Formalizing MongoDB Queries

Elena Botoeva, Diego Calvanese, Benjamin Cogrel, and Guohui Xiao

Free University of Bozen-Bolzano, Italy
lastname@inf.unibz.it

Abstract. In this paper, we report on our ongoing work in which we formalize MongoDB, a widely adopted document database system managing complex (tree structured) values represented in a JSON-based data model, equipped with a powerful query mechanism. We study the expressiveness of the MongoDB query language, showing its equivalence with nested relational algebra, and we investigate the computational complexity of significant fragments of it.

1 Introduction

In the past decade numerous database (DB) architectures and data models have been proposed as attempts to better address the varying demands of modern data-intensive applications. Many of these new systems do not rely on the relational model but instead adopt a semi-structured data format, and alternative query mechanisms, which combine an increased flexibility in data handling, with a higher efficiency (at least for the most common operations). These systems are generally categorized under the term *NoSQL* (which stands for "not only SQL") [6,13].

A popular design among these *non-relational* systems consists in organizing data in collections of semi-structured, tree-shaped documents in the JavaScript Object Notation (JSON) format. Such documents can be seen as complex values [9,1,18,8], in particular when they contain nested arrays. As an example, consider the document in Figure 1, containing personal information about Kristen Nygaard (e.g., name and birth-date) together with information about the awards he received, stored in an array.

Unsurprisingly, many similarities can be observed between non-relational languages for querying JSON collections having rich capabilities (see, e.g., [2,14,16]), and well-known query languages for complex values, such as nested relational algebra

```
{"_id": 4,
 "awards": [
   { "award": "Rosing Prize", "year": 1999, "by": "Norwegian Data Association" },
   { "award": "Turing Award", "year": 2001, "by": "ACM" },
   { "award": "IEEE John von Neumann Medal", "year": 2001, "by": "IEEE" } ],
 "birth": "1926-08-27",
 "contribs": ["OOP", "Simula"],
 "death": "2002-08-10",
 "name": { "first": "Kristen", "last": "Nygaard" }
}
```

Fig. 1. A sample MongoDB document

$$\begin{array}{rcl} \textsc{Value} & ::= & \textsc{Literal} \mid \textsc{Object} \mid \textsc{Array} \\ \textsc{Object} & ::= & \{\!\{ \textsc{List}\langle \textsc{Key} : \textsc{Value} \rangle \}\!\} \\ \textsc{Array} & ::= & [\ \textsc{List}\langle \textsc{Value} \rangle\ ] \end{array} \qquad \begin{array}{rcl} \textsc{List}\langle T \rangle & ::= & \varepsilon \mid \textsc{List}^+ \langle T \rangle \\ \textsc{List}^+ \langle T \rangle & ::= & T \mid T\ ,\ \textsc{List}^+ \langle T \rangle \end{array}$$

Fig. 2. Syntax of JSON objects. We use double curly brackets to distinguish objects from sets

(NRA) [15,17], monad algebra [5,12] and Core XQuery [12]. However, the formal semantics and the computational properties of these query languages are still largely not understood and are being actively investigated [4,10].

In this work, we conduct the first major study into the formal foundations and properties of the data model and query language of MongoDB, a widely adopted distributed JSON-based document database. MongoDB provides rich querying capabilities by means of the *aggregation framework*[1]. In this framework, a query is a multi-stage pipeline, where each stage defines a transformation, using a MongoDB-specific operator, applied to the set of documents produced by the previous stage.

Our first contribution is a formalization of the MongoDB data model and of the fragment of the aggregation framework query language that includes the ***m**atch*, ***u**nwind*, ***p**roject*, ***g**roup*, and ***l**ookup* operators, and which we call *MQuery*. Each of these operators roughly corresponds to an operator of NRA: match corresponds to select, project to project, lookup to left join, group to nest, and unwind to unnest.

Our second contribution is a characterization of the expressive power of MQuery obtained by comparing it with NRA. We devise translations in both directions between the two languages showing that they are equivalent in expressive power.

Finally, we carry out an investigation of the computational complexity of $\mathcal{M}^{\text{MUPGL}}$ and of various fragments of it. Interestingly, since our translations between MQuery and NRA are compact (i.e., polynomial), they allow us also to carry over complexity results between MQuery and NRA.

We provide here only an overview, and refer to the full version for more details [3].

2 MQuery

Objects in the JSON format are defined inductively as consisting of key-value pairs, where a *key* is a string, and a *value* can be a literal, an object, or an array of values, constructed inductively according to the grammar in Figure 2 (where terminals are written

[1] https://docs.mongodb.com/core/aggregation-pipeline/

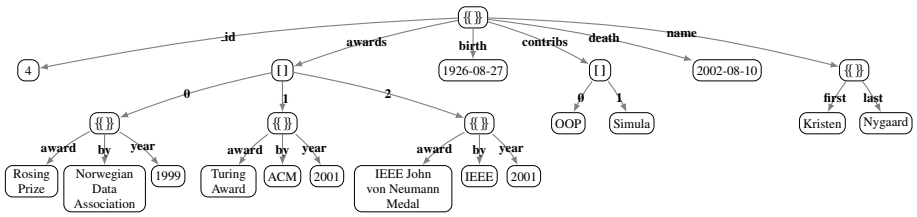

Fig. 3. The tree representation of the MongoDB document in Figure 1

$$\varphi ::= p = v \mid \exists p \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \neg\varphi \qquad P ::= p \mid p/d \mid p, P \mid p/d, P$$
$$d ::= v \mid p \mid [d, \ldots, d] \mid \beta \mid \kappa \qquad G, A ::= p/p' \mid p/p', G$$
$$\beta ::= p = p \mid p = v \mid v = v \mid \exists p \mid d \vee d \mid d \wedge d \mid \neg d \qquad s ::= \mu_\varphi \mid \omega_p^n \mid \rho_P^{ni} \mid \gamma_{G:A} \mid \lambda_p^{p_1=C.p_2}$$
$$\kappa ::= (d?d{:}d) \qquad \text{MQuery} ::= C \triangleright s \triangleright \cdots \triangleright s$$

Fig. 4. Algebra for MQuery. Here, p denotes a path, v a value, and C a collection name

in black, and non-terminals in blue). The set of key-value pairs constituting a JSON object may not contain the same key twice. A MongoDB database stores collections of documents, where each *collection* has a name, and consists of a finite set of documents. Each *document* is a JSON object (not nested within any other object) with a special key '`_id`', which is used to identify the document. Figure 1 shows a MongoDB document in which, apart from `_id`, the keys are `birth`, `name`, `awards`, etc. Intuitively, a collection corresponds to a table in a (nested) relational database, and a document to a row in a table. We formalize documents as finite *unordered*, *unranked*, *node-labeled*, and *edge-labeled trees*, and collections as forests. The tree corresponding to the document in Figure 1 is depicted in Figure 3.

MongoDB is equipped with a powerful query mechanism provided by the *aggregation framework*. As a first contribution, we formalize the core aspects of such query language. We call our language *MQuery*, and consider also different fragments of it. An *MQuery* is a sequence of stages s, also called a *pipeline*, applied to a collection name C, where each of the stages roughly corresponds to a relational algebra operation, and transforms a forest into another forest. Here we are not concerned with syntactic aspects of MQuery, and instead propose for it an algebra, shown in Figure 4.

In an MQuery, *paths*, which are (possibly empty) concatenations of keys, are used to access actual values in a tree, similarly to how attributes are used in relational algebra. We use ε to denote the empty path. MQuery allows for five types of stages:

- ***m**atch* μ_φ, selecting trees according to the criterion φ, which is a Boolean combination of atomic conditions that express either the equality of a path p to a value v, or the existence of a path p.
- ***u**nwind* ω_p and ω_p^+, which flatten an array reached through a path p in the input tree, and output a tree for each element of the array; the latter operator preserves a tree even when the array does not exist or is empty.
- ***p**roject* ρ_P and $\rho_P^{\pm\mathtt{id}}$, which modify trees by projecting away paths, renaming paths, or introducing new paths; the latter version projects away `_id`, which otherwise is kept by default. Here P is a sequence of elements of the form p or q/d, where p is a path to be kept, and q is a new path whose value is defined by d. Such a *value definition* d can provide for q a constant v, the value reached through a path p (i.e., *renaming* path p to q), a new array defined through its values, the value of a Boolean expression β, or a value computed through a conditional expression κ. Note that, in a Boolean value definition β, one can also compare the values of two paths, while in a criterion φ one can only compare the value of a path to a constant value. Also note that each value definition can be evaluated to a Boolean value.
- ***g**roup* $\gamma_{G:A}$, which groups trees according to a grouping condition G and collects values of interest according to an aggregation condition A. Both G and A are (pos-

sibly empty) sequences of elements of the form p/p', where p' is a path in the input documents, and p a path in the output document. In these sequences, if p coincides with p', then we simply write p instead of p/p. Each group in the output will have an `_id` whose value is given by the values of p' in G for that group.

– ***lookup*** $\lambda_p^{p_1=C.p_2}$, which joins input trees with trees in an external collection C, using a local path p_1 and a path p_2 in C to express the join condition, and uses a path p to store the matching trees in an array.

We consider also various fragments of MQuery, and we denote each fragment by $\mathcal{M}^{\alpha}$, where α consists of the initials of the stages that can be used in queries in the fragment. Hence, $\mathcal{M}^{\text{MUPGL}}$ denotes MQuery itself, e.g., $\mathcal{M}^{\text{MUPG}}$ the fragment of $\mathcal{M}^{\text{MUPGL}}$ that does not use lookup, and $\mathcal{M}^{\text{MUP}}$ the fragment that additionally does not use group.

3 Results

We study the expressiveness and the computational complexity of MQuery.

First, we show that for MongoDB instances of a certain regular structure, MQuery is essentially equivalent to nested relational algebra (NRA). Regularity of MongoDB instances allows for defining a nested relational view of MongoDB documents, which serves for establishing the correctness of the translations between MQuery and NRA, and hence the equivalence between $\mathcal{M}^{\text{MUPGL}}$ and NRA. We also consider the $\mathcal{M}^{\text{MUPG}}$ fragment, where we rule out the lookup operator, which allows for joining a given document collection with external ones, and establish that already $\mathcal{M}^{\text{MUPG}}$ is equivalent to NRA over a single relation, and hence is capable of expressing arbitrary joins (within one collection), contrary to what is believed in the community of MongoDB practitioners and users.

Second, we obtain the exact complexity of $\mathcal{M}^{\text{MUPGL}}$ and of some of its fragments:

– What we consider the minimal fragment, namely $\mathcal{M}^{\text{M}}$, which allows only for match, is LOGSPACE-complete in combined complexity.
– Projection and grouping allow one to create exponentially large objects, but by representing intermediate results compactly as DAGs, one can still evaluate $\mathcal{M}^{\text{MPGL}}$ queries in PTIME. Specifically, $\mathcal{M}^{\text{MP}}$ is PTIME-hard in query complexity and $\mathcal{M}^{\text{MPGL}}$ is in PTIME in combined complexity.
– For $\mathcal{M}^{\text{MU}}$, the use of unwind causes loss of tractability in combined complexity, specifically it leads to NP-completeness, but the language remains LOGSPACE-complete in query complexity.
– Further adding project in $\mathcal{M}^{\text{MUP}}$, or lookup in $\mathcal{M}^{\text{MUL}}$, leads again to NP-harness even in query complexity, although $\mathcal{M}^{\text{MUPL}}$ stays NP-complete in combined complexity.
– In the presence of unwind, grouping provides another source of complexity, since it allows one to create doubly-exponentially large objects; indeed $\mathcal{M}^{\text{MUG}}$ is PSPACE-hard in query complexity.
– The full language $\mathcal{M}^{\text{MUPGL}}$ and also the $\mathcal{M}^{\text{MUPG}}$ fragment are complete for $\text{TA}[2^{n^{O(1)}}, n^{O(1)}]$ (i.e., exponential time with a polynomial number of alternations [7,11]) in combined complexity, and in AC^0 in data complexity.

The latter result provides also a tight $\text{TA}[2^{n^{O(1)}}, n^{O(1)}]$ bound for the combined complexity of Boolean query evaluation in NRA, whose exact complexity was open [12].

Acknowledgements. This research has been partially supported by the project "Ontology-based Data Access for NoSQL Databases" (OBDAM), funded through the 2016 call issued by the Research Committee of the Free University of Bozen-Bolzano.

References

1. S. Abiteboul and C. Beeri. The power of languages for the manipulation of complex values. *Very Large Database J.*, 4(4):727–794, 1995.
2. K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *Proc. of the VLDB Endowment*, 4(12):1272–1283, 2011.
3. E. Botoeva, D. Calvanese, B. Cogrel, and G. Xiao. Expressivity and complexity of MongoDB (Extended Version). CoRR Technical Report arXiv:1603.09291, arXiv.org e-Print archive, 2017. Available at `http://arxiv.org/abs/1603.09291`.
4. P. Bourhis, J. L. Reutter, F. Suárez, and D. Vrgoč. JSON: Data model, query languages and schema specification. In *Proc. of the 36th Symp. on Principles of Database Systems (PODS)*, pages 123–135, 2017.
5. P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.
6. R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, May 2011.
7. A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.
8. E. Dantsin and A. Voronkov. Complexity of query answering in logic databases with complex values. In *Proc. of the 4th Int. Symp. on Logical Foundations of Computer Science (LFCS)*, pages 56–66, 1997.
9. S. Grumbach and V. Vianu. Tractable query languages for complex object databases. In *Proc. of the 10th Symp. on Principles of Database Systems (PODS)*, 1991.
10. J. Hidders, J. Paredaens, and J. Van den Bussche. J-Logic: Logical foundations for JSON querying. In *Proc. of the 36th Symp. on Principles of Database Systems (PODS)*, pages 137–149, 2017.
11. D. S. Johnson. A catalog of complexity classes. In *Handbook of Theoretical Computer Science*, volume A, chapter 2. Elsevier Science Publishers, 1990.
12. C. Koch. On the complexity of nonrecursive XQuery and functional query languages on complex values. *ACM Trans. on Database Systems*, 31(4):1215–1256, 2006.
13. N. Leavitt. Will NoSQL databases live up to their promise? *Computer*, 43(2):12–14, Feb. 2010.
14. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 1099–1110, 2008.
15. S. J. Thomas and P. C. Fischer. Nested relational structures. *Advances in Computing Research*, 3:269–307, 1986.
16. A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. of the VLDB Endowment*, 2(2):1626–1629, 2009.
17. J. Van den Bussche. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theoretical Computer Science*, 254(1):363–377, 2001.
18. J. Van den Bussche and J. Paredaens. The expressive power of complex values in object-based data models. *Information and Computation*, 120(2):220–236, 1995.