Formalization and Complexity of MongoDB Queries (Extended Abstract)*

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

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

Abstract. In this paper, we study MongoDB, a widely adopted but not formally understood database system managing JSON documents and equipped with a powerful query mechanism, called the aggregation framework. We define its formal abstraction MQuery, of which we study expressivity and computational complexity. We show the equivalence of MQuery and nested relational algebra, and obtain (tight) bounds in combined complexity, which range from LOGSPACE to alternating exponential-time with a polynomial number of alternations.

1 Introduction

JavaScript Object Notation (JSON) is currently adopted extensively as the de-facto standard format for representing nested data. JSON organizes data as semi-structured tree-shaped documents, with a minimalistic set of node types, and as such is commonly considered a lightweight alternative to XML. JSON documents can also be seen as complex values [7,1,15], in particular due to the presence of nested arrays. Consider, e.g., the document in Figure 1, containing personal information (such as name and birth-date) about Kristen Nygaard, and information about the awards he received, the latter stored inside an array.

Following its massive adoption by practitioners, recently JSON has also received attention in the database theory community. A powerful (Turing-complete, in its full generality) Datalog-like query language for JSON named JLogic is introduced in [8], where the expressive power and complexity of the full language and of significant fragments are studied. In [4], both JSON and its main schema language JSON Schema[1] are formalized, and their expressive power and the computational complexity of basic computational tasks, such as satisfiability and evaluation of expressions, are studied. Although some of the latter results apply to the simple *find* query language[2] of the widespread JSON-based document database system MongoDB, still little is known about the precise formal properties of the query languages for JSON with rich capabilities popular among practitioners, such as JSONiq [6] and SQL++ [12].

* SEBD 2018, June 24–27, 2018, Castellaneta Marina, Italy. Copyright held by the author(s). This is an abridged version of [3].

[1] http://json-schema.org/

[2] https://docs.mongodb.com/manual/crud/

```
{ "_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 JSON document in the `bios` collection.

Differently from XML, where XQuery is the official standard query language, embraced also by the developer community, so far there is no standard query language for JSON. However, in terms of adoption, the *MongoDB aggregation framework*[3] is currently the most prominent language providing rich querying capabilities over collections of JSON documents, and hence has become the de-facto standard language for JSON. This language is modeled on the flexible notion of a data processing pipeline, where a query consists of multiple stages, each defining a transformation using a specific operator, applied to the set of documents produced by the previous stage. As such, the language is very expressive and rich in features, but it has been developed in an ad-hoc manner, resulting in some counter-intuitive behavior.

We study the formal foundations and computational properties of the MongoDB aggregation framework, which has many similarities with well-known query languages for complex values, e.g., nested relational algebra (NRA) [14] and Core XQuery [11].

Our first contribution is a formalization of the JSON data model and of the aggregation framework query language, in which we deliberately abstract away some low-level features of MongoDB: we adopt set semantics (as opposed to bag or list semantics), and we abstract away from order within documents. Our formal language, which we call *MQuery*, includes the ***m**atch*, ***u**nwind*, ***p**roject*, ***g**roup*, and ***l**ookup* operators, roughly corresponding to the NRA operators select, unnest, project, nest, and left join, respectively. In our investigation, we consider various fragments of MQuery, which we denote by $\mathcal{M}^{\alpha}$, where α consists of the initials of the stages allowed in the fragment.

Our second contribution is a characterization of the expressive power of MQuery obtained by devising translations in both directions between (a suitably defined *well-typed* fragment of) MQuery and NRA, showing that the two languages are equivalent in expressive power. Our translations are compact (i.e., polynomial), hence complexity results between MQuery and NRA carry over.

Our third contribution is an investigation of the computational complexity of $\mathcal{M}^{\text{MUPGL}}$ and its fragments. We establish several tight bounds (in combined complexity), which range from LOGSPACE-complete for $\mathcal{M}^{\text{M}}$ to TA$[2^{n^{O(1)}}, n^{O(1)}]$-complete for MQuery itself. As a byproduct, we also establish a tight lower bound for the combined complexity of Boolean query evaluation in NRA.

In the following, we assume familiarity with nested relational algebra (NRA) [9,14].

[3] `https://docs.mongodb.com/manual/core/aggregation-pipeline/`

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

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

2 JSON Documents

In this section, we propose a formalization of the syntax and the semantics of JSON documents. With respect to MongoDB, we abstract away the order of key-value pairs within a document.

A MongoDB database stores collections of documents, where a collection corresponds to a table in a (nested) relational database, and a document to a row in a table. We define the syntax of documents. *Literals* are atomic values, such as strings, numbers, and Booleans. A *JSON object* is a finite set 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 the terminals are '$\{\!\{$', '$\}\!\}$', '[', ']', ':', and ','). We require that the set of key-value pairs constituting a JSON object does not contain the same key twice. A *(MongoDB) document* is a JSON object not nested within any other object, with a special key '`_id`', used to identify the document. Figure 1 shows a document with keys `_id`, `awards`, `birth`, etc. Given a collection name C, a *(MongoDB) collection for* C is a finite set F_C of documents, each identified by its value of `_id`, i.e., each value of `_id` is unique in F_C. Given a set $\mathbb{C}$ of collection names, a *MongoDB database instance* D *(over* $\mathbb{C}$*)* is a set of collections, one for each name $C \in \mathbb{C}$. We write $D.C$ to denote the collection for name C.

We formalize JSON objects as finite *unordered, unranked, node-labeled, and edge-labeled trees* (see Figure 3 for the tree t_{KN} corresponding to the document in Figure 1, where we have additionally labeled nodes with n_i, to refer to them later). We assume three disjoint sets of labels: the sets K of *keys* and I of *indexes* (non-negative integers), used as edge-labels, and the set V of *literals*, containing the special elements **null**, **true**, and **false**, and used as node labels. A *tree* is a tuple $(N, E, L_{\mathsf{n}}, L_{\mathsf{e}})$, where N is a set of nodes, E is the edge relation, $L_{\mathsf{n}} : N \to V \cup \{\text{'}\{\!\{\}\!\}\text{'}, \text{'}[\,]\text{'}\}$ is a node labeling function, and $L_{\mathsf{e}} : E \to K \cup I$ is an edge labeling function, such that *(i)* (N, E) forms a tree, *(ii)* a node labeled by a literal must be a leaf, *(iii)* all outgoing edges of a node labeled by '$\{\!\{\}\!\}$' must be labeled by keys, and *(iv)* all outgoing edges of a node labeled by '[ ]'

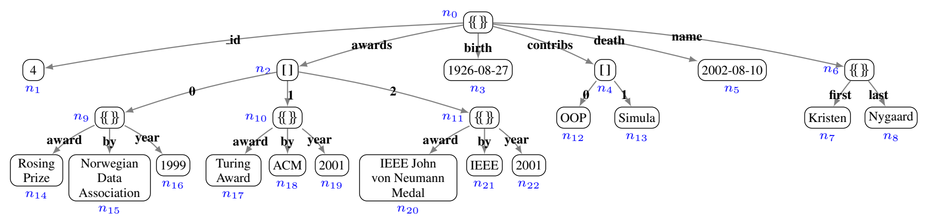

Fig. 3. The tree t_{KN} corresponding to the JSON document in Figure 1.

must be labeled by distinct indexes. The *type* of a node x in a tree t, denoted $\mathsf{type}(x,t)$, is defined as $\mathsf{literal}$ if $L_{\mathsf{n}}(x) \in V$, object if $L_{\mathsf{n}}(x) = $ '$\{\!\{\}\!\}$', and array if $L_{\mathsf{n}}(x) = $ '[ ]'. $\mathsf{root}(t)$ denotes the root of t. A *forest* is a set of trees.

We define inductively the *value represented by* a node x in a tree t, denoted $\mathsf{value}(x,t)$: *(i)* $\mathsf{value}(x,t) = L_{\mathsf{n}}(x)$, if x is a leaf in t; *(ii)* let $x_1,\dots,x_m$, be all children of x with $L_{\mathsf{e}}(x,x_i) = k_i$. Then $\mathsf{value}(x,t)$ is $\{\!\{k_1{:}\mathsf{value}(x_1,t),\dots,k_m{:}\mathsf{value}(x_m,t)\}\!\}$ if $\mathsf{type}(x,t) = \mathsf{object}$, and $[\mathsf{value}(x_1,t),\dots,\mathsf{value}(x_m,t)]$, if $\mathsf{type}(x,t) = \mathsf{array}$. The *JSON value represented by* t is then $\mathsf{value}(\mathsf{root}(t),t)$. Conversely, the *tree corresponding to a value* u, denoted $\mathsf{tree}(u)$, is defined as $(N, E, L_{\mathsf{n}}, L_{\mathsf{e}})$, where N is the set of all x_v such that v is an object, array, or literal value appearing in u, and for $x_v \in N$: *(i)* if v is a literal, then $L_{\mathsf{n}}(x_v) = v$ and x_v is a leaf; *(ii)* if $v = \{\!\{k_1{:}v_1,\dots,k_m{:}v_m\}\!\}$ for $m \geq 0$, then $L_{\mathsf{n}}(x_v) = $ '$\{\!\{\}\!\}$', and x_v has m children $x_{v_1},\dots,x_{v_m}$ with $L_{\mathsf{e}}(x_v,x_{v_i}) = k_i$; *(iii)* if $v = [v_1,\dots,v_m]$ for $m \geq 0$, then $L_{\mathsf{n}}(x_v) = $ '[ ]', and x_v has m children $x_{v_1},\dots,x_{v_m}$ with $L_{\mathsf{e}}(x_v,x_{v_i}) = i-1$. In the following, when convenient, we blur the distinction between JSON values and the corresponding trees.

3 The MQuery Language

MongoDB is equipped with an expressive query mechanism provided by the *aggregation framework* (see [2] for its formal syntax. Our first contribution is a formalization of the core aspects of this query language, where we use set (as opposed to bag and list) semantics, and we deliberately abstract away some low-level features that either are not relevant for understanding the expressive power and computational properties of the language, or appear ad-hoc and possibly are remnants of experimental development. We call the resulting language *MQuery*.

An *MQuery* is a sequence of stages, also called a *pipeline*, applied to a collection name C, where each stage transforms a forest into another forest. The grammar of MQuery is given 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. For two paths p and p', we say that p' is a *(strict) prefix* of p, if $p = p'.p''$, for some (non-empty) path p''. MQuery allows for five types of *stages*:

- ***match*** μ_φ, which selects trees according to criterion φ. Such criterion is a Boolean combination of atomic conditions $p = v$, expressing the equality of a path p to a value v, or $\exists p$, expressing the existence of a path p. E.g., for $\varphi_1 = (\texttt{\_id}{=}4)$, $\varphi_2 = (\text{awards.award="Turing Award"})$, and $\varphi_3 = (\text{name} = \{\!\{\text{first: "Kristen"}\}\!\})$, μ_{φ_1} and μ_{φ_2} select t_{KN}, but μ_{φ_3} does not.

$\varphi ::= \mathbf{true} \mid p = v \mid \exists p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi$	$P ::= p \mid p/d \mid p, P \mid p/d, P$
$d ::= v \mid p \mid [d,\dots,d] \mid \beta \mid (\beta?\, d{:}\, d)$	$G ::= p/p \mid p/p, G$
$\beta ::= \mathbf{true} \mid p = p \mid p = v \mid \exists p \mid \neg\beta \mid \beta \vee \beta \mid \beta \wedge \beta$	$A ::= p/p \mid p/p, A$
$s ::= \mu_\varphi \mid \omega_p \mid \rho_P \mid \gamma_{G:A} \mid \lambda_p^{p=C.p}$	$\text{MQuery} ::= C \triangleright s \triangleright \cdots \triangleright s$

Fig. 4. The MQuery language. Here, p denotes a path, v a value, C a collection name, φ a criterion, d a value definition, β a Boolean value definition, s a stage, P a list for project, G a list for grouping, and A a list for aggregation.

– ***unwind*** ω_p, which flattens an array reached through a path p in the input tree, and outputs a tree for each element of the array. E.g., ω_{awards} applied to t_{KN} produces three trees, which coincide on all key-value pairs, except for the `awards` key, whose values are nested objects such as, e.g., $\{\!\{$award: "Turing Award", year: 2001, by: "ACM"$\}\!\}$.
– ***project*** ρ_P, which modifies trees by projecting away paths, renaming paths, or introducing new paths. Here P is a sequence of elements of the form p or q/d, where p is a path to be kept, q is a new path whose value is defined by d, and among all such paths p and q, there is no pair p, p' where p is a prefix of p'. A *value definition* d can provide for q: *(i)* a constant v, *(ii)* the value reached through a path p (i.e., *renaming* path p to q), *(iii)* a new array defined through its values, *(iv)* the value of a Boolean expression β, or *(v)* a value computed through a conditional expression $(\beta ?\, d_1 \colon d_2)$. E.g., $\rho_{\text{bool}/(\text{birth}=\text{death}),\ \text{cond}/((\exists\text{awards})?\ \text{contribs}:\ \texttt{\_id}),\ \text{newArray}/[0,1]}$ applied to t_{KN} produces $\{\!\{$bool: **false**, cond: ["OOP", "Simula"], newArray: [0,1]$\}\!\}$.
– ***group*** $\gamma_{G\,:\,A}$, which groups trees according to a grouping condition G and aggregates values of interest according to A. Both G and A are (possibly empty) sequences of elements of the form p/p', where p' is a path in the input trees, and p a path in the output trees. Each different combination $\boldsymbol{v}$ of values in the input trees for the p's in G determines a group. For each such group there is a tree in the output with an `_id` whose value is constructed from $\boldsymbol{v}$ and the ps in G. The remaining keys in each output tree have as value an array constructed using the aggregation expression A. Consider, e.g., as input $\{\!\{$a: 1, b: "x"$\}\!\}$, $\{\!\{$a: 1, b: "y"$\}\!\}$, and $\{\!\{$a: 2, b: "z"$\}\!\}$. Then $\gamma_{\text{c}/\text{a}\,:\,\text{bs}/\text{b}}$ produces the two groups $\{\!\{$`_id`: $\{\!\{$c: 1$\}\!\}$, bs: ["x","y"]$\}\!\}$ and $\{\!\{$`_id`: $\{\!\{$c: 2$\}\!\}$, bs: ["z"]$\}\!\}$.
– ***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 stores the matching trees in an array under a path p. E.g., let C consist of $\{\!\{$`_id`: 1, a: 3$\}\!\}$ and $\{\!\{$`_id`: 2, a: 4$\}\!\}$. Then $\lambda_{\text{docs}}^{\texttt{\_id}=C.\text{a}}$ evaluated over t_{KN} adds to it `docs`: [$\{\!\{$`_id`: 2, a: 4$\}\!\}$].

We consider various fragments $\mathcal{M}^{\alpha}$ of MQuery, where α consists of the initials of the allowed stages. E.g., $\mathcal{M}^{\text{MUPGL}}$ denotes MQuery itself, while $\mathcal{M}^{\text{MUPG}}$ disallows lookup.

For the formal semantics of MQuery, we first observe that a path p over a tree t is interpreted as the set of nodes reachable via p from the root of t, where the indexes of intermediate arrays encountered in the tree are skipped. Then, given a forest F and a stage s, we can define the forest $F \triangleright s$ (for a lookup stage, we also require an additional forest F' as parameter) obtained by applying s to F. We refer to [3] for details. The semantics of an MQuery is obtained by composing (via $\triangleright$) the answers of its stages.

Definition 1. *Let $\boldsymbol{q} = C \triangleright s_1 \triangleright \cdots \triangleright s_n$ be an MQuery. The* result of evaluating $\boldsymbol{q}$ over *a MongoDB instance D, denoted $ans_{\mathsf{mo}}(\boldsymbol{q}, D)$, is defined as F_n, where $F_0 = D.C$, and for $i \in \{1, \ldots, n\}$, $F_i = (F_{i-1} \triangleright s_i)$ if s_i is not a lookup stage, and $F_i = (F_{i-1} \triangleright s_i[D.C'])$ if s_i is a lookup stage referring to an external collection name C'.*

4 Expressivity of MQuery

In this section we characterize the expressivity of MQuery in terms of nested relational algebra (NRA), and we do so by developing translations between the two languages. To make it possible to compare MQuery and NRA, we need to define how MongoDB instances can be viewed as nested relations. In the case of a MongoDB instance with an irregular structure, there is no natural way to define such a relational view. This happens either when the type of a path in a tree is not defined, or when a path has different types in two trees in the instance. Therefore, in order to define a schema for the relational view, which is also independent of the actual MongoDB instances, we impose on them some form of regularity. This is done by introducing the notion of *type* τ of a tree, which is analogous to complex object types [11], and similar to JSON schema [13].

A forest F is of type τ if all its trees are of type τ, and it is *well-typed* if it is of some type τ. We can then associate to each type τ a relation schema $\mathsf{rschema}(\tau)$ in which, intuitively, attributes correspond to paths, and each nested relation corresponds to an array in τ. The names of sub-relations and of atomic attributes in $\mathsf{rschema}(\tau)$ are given by paths from the root in τ, and therefore are unique. And we can define the relational view $\mathsf{rel}(F)$, of a well-typed forest F.

To define the relational view of MongoDB instances, we introduce the notion of *(MongoDB) type constraints*, which are given by a set $\mathcal{S}$ of pairs (C, τ), one for each collection name C, where τ is a type. We say that a database D *satisfies* the constraints $\mathcal{S}$ if $D.C$ is of type τ, for each $(C, \tau) \in \mathcal{S}$. For a given $\mathcal{S}$, for each $(C, \tau) \in \mathcal{S}$, we refer to τ by τ_C. Moreover, we assume that in $\mathsf{rschema}(\tau_C)$, the relation name R_{τ_C} is actually C. Then, for a set $\mathcal{S}$ of type constraints and a MongoDB instance D satisfying $\mathcal{S}$, the *relational view* $\mathsf{rdb}_{\mathcal{S}}(D)$ of D with respect to $\mathcal{S}$ is the instance $\{\mathsf{rel}(D.C) \mid (C, \tau) \in \mathcal{S}\}$.

Finally, we define equivalence between MQueries and NRA queries. To this purpose, we also define equivalence between two kinds of answers: well-typed forests and nested relations. We say that a well-typed forest F is *equivalent* to a nested relation $\mathcal{R}$, denoted $F \simeq \mathcal{R}$, if $\mathsf{rel}(F) = \mathcal{R}$. An MQuery $\boldsymbol{q}$ is *equivalent to* an NRA query Q w.r.t. type constraints $\mathcal{S}$, denoted $\boldsymbol{q} \equiv_{\mathcal{S}} Q$, if $ans_{\mathsf{mo}}(\boldsymbol{q}, D) \simeq ans_{\mathsf{ra}}(Q, \mathsf{rdb}_{\mathcal{S}}(D))$, for each MongoDB instance D satisfying $\mathcal{S}$ (where $ans_{\mathsf{ra}}(Q, \mathcal{R})$ denotes the answer to the NRA query Q computed over the nested relation $\mathcal{R}$).

We are now ready to establish the correspondence between NRA and MQuery. On the one hand, we show that $\mathcal{M}^{\text{MUPGL}}$ captures NRA, while $\mathcal{M}^{\text{MUPG}}$ captures NRA over a single collection. In our translation from NRA to MQuery, we have to deal with the fact that an NRA query in general has a *tree* structure where the leaves are relation names, while an MQuery contains *one sequence* of stages. So, we have to show how to "linearize" tree-shaped NRA expressions into a MongoDB pipeline. More precisely, we can show that it is possible to combine two $\mathcal{M}^{\text{MUPG}}$ sequences $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ of stages into a single $\mathcal{M}^{\text{MUPG}}$ sequence $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$, so that the results of $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ can be accessed from the result of $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$ for further processing. Having defined $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$, we are ready to show how to translate NRA to MQuery. For a singleton set $\mathcal{S} = \{(C, \tau_C)\}$ of type constraints for a collection name C and an NRA query Q over the relation name C (with schema $\mathsf{rschema}(\tau_C)$), we can define inductively on the structure of Q a pipeline $\mathsf{nra2mq}(Q)$, and then translate Q into the $\mathcal{M}^{\text{MUPG}}$ query $C \triangleright \mathsf{nra2mq}(Q)$. We obtain that $C \triangleright \mathsf{nra2mq}(Q) \equiv_{\mathcal{S}} Q$. This result can be generalized to NRA queries over multiple collections, by making use of *lookup*. We obtain:

Theorem 1. *$\mathcal{M}^{\text{MUPG}}$ captures NRA over a single collection, while $\mathcal{M}^{\text{MUPGL}}$ captures full NRA. Moreover there are polynomial translations from NRA to $\mathcal{M}^{\text{MUPG}}$/$\mathcal{M}^{\text{MUPGL}}$.*

To define a translation from MQuery to NRA we want to exploit the structure, i.e., the stages of MQueries. Hence, we define a translation $\mathsf{mq2nra}(s)$ from stages s to NRA expressions such that, for an MQuery $C \triangleright s_1 \triangleright \cdots \triangleright s_n$, the corresponding NRA query is defined as $C \circ \mathsf{mq2nra}(s_1) \circ \cdots \circ \mathsf{mq2nra}(s_n)$[4], where we identify the collection name C with the corresponding relation schema in the relational view. However, such a translation might not always be possible, since MQuery is capable of producing non well-typed forests, for which the relational view is not defined. Therefore, we restrict our attention to MQueries with stages preserving well-typedness. It is possible to check whether an MQuery satisfies this property.

The translation $\mathsf{mq2nra}(s)$, for well-typed stages s, is quite natural, although it requires some attention to properly capture the semantics of MQuery. It is given in [2].

Theorem 2. *Let $\mathcal{S}$ be a set of type constraints, $\boldsymbol{q}$ an MQuery $C \triangleright s_1 \triangleright \cdots \triangleright s_m$ in which each stage is well-typed for its input type, and $Q = C \circ \mathsf{mq2nra}(s_1) \circ \cdots \circ \mathsf{mq2nra}(s_m)$. Then $\boldsymbol{q} \equiv_{\mathcal{S}} Q$. Moreover, the size of Q is polynomial in the size of $\boldsymbol{q}$ and $\mathcal{S}$.*

5 Complexity of MQuery

We have studied the complexity of $\mathcal{M}^{\text{MUPGL}}$ and of some of its fragments, and have obtained the following results:

- 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 TA$[2^{n^{O(1)}}, n^{O(1)}]$ (i.e., exponential time with a polynomial number of alternations [5,10]) in combined complexity, and in AC^0 in data complexity.

The latter result provides also a tight TA$[2^{n^{O(1)}}, n^{O(1)}]$ bound for the combined complexity of Boolean query evaluation in NRA, whose exact complexity was open [11].

6 Conclusions and Future Work

We have carried out a first formal investigation on the foundations and computational properties of the MongoDB aggregation framework, currently the most widely adopted

[4] We follow the convention that $(f \circ g)(x) = g(f(x))$.

expressive query language for JSON. We proposed a clean abstraction for its five main operators, which we called MQuery. We have studied the expressivity of MQuery, establishing the equivalence between its well-typed fragment and NRA, by developing compact translations in both directions. This shows that, despite its design driven by practical requirements, the aggregation framework relies on solid foundations. Moreover, we analyzed the computational complexity of significant fragments of MQuery, obtaining several (tight) bounds. As a byproduct, we obtained also a tight bound for NRA.

We are currently working on applying our results to provide high-level access to MongoDB data sources by relying on the standard ontology-based data access (OBDA) paradigm [16]. We build on the translation from NRA to MQuery presented in Section 4.

Acknowledgements. This research has been partially supported by the projects OnProm and STyLoLa, respectively funded through the 2015 Call and the 2017 Interdisciplinary Call, both 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. *VLDBJ*, 4(4):727–794, 1995.
2. E. Botoeva, D. Calvanese, B. Cogrel, and G. Xiao. Expressivity and complexity of MongoDB (Extended version). CoRR Technical Report arXiv:1603.09291, 2017.
3. E. Botoeva, D. Calvanese, B. Cogrel, and G. Xiao. Expressivity and complexity of MongoDB queries. In *Proc. ICDT*, volume 98 of *LIPIcs*, pages 9:1–9:22, Dagstuhl, Germany, 2018.
4. P. Bourhis, J. L. Reutter, F. Suárez, and D. Vrgoč. JSON: Data model, query languages and schema specification. In *Proc. PODS*, pages 123–135, 2017.
5. A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *JACM*, 28(1):114–133, 1981.
6. D. Florescu and G. Fourny. JSONiq: The history of a query language. *IEEE Internet Computing*, 17(5):86–90, 2013.
7. S. Grumbach and V. Vianu. Tractable query languages for complex object databases. In *Proc. PODS*, pages 315–327, 1991.
8. J. Hidders, J. Paredaens, and J. Van den Bussche. J-Logic: Logical foundations for JSON querying. In *Proc. PODS*, pages 137–149, 2017.
9. G. Jaeschke and H.-J. Schek. Remarks on the algebra of non first normal form relations. In *Proc. PODS*, pages 124–138, 1982.
10. D. S. Johnson. A catalog of complexity classes. In *Handbook of Theoretical Computer Science*, volume A, chapter 2, pages 67–161. Elsevier, 1990.
11. C. Koch. On the complexity of nonrecursive XQuery and functional query languages on complex values. *ACM TODS*, 31(4):1215–1256, 2006.
12. K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ semi-structured data model and query language: A capabilities survey of SQL-on-Hadoop, NoSQL and NewSQL databases. CoRR Technical Report arXiv:1405.3631, 2017.
13. F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. Foundations of JSON schema. In *Proc. WWW*, pages 263–273, 2016.
14. 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. *TCS*, 254(1):363–377, 2001.
15. 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.
16. G. Xiao, D. Calvanese, R. Kontchakov, D. Lembo, A. Poggi, R. Rosati, and M. Zakharyaschev. Ontology-based data access: A survey. In *Proc. IJCAI*. AAAI Press, 2018.