilities that sometimes are compared to the normalized and de-normalized alternatives of the relational data model [4]. As will be evident in the following sections, even if NoSQL trends are prone to using schema-less data bases, there is an "implicit" schema and its choice is important.

B. Query language

MongoDB[3] provides a document-based query language to retrieve data. A query is always applied on a concrete document collection. Filters, projections, selections and other operators can be used to retrieve particular information for each document. MongoDB provides many operators in order to compare data, find out the existence of a particular field or find elements within arrays.

[3]Version 3.1.9

The implementation of a query depends on the data structure and can be performed in many different ways. The complexity of a query program is related to the number of collections involved and to the embedding level where the required data is located.

The performance and the readability of the query programs relies on the application developer's skills and knowledge.

III. Data structuring alternatives

This section explores the data structuring alternatives that are possible in MongoDB, in order to analyse their advantages and possible drawbacks according to the application needs. We used a simple example called *the Company*, which was already mentioned. Figure 1 shows the entity-relationship model that involves Company, Department and Employee. Relationships are 1..* with the usual semantics. A company is organized in one or several departments. A department is part of exactly one company and has several employees. Each employee works for a single department. This example does not consider entities with several relationships nor many to many relationships.

We created a set of Mongo databases for the `Company` case. As there is no database schema definition in Mongo, a database is a set of data collections that contains documents according to different data structuring. As embedding documents and sets of documents are particular features, we worked with six document schemes (named S1 up to S6), and mixed those choices thus leading to different access levels. Figure 2 illustrates the six schemes. A circle represents a collection whereas circles with several lines correspond to several sets of documents embedded into several documents. Next, we introduce the data structuring alternatives.

Separate collections & no document embedding: Document schemes S1 and S4 create a separate collection for each entity type. Each document has a field expected to be an identifier. Such identifiers are used for the references among documents to represent the relationships [4]. In S1, the references are in the collections of the entity participating with cardinality 1 to the relationships. In S4, the references are in a separated collection `CDES`. No documents are embedded.

Full embedded single collection: in S3 and S5, all relationships are materialized with embedded documents. In the `Company` case, the traversal of the relationships leads to one single collection embedding documents up to level 3. In S3, `company` documents are at the first level of the collection. Each document embeds n `department` documents (level 2) which in turn embed n `employee` documents (level 3). The embedding choice in S3 reflects the *1 to many* direction of the relationship. In S5, the rational is similar, but it uses the *many to 1* direction. `Employee` documents are at level 1 of the collection whereas `company` documents are embedded at level 3. This choice introduces redundancy of the embedded documents in a collection (e.g. company and department).

Embedding & referencing: S2 and S6 combine collections with embedded documents and with references.

[4]This is similar to normalized relational schemes with primary and foreign keys

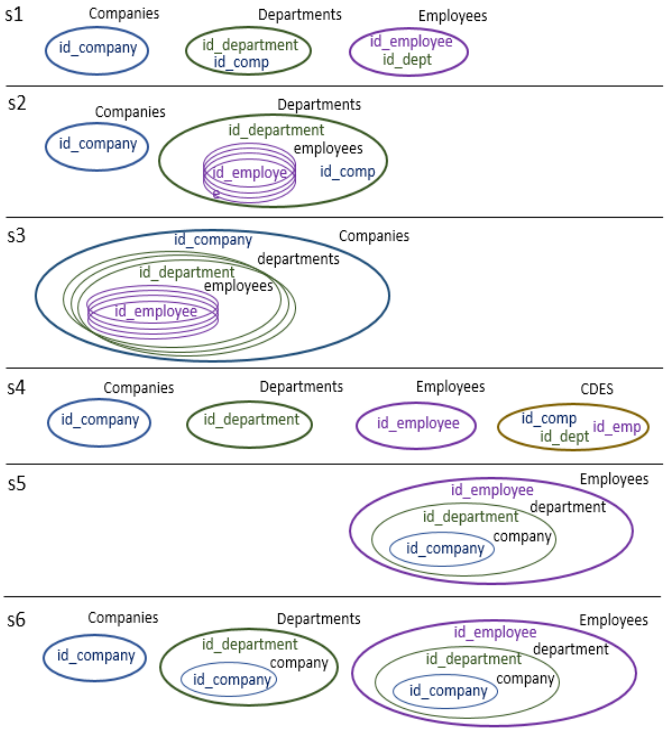
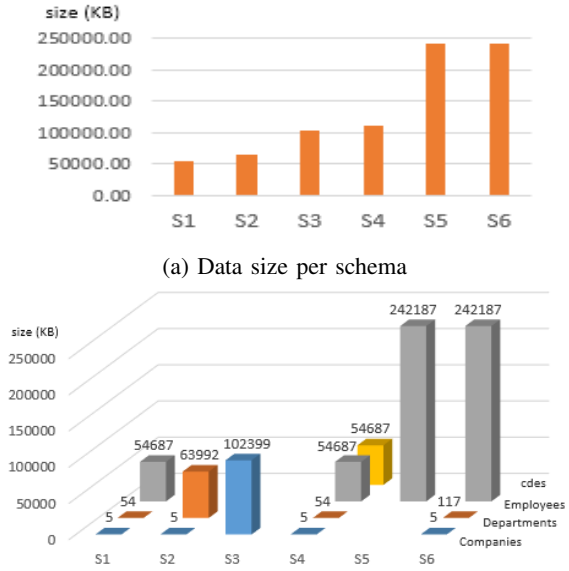


Fig. 2: Representative set of document schemes



(a) Data size per schema

(b) Data size per collection

Fig. 3: Data size

In S2, `Department` is a collection and its relationship with `employee` is treated by embedding the `employee` documents (i.e. array of documents) into the corresponding `department` document. This represents the *1 to many* direction of the relationship. The relationship between `Department` and `Company` is treated by referencing the `company` in each `department` through the `company id` (i.e. atomic field as the cardinality is 1). `Companies` is an independent collection.

Embedding & replicating: S6 has been created with the approach presented in [6]. It uses the *many to 1* direction of the relationships to embed the documents (as in S5) but it also replicates all the documents so as to have a "first level" collection for each entity type. Figure 2 shows the three collections of S6, where for instance, `Company` exists as 1) a separate collection, 2) as embedded documents in the `Departments` collection and 3) embedded at the third level in the `Employees` collection. There is also redundancy for the embedded documents, as was previously mentioned for S5.

IV. Experiment

This section is devoted to the experiment conducted with the data structuring alternatives introduced in Section III. Section IV-A presents the experiment setup. In Section IV-B, we discuss memory requirements regarding the data structuring choices. Section IV-C presents the queries implemented for different document schemes.

A. Experiment set up

For our study, we created six Mongo databases using the schemes S1 to S6. Each of them has been populated with the same data: 10 companies, 50 departments per company and 1000 employees per department. Data is consistent.

We implemented the queries introduced in Section IV-C for all the databases. In some cases, the implementation of a query differs a lot from one schema to another. For each db, each query has been executed 31 times with indexes, and 31 times without indexes. The first execution of each sequence was separated in order to avoid load in memory effects. The experiment was run on a workstation (Intel Core i7 Processor with 1.8 GHz and 8GB of RAM).

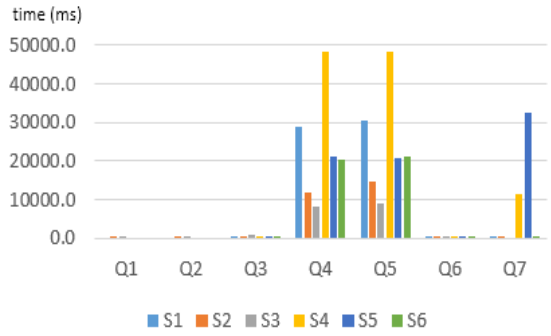
B. Data size evaluation

Figures 3a shows the size of each database and figure 3b shows the size of each collection. Note the large size of S5 and S6 databases with respect to the other options. The size of S5 and S6 are mainly dominated by collection `Employee` which has a fully embedded structure following the many-to-one relationships. `Company` and `department` documents are replicated in `Employees`. S6 is a little larger than S5, because of collections `Companies` and `Departments` which are extra copies in S6.

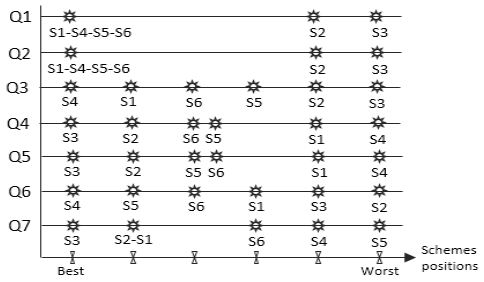
S4 and S1 do not embed documents. The size of S4 is twice the size of S1 because of the size of the `cdes` collection (representing the relationships) which is meaningful in our set-up.

The size of embedded collections tends to grow even if they don't have any replication. See S2 and S3 wrt S1.

$size(\texttt{Departments@S2}) > (size(\texttt{Employees@S1}) + size(\texttt{Department@S1}))$,

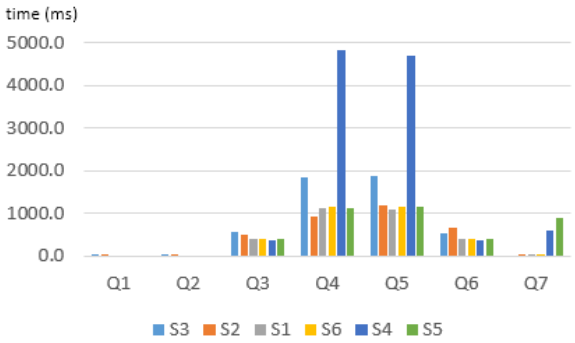


(a) Median execution time of queries by schema

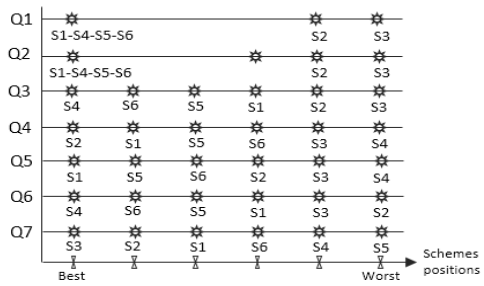


(b) Relative schemes positions

Fig. 4: Synthesis of executions



(a) Median execution time of queries using indexes



(b) Relative schemes positions using indexes

Fig. 5: Synthesis of executions using indexes

so as,
$size(\texttt{Companies@S3}) > (size(\texttt{Employees@S1}) + size(\texttt{Department@S1}) + size(\texttt{Companies@S1}))$

With respect to S1, both S2 and S3 have more complex structures with arrays of embedded documents at different levels.

C. Queries

To analyse the impact of the data structuring, we established a set of queries Q to be executed and evaluated using each of the schemes. Q includes queries with increasing complexity: a selection with a simple equality/inequality predicate on a single entity type (see Q1, Q2, Q6), queries requiring partitioning (see Q4), queries involving several entity types (see Q5, Q7) and aggregate functions (see Q3).

[Q1] Employees with a salary equal to $1003.
[Q2] Employees with a salary higher than $1003.
[Q3] Employees with the highest salary.
[Q4] Employees with the highest salary per company and the company id.
[Q5] Employees with the highest salary per company and the company name.
[Q6] The highest salary.
[Q7] Information about the companies including the name of their departments.

These queries have been used to evaluate the performance in each scheme.

V. Results analysis [5]

In this section, we analyse the results of the experiment from the point of view of data structure and query performance (Section V-A). In Section V-B, we focus on the impact of indexing. In Section V-C, we propose a discussion around the facts that were confirmed and found after performing this experiment.

A. Databases without indexes

As was previously mentioned, the experiment includes the execution of queries Q1 to Q7 on the 6 databases — S1 to S6 —. In order to ease the analysis, Figure 4 shows the median execution time for each query in the corresponding schema. In addition, Figure 4b depicts the relative performance for each query with respect to all schemes. There is a line per query. Schemes are ordered from left to right, starting with the one where performances are the best. For example, for Q7, the best performance was obtained in S3 and the worst in S5. For Q2, the performances in S1, S4, S5 and S6 are very close and clearly better than the performance in S3.

Let's first focus on schemes with *full embedded single collections*. For such schemes, performance is very good when the query access pattern is in the same direction of the embedded structure. This can be appreciated in S3 and S5 for queries Q4, Q5 and Q7. When Q4 and Q5 search for data of `employees` by `company`, S5 is not good because the data is dispersed at levels 1 and 3, respectively. This implies

[5]Data of full results are provided in:
https://undertracks.imag.fr/php/studies/study.php/data_schemes_in_nosql_systems

traversing down and up several times into the nested structure and discarding the unnecessary information involved in the intermediate level.

We will use the term *intrajoins* to refer to the process of traversing down and up nested structures in order to find and relate data.

The case is different in S3. Here, `employees` are at level 3 and `companies` at level 1, thus matching with the access pattern of Q7. This implies dealing with an intermediate level as in S5; however, in S3, it is not necessary to go up and down between levels. Therefore, S3 performs the best.

A similar behaviour occurs in Q7, which requires `company` data and the names of its `departments`. S5 has the required data at levels 3 and 2 respectively —in an inverse order— and they are not grouped. This implies crossing through each document at level 1, discarding useless information, and moving down to levels 2 and 3. All this to group the data located at level 2 based on the data at level 3. The worst performance of Q7 appears in S5.

Regarding S3, the access pattern of Q7 matches its embedding structure perfectly. Here the `companies`' data are at level 1 and the `departments`' data are grouped at level 2. S3 performs best again. However, this is not true for all the queries. Q3 on S3 has the worst performance because the required data are nested at level 3.

In order to favor certain access, several copies of the same data can be created using collections with different structures, as is done in S6. S6 extends S5 by including additional collections such as `Departments`, where `company` is embedded. This collection corresponds exactly to the data required by Q7. Q7 does not use `employees`. In this case, even if *intrajoins* are necessary, and the query access pattern is in the other direction, Q7 performs much better in S6 than in S5. The main reason is that the required data are at levels 1 and 2 in S6, thus avoiding any intermediate or superior levels. In S5, the employees have to be traversed even if their data is not useful for Q7.

When considering schemes with *separate collections without document embedding* such as S1 and S4, it is evident that they perform the best for queries Q1, Q2 and Q3. This is explained by the fact that the read-set of these queries perfectly matches a collection. No useless data are read, nor complex structures need to be manipulated. This is not the case for Q4 and Q5, which perform badly in S1 and S4. In both cases, the queries require data concerning employees and companies —or their relationship—, which are stored in separate collections —i.e., `Cdes@S4` and `Departments@S1`—. Therefore in S1 and S4, the query evaluation requires join and grouping operations, whereas in S3, the `Company` collection already reflects that.

Q7 is also very inefficient in S4. The overhead is due to the search of the departments-company relationship in the `Cdes` collection.

It should be stressed that the hypothesis on *data consistency* is important for query implementation. For example, for Q7 in S5 and S6, this hypothesis allows the extraction of the name of the first `department` we found without scanning the whole extension of department.

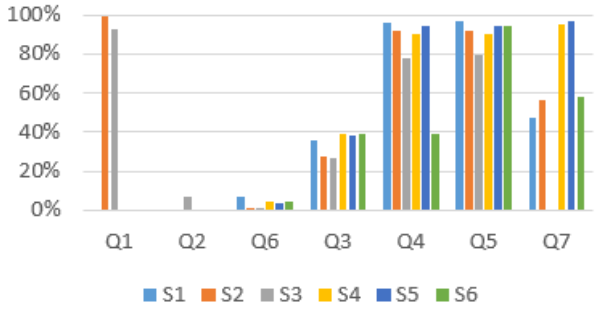


Fig. 6: Impact index

B. Indexes impact

We complete our study by creating indexes in all databases and analyzing the benefit on the performances of Q1 up to Q7. We created indexes on the identifiers (e.g.. company id) and on the salary. Figures 5a and 5b provide a synthesis of the results. Figure 6 shows the speedup obtained using the index with respect to the setup without index.

As expected, indexes improve performances. In particular, the improvement for Q4 and Q5 is high in all schemes, and reaches 96% in S1 (see Figure 6). Some schemes, such as S1 and S5, which perform the worst without indexes, perform very well with indexes (see Figure 5b). Interestingly, the benefit of the indexes in S3, is less important than for the other schemes. S3 is relegated to the last performance position when compared to the other schemes.

Furthermore, indexes improve the performances of Q7 in all schemes. Nevertheless, the improvement obtained in S4 and S5 is not enough to overtake S3. This shows that even if indexes are efficient, there are cases where they cannot compete against an unsuitable schema.

C. Discussion

As was previously discussed, the experiment allowed us to confirm some of the intuitive ideas on data modeling with MongoDB, but it also pointed out unexpected aspects. These aspects concern the performances but also the readability of the code used for implementing the queries.

The embedding level of the data has an impact on performances. Accessing data at the first level of a collection is faster and easier than accessing data in deeper levels.

Querying data stored at different embedding levels in a collection may require complex manipulations. For example, when the structure embeds arrays of documents, the algorithms to manage them are similar to intra-collection joins. They affect performances, but also require more elaborate programming.

The type of results, impacts performances. This point may be an issue when working with complex data. The structure of selected data is pre-formated with the structure of the queried collection. For example, when extracting a field A appearing in documents embedded at level k, the result will maintain the embedded structure if no restructuring operations are performed. This means that the answer may have useless embedding levels

and not required information issued from any of the k-1 levels that have been traversed to access field A. Changing such a structure to provide data in another format requires extra processing. The cost of this extra processing should not be neglected.

Concerning storage requirements, our experiment revealed that using collections with embedded documents tends to require more storage than using separate "flat" collections and references for the same data.

VI. Related work

Many works focus on automatically transforming a relational schema to a data model supported by a NoSQL system. [6] [7]. Some works compare performance, scalability and other properties between relational and NoSQL systems [8] [9] [10]. Others such as [11] [12] introduce schema management on top of schema-less document systems.

Zhao et al in [6] present a systematic schema conversion that follows a de-normalization approach where: 1) each table is a new collection, 2) each foreign key in a table is transformed into an embedded structure where the keys are replaced by the actual data values. They pre-calculate some natural joins with embedded schemes to improve the query performance with respect to a relational DBMS. Additionally, the paper shows that, even though query performance for some "joins" is better, space performance is worst due to data replication. In our experiment, schema S6 corresponds to the Zhao's strategy. According to our results, if the queries follow the embedded structure, performance is better than in other schemes. In the other cases performance is poorer.

Lombardo et al in [13] propose a framework to determine the key-value tables that are most suited to optimize the execution of the queries. From an entity-relationship diagram and the most used queries, they propose to create redundant tables to improve performance (they do not talk about the cost in storage). The goal of their framework is similar to ours: To help NoSQL developers to choose the most suitable data structuring according to the needs of the application. The approach is different because we intend to create several schemes and evaluate them before making a final decision.

Mior in [14] follows a similar approach to ours but focus on Cassadra. He proposes a static analysis of the storage impact comparing a set of queries in different schemes. We compare and analyse based on experimentation.

VII. Conclusion and future work

This paper reports a study on the impact of data structuring in document-based systems such as MongoDB. This system is scheme-less, as are several NoSQL systems. We worked with several data structuring schemes in order to evaluate size performance, and with several queries in order to analyse the execution time with respect to the schemes. The experiments demonstrated that data structure choices do matter. Collections with embedded documents have a positive impact on queries that follow the embedding order. However, there is no benefit —or there is the possibility of bad performance— for queries accessing the data in another embedding order or requiring to access data embedded at different levels in the same collection. The reason for the latter is that the required manipulations are of similar complexity than that of joins of several collections. Also, collections with embedded documents —even those without replication— tend to require more storage than the same data represented on separate collections.

Future work includes extending our study to cover more complex data scenarios. For instance, relationships with 0 and N..M cardinality and several relationships for an entity type. Considering the semi-structured data model, it would also be interesting to explore more modeling alternatives. For example, the use of fragmented documents and partial replication. Our long-term objective is to provide developers with a design assistant tool to help them solve trade-offs.

Acknowledgment

The authors would like to thank N. Mandran, C. Labbé and C. Vega for their helpful comments on this work.

References

[1] P. J. Sadalage and M. Fowler, *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2012.

[2] N. Leavitt, "Will nosql databases live up to their promise?" *Computer*, vol. 43, no. 2, pp. 12–14, Feb 2010.

[3] J. Han, E. Haihong, G. Le, and J. Du, "Survey on nosql database," in *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on*, Oct 2011, pp. 363–366.

[4] A. Kanade, A. Gopal, and S. Kanade, "A study of normalization and embedding in mongodb," in *Advance Computing Conference (IACC), 2014 IEEE International*, Feb 2014, pp. 416–421.

[5] S. Alsubaiee, A. Behm, R. Grover, R. Vernica, V. Borkar, M. J. Carey, and C. Li, "Asterix: scalable warehouse-style web data integration," in *Proceedings of the Ninth International Workshop on Information Integration on the Web*. ACM, 2012, p. 2.

[6] G. Zhao, Q. Lin, L. Li, and Z. Li, "Schema conversion model of sql database to nosql," in *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2014 Ninth International Conference on*, Nov 2014, pp. 355–362.

[7] W.-C. Chung, H.-P. Lin, S.-C. Chen, M.-F. Jiang, and Y.-C. Chung, "Jackhare: a framework for sql to nosql translation using mapreduce," *Automated Software Engineering*, vol. 21, no. 4, pp. 489–508, 2014.

[8] R. Cattell, "Scalable sql and nosql data stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2011.

[9] Y. Li and S. Manoharan, "A performance comparison of sql and nosql databases," in *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*. IEEE, 2013, pp. 15–19.

[10] A. Boicea, F. Radulescu, and L. I. Agapin, "Mongodb vs oracle–database comparison," in *2012 Third International Conference on Emerging Intelligent Data and Web Technologies*. IEEE, 2012, pp. 330–335.

[11] L. Wang, S. Zhang, J. Shi, L. Jiao, O. Hassanzadeh, J. Zou, and C. Wangz, "Schema management for document stores," *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 922–933, 2015.

[12] D. S. Ruiz, S. F. Morales, and J. G. Molina, "Inferring versioned schemas from nosql databases and its applications," in *Conceptual Modeling*. Springer, 2015, pp. 467–480.

[13] S. Lombardo, E. D. Nitto, and D. Ardagna, "Issues in handling complex data structures with nosql databases," in *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2012, Timisoara, Romania, September 26-29, 2012*, 2012, pp. 443–448. [Online]. Available: http://dx.doi.org/10.1109/SYNASC.2012.59

[14] M. J. Mior, "Automated schema design for nosql databases," in *Proceedings of the 2014 SIGMOD PhD symposium*. ACM, 2014, pp. 41–45.