Expressivity and Complexity of MongoDB Queries*

Elena Botoeva[1], Diego Calvanese[2], Benjamin Cogrel[3], and Guohui Xiao[†4]

1 **Faculty of Computer Science, Free University of Bozen-Bolzano, Italy**
`botoeva@inf.unibz.it`
2 **Faculty of Computer Science, Free University of Bozen-Bolzano, Italy**
`calvanese@inf.unibz.it`
3 **Faculty of Computer Science, Free University of Bozen-Bolzano, Italy**
`cogrel@inf.unibz.it`
4 **Faculty of Computer Science, Free University of Bozen-Bolzano, Italy**
`xiao@inf.unibz.it`

Abstract

In this paper, we consider 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 provide a clean formal abstraction of this query language, which we call MQuery. We study the expressivity of MQuery, showing the equivalence of its well-typed fragment with nested relational algebra. We further investigate the computational complexity of significant fragments of it, obtaining several (tight) bounds in combined complexity, which range from LogSpace to alternating exponential-time with a polynomial number of alternations.

1998 ACM Subject Classification H.2.1 Logical Design, H.2.3 Languages–Query languages

Keywords and phrases MongoDB – NoSQL – aggregation framework – expressivity

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.9

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 [9, 1, 19, 7], 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 [10], 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

* A full version of this paper with more details and selected proofs is available as a technical report [3].
† Corresponding author
1 `http://json-schema.org/`

```
{ "_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" } }
```

Figure 1 A sample JSON document in the `bios` collection.

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 [8] and SQL++ [14].

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.

Here, we propose a first study on the formal foundations and computational properties of the MongoDB aggregation framework. Since JSON documents can be seen as complex values and are closely related to XML documents, we expect the aggregation framework to have many similarities with well-known query languages for complex values, such as monad algebra [5, 13], nested relational algebra (NRA) [17, 18] and Core XQuery [13].

Our first contribution is a formalization of the JSON data model and of the aggregation framework query language. We aim at achieving a good balance between the contrasting requirements of capturing all aspects of MongoDB, and of keeping the formalization sufficiently simple and streamlined so as to allow for a formal study of the language properties. To do so, we deliberately abstract away some low-level features of MongoDB, which appear to be motivated by implementation aspects and possibly by ad-hoc choices, and we make some simplifying assumptions, commonly considered in database theory. Specifically, 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. As a useful side-effect of our formalization effort, we point out different "features" exhibited by MongoDB's query language that are somewhat counter-intuitive, and that might need to be reconsidered by the MongoDB developers.

Our second contribution is a characterization of the expressive power of MQuery obtained by comparing it with NRA. Given the regular structure of nested relations, the comparison requires considering JSON documents that are suitably *well-typed*, for which we define a relational view, and restricting the attention to *well-typed* MQuery, given that an arbitrary

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

$$e ::= a \mid c \mid f \mid (f?\, e{:}\, e) \mid \mathsf{subrel}(t, \ldots, t) \qquad t ::= \{b{:}e, \ldots, b{:}e\}$$
$$f ::= \mathbf{true} \mid a = a \mid a = c \mid \neg f \mid f \wedge f \mid f \vee f$$

Figure 2 Syntax of expressions e used in extended projection. Here, $a \in \mathsf{att}(R)$, c is a constant, f a Boolean expression, b a fresh attribute name, t a tuple definition, and $\mathsf{subrel}(t_1, \ldots, t_n)$ a relation definition, which constructs a relation from the tuples $t_1, \ldots, t_n$ of the same schema.

MQuery might produce non well-typed documents. We devise translations in both directions between well-typed MQuery and NRA, showing that the two languages are equivalent in expressive power. 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. Actually, we 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. Interestingly, all our translations are compact (i.e., polynomial), hence complexity results between MQuery and NRA carry over.

Finally, we carry out an investigation of the computational complexity of $\mathcal{M}^{\text{MUPGL}}$ and its fragments. In particular, we establish that what we consider the minimal fragment, 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. The use of unwind alone causes loss of tractability in combined complexity, specifically it leads to NP-completeness, but remains LogSpace-complete in query complexity. Adding also project or lookup leads again to intractability 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 we show PSpace-hardness of $\mathcal{M}^{\text{MUG}}$. Finally, we establish that the full language and also the $\mathcal{M}^{\text{MUPG}}$ fragment are TA$[2^{n^{O(1)}}, n^{O(1)}]$-complete (i.e., complete for exponential time with a polynomial number of alternations under LogSpace reductions) in combined complexity. As a byproduct of this study, we also establish that the TA$[2^{n^{O(1)}}, n^{O(1)}]$ lower bound previously known for the combined complexity of Boolean query evaluation in NRA is actually tight (the best known upper bound was ExpSpace [13]).

2 Preliminaries

We recap the basics of nested relational algebra (NRA) [11, 18], mainly to fix the notation.

Let $\mathcal{A}$ be a countably infinite set of attribute names and relation schema names. A *relation schema* has the form $R(S)$, where $R \in \mathcal{A}$ is a relation schema name and S is a finite set of attributes, each of which is an atomic attribute (i.e., an attribute name in $\mathcal{A}$) or a schema of a sub-relation. A relation schema can also be obtained through an NRA operation (see below). We use the function att to retrieve the attributes from a relation schema name, i.e., $\mathsf{att}(R) = S$. Let Δ be the domain of all atomic attributes in $\mathcal{A}$. An *instance* $\mathcal{R}$ of a relation schema $R(S)$ is a finite set of tuples over $R(S)$. A *tuple* t *over* $R(S)$ is a finite set $\{a_1{:}v_1, \ldots, a_n{:}v_n\}$ such that if a_i is an atomic attribute, then $v_i \in \Delta$, and if a_i is a relation schema, then v_i is an instance of a_i. We may refer to relation schemas by their name only.

A *filter* ψ over a set $A \subseteq \mathcal{A}$ is a Boolean formula constructed from atoms of the form $(a = v)$ or $(a = a')$, where $\{a, a'\} \subseteq A$, and v is an atomic value or a relation. Let R and R' be relation schemas. We use the following operators: *(1) set union* $R \cup R'$ and *set difference* $R \setminus R'$, for $\mathsf{att}(R) = \mathsf{att}(R')$; *(2) cross-product* $R \times R'$, resulting in a relation schema with

attributes $\{\text{rel1}.a \mid a \in \mathsf{att}(R)\} \cup \{\text{rel2}.a \mid a \in \mathsf{att}(R')\}$; *(3) selection* $\sigma_\psi(R)$, where ψ is a filter over $\mathsf{att}(R)$; *(4) projection* $\pi_P(R)$, for $P \subseteq \mathsf{att}(R)$; *(5) extended projection* $\pi_P(R)$, where P may also contain elements of the form b/e, for b a fresh attribute name, and e an expression constructed according to the grammar shown in Figure 2. Notice that such an expression is computable in AC^0 in data complexity; *(6) nest* $\nu_{\{a_1,\ldots,a_n\}\to b}(R)$, resulting in a schema with attributes $(\mathsf{att}(R) \setminus \{a_1,\ldots,a_n\}) \cup \{b(a_1,\ldots,a_n)\}$; and *(7) unnest* $\chi_a(R)$, resulting in a schema with attributes $(\mathsf{att}(R) \setminus \{a\}) \cup \mathsf{att}(a)$. For more details on (5)–(7) (including the semantics of extended projection), we refer to [3]. Given an NRA query Q and a (relational) database $\mathcal{D}$, the result of evaluating Q over $\mathcal{D}$ is denoted by $ans_{\mathsf{ra}}(Q, \mathcal{D})$.

3 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 3 (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 4 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 '[]' must be labeled by distinct indexes. The *type* of a node x in a tree t, denoted $\mathsf{type}(x, t)$, is defined as literal if $L_{\mathsf{n}}(x) \in V$, object if $L_{\mathsf{n}}(x) = \text{'}\{\{\}\}\text{'}$, and array if $L_{\mathsf{n}}(x) = \text{'}[]\text{'}$. $\mathsf{root}(t)$ 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 $x_1, \ldots, x_m$, be all children of x with $L_{\mathsf{e}}(x, x_i) = k_i$. Then $\mathsf{value}(x, t)$ is $\{\!\{k_1{:}\mathsf{value}(x_1, t), \ldots, k_m{:}\mathsf{value}(x_m, t)\}\!\}$ if $\mathsf{type}(x, t) = \mathsf{object}$, and $[\mathsf{value}(x_1, t), \ldots, \mathsf{value}(x_m, t)]$, if $\mathsf{type}(x, t) = \mathsf{array}$. The *JSON value represented by* t is

VALUE ::= LITERAL | OBJECT | ARRAY
OBJECT ::= {{ LIST<KEY : VALUE> }}
ARRAY ::= [LIST<VALUE>]

LIST<T> ::= ε | LIST$^+$<T>
LIST$^+$<T> ::= T | T , LIST$^+$<T>

Figure 3 Syntax of JSON objects. We use double curly brackets to distinguish objects from sets.

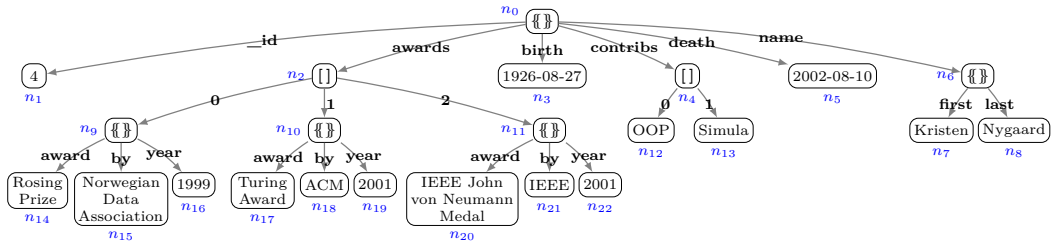

Figure 4 The tree t_{KN} corresponding to the JSON document in Figure 1.

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

Figure 5 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.

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 $v = \{\!\{k_1{:}v_1, \ldots, k_m{:}v_m\}\!\}$ for $m \geq 0$, then $L_{\mathsf{n}}(x_v) =$ '{{}}', and x_v has m children $x_{v_1}, \ldots, x_{v_m}$ with $L_{\mathsf{e}}(x_v, x_{v_i}) = k_i$; *(iii)* if $v = [v_1, \ldots, v_m]$ for $m \geq 0$, then $L_{\mathsf{n}}(x_v) =$ '[]', and x_v has m children $x_{v_1}, \ldots, x_{v_m}$ 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.

4 The MQuery Language

MongoDB is equipped with an expressive query mechanism provided by the *aggregation framework* (we refer to [3] for its formal syntax, but we provide in App. A some examples to illustrate its main features). 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 5. 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*[4]:

- ***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 $\varphi_1 = (\texttt{\_id}{=}4)$,

[4] We suggest readers unfamiliar with MongoDB to read the following paragraphs in parallel to the respective subsections in App. A, which contain additional examples and the actual syntax of MongoDB.

φ_2 = (awards.award="Turing Award"), and φ_3 = (name = {{first: "Kristen"}}), μ_{φ_1} and μ_{φ_2} select t_{KN}, but μ_{φ_3} does not. (See App. A.1 for details.)

- ***u**nwind* 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"}}. (See App. A.2.)
- ***p**roject* 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 $(\beta ?\, d_1 : d_2)$. E.g., $\rho_{\text{bool}/(\text{birth=death}),\, \text{cond}/((\exists\text{awards})?\, \text{contribs: \_id}),\, \text{newArray}/[0,1]}$ applied to t_{KN} produces {{bool: **false**, cond: ["OOP", "Simula"], newArray: [0,1]}}. (See App. A.3.)
- ***g**roup* $\gamma_{G:A}$, 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 $\vec{v}$ 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 $\vec{v}$ 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"] }}. (See App. A.4.)
- ***l**ookup* $\lambda_p^{p_1=C.p_2}$, 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 }}]. (See App. A.5.)

Observe that the Boolean expressions β allowed in a project stage are more expressive than those in the criterion φ of a match stage, since in the former one can also compare the values of two paths, while in the latter one can only compare the value of a path to a constant value. We consider also 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.

To define the semantics of MQuery, we introduce some auxiliary notions.

First, we show how to interpret paths over trees. Specifically, a path p is interpreted as the set of nodes reachable via p from the root, where the indexes of intermediate arrays that might be encountered in the tree are skipped. Given a tree $t = (N, E, L_{\mathsf{n}}, L_{\mathsf{e}})$, we interpret a (possibly empty) path p, and its concatenation $p.i_1 \dots i_m$ with indexes $i_1, \dots, i_m$, respectively as the sets of nodes $[\![p]\!]^t$ and $[\![p.i_1 \dots i_m]\!]^t$, according to the following inductive definition (below, q is a path, $j_1, \dots, j_n$ are indexes, and k is a key): $[\![\varepsilon]\!]^t = \{\mathsf{root}(t)\}$, and

$$[\![q.j_1 \dots j_n]\!]^t = \{y \in N \mid \text{there is } x \in [\![q.j_1 \dots j_{n-1}]\!]^t \text{ s.t. } (x,y) \in E \text{ and } L_{\mathsf{e}}(x,y) = j_n\}$$

$$[\![q.k]\!]^t = \{y \in N \mid \text{there are } j_1, \dots, j_n,\ n \geq 0, \text{ and } x \in [\![q.j_1 \dots j_n]\!]^t \text{ s.t. } (x,y) \in E \text{ and } L_{\mathsf{e}}(x,y) = k\}$$

For example, referring to the tree t_{KN} in Figure 4, $[\![\varepsilon]\!]^{t_{\text{KN}}} = \{n_0\}$, $[\![\texttt{\_id}]\!]^{t_{\text{KN}}} = \{n_1\}$, $[\![\text{awards}]\!]^{t_{\text{KN}}} = \{n_2\}$, $[\![\text{awards.1}]\!]^{t_{\text{KN}}} = \{n_{10}\}$, and $[\![\text{awards.award}]\!]^{t_{\text{KN}}} = \{n_{14}, n_{17}, n_{20}\}$. When $[\![p]\!]^t = \emptyset$, we say that the path p is *missing* in t.

Given a tree t and a path p, when $\mathsf{type}(x,t) = \mathsf{ty}$, for each $x \in [\![p]\!]^t$, where $\mathsf{ty} \in \{\mathsf{array}, \mathsf{literal}, \mathsf{object}\}$, we define the *type of* p *in* t, denoted $\mathsf{type}(p,t)$, to be ty. Also, when

Match	$F \triangleright \mu_\varphi = \{t \mid t \in F \text{ and } t \models \varphi\}$
Unwind	$\omega_p(t) = \{\mathsf{replace}(t, \mathsf{subtree}(t,p), \mathsf{subtree}(t,p.i))\}_{[\![p.i]\!]^t \neq \emptyset,\, i \in I}$ if p is a first array, and $\omega_p(t) = \emptyset$ otherwise $F \triangleright \omega_p = \bigcup_{t \in F} \omega_p(t)$
Project	$\rho_p(t) = \mathsf{subtree}(t, N_p)$, where N_p are all the nodes in t on a path from $\mathsf{root}(t)$ to a leaf via some $x \in [\![p]\!]^t$ $\rho_{q/d}(t) = \mathsf{attach}(q, \mathsf{eval}(d,t))$, unless d is a path and $t \not\models \exists d$, in which case $\rho_{q/d}(t)$ returns the empty tree $F \triangleright \rho_P = \{\bigoplus_{p \in P} \rho_p(t) \;\oplus\; \bigoplus_{(q/d) \in P} \rho_{q/d}(t) \mid t \in F\}$
Group	$F \triangleright \gamma_{g_1/y_1,\ldots,g_n/y_n\,:\,a_1/b_1,\ldots,a_m/b_m} =$ $\Big\{\mathsf{attach}(\mathtt{\_id}, \mathbf{null}) \oplus \bigoplus_{i=1}^m \mathsf{attach}\big(a_i, \mathsf{array}(F \triangleright \mu_{\varphi \wedge \exists b_i}, b_i)\big) \mid \varphi = \bigwedge_{j=1}^n (\neg \exists y_j),\ (F \triangleright \mu_\varphi) \neq \emptyset \Big\} \cup$ $\Big\{\bigoplus_{j \in J} \mathsf{attach}(\mathtt{\_id}.g_j, t_j) \oplus \bigoplus_{i=1}^m \mathsf{attach}\big(a_i, \mathsf{array}(F \triangleright \mu_{\psi \wedge \exists b_i}, b_i)\big) \mid J \in 2^{\{1,\ldots,n\}} \setminus \emptyset,$ $t_j \in \mathsf{forest}(F, y_j) \text{ for } j \in J,\ \psi = \bigwedge_{j \in J}((y_j = t_j) \wedge \exists y_j) \wedge \bigwedge_{j \notin J}(\neg \exists y_j),\ (F \triangleright \mu_\psi) \neq \emptyset \Big\}$
Lookup	$\lambda_p^{p_1 = C.p_2}[F'](t) = t \oplus \mathsf{attach}(p, \mathsf{array}(F' \triangleright \mu_\varphi, \varepsilon))$, for $\varphi = (p_2 = \mathsf{subtree}(t, p_1))$ if $t \models \exists p_1$, and $\varphi = \neg \exists p_2$ otherwise $F \triangleright \lambda_p^{p_1 = C.p_2}[F'] = \{\lambda_p^{p_1 = C.p_2}[F'](t) \mid t \in F\}$

Figure 6 The semantics of MQuery stages.

$\mathsf{type}(p,t) = \mathsf{array}$ and $\mathsf{type}(x,t) = \mathsf{ty}$ for each $x \in [\![p.i]\!]^t$ for $i \in I$, we write $\mathsf{type}(p[\,],t) = \mathsf{ty}$.

Second, we define when a tree t satisfies a criterion or a Boolean value definition φ, denoted $t \models \varphi$, as follows. It always holds that $t \models \mathbf{true}$, while:

$t \models (p = v)$, if there is x in $[\![p]\!]^t$ or in $[\![p.i]\!]^t$ for $i \in I$ s.t. $\mathsf{value}(x,t) = v$ holds
$t \models (\exists p)$, if $[\![p]\!]^t \neq \emptyset$
$t \models \neg\varphi$, if $t \not\models \varphi$
$t \models (p_1 = p_2)$, if there is a value v s.t. $t \models (p_1 = v) \wedge (p_2 = v)$, or $t \models \neg(\exists p_1) \wedge \neg(\exists p_2)$
$t \models \varphi_1 \wedge \varphi_2$, if $t \models \varphi_1$ and $t \models \varphi_2$
$t \models \varphi_1 \vee \varphi_2$, if $t \models \varphi_1$ or $t \models \varphi_2$

In this definition, we employ the classical semantics for "deep" equality of non-literal values, where we ignore duplicates and order in arrays, in line with set semantics. We also assume that $(v = \mathbf{null})$ holds iff v is **null**. Note that, the equality $(p = v)$ may hold both when v is the array reached by p and when v is an element inside this array. E.g., $t_{\mathrm{KN}} \models (\mathsf{contribs} = [\text{"OOP"}, \text{"Simula"}])$ and $t_{\mathrm{KN}} \models (\mathsf{contribs} = \text{"OOP"})$. Also note that the values of several paths inside an array can come from different array elements. E.g., $t_{\mathrm{KN}} \models (\mathsf{awards.award} = \text{"Rosing Prize"}) \wedge (\mathsf{awards.year} = 2001)$.

Next, we define the evaluation of a value definition d in a tree t, denoted by $\mathsf{eval}(d,t)$, as:

d, if $d \in V$;
$\mathsf{subtree}(t,d)$, if d is a path;
$\mathsf{eval}(d',t)$, if $d = (c?\, d_1 \colon d_2)$, where $d' = d_1$ when $t \models c$ and $d' = d_2$ otherwise.
the value of $(t \models d)$, if d is a Boolean value definition;
$[\mathsf{eval}(d_1,t), \ldots, \mathsf{eval}(d_m,t)]$, if $d = [d_1, \ldots, d_m]$;

Finally, we informally introduce some auxiliary operators over trees (for a formal definition, see App. B). Let t, t_1, t_2 be trees, F a forest, p a path, and N a set of nodes. Then:

- $\mathsf{subtree}(t,p)$ returns the subtree of t rooted at the single node in $[\![p]\!]^t$ when $|[\![p]\!]^t| = 1$. Instead, when $|[\![p]\!]^t| > 1$, it returns the array of all subtrees rooted at the nodes in $[\![p]\!]^t$, and when $[\![p]\!]^t = \emptyset$, it returns **null**. E.g., $\mathsf{subtree}(t_{\mathrm{KN}}, \mathsf{name})$ returns {{first: "Kristen", last: "Nygaard"}}, and $\mathsf{subtree}(t_{\mathrm{KN}}, \mathsf{awards.year})$ returns [1999,2001,2001].
- $\mathsf{subtree}(t,N)$ returns the subtree (i.e., a subgraph) of t induced by the set N of nodes.
- $\mathsf{attach}(p,t)$ constructs a new tree by attaching t (via its root) to the end of the path p. E.g., $\mathsf{attach}(\mathsf{info.name}, \{\{\text{first: "Kristen"}\}\})$ returns {{info: {{name: {{first: "Kristen"}}}}}}.
- $\mathsf{replace}(t, t_1, t_2)$ constructs a new tree by replacing in t its subtree t_1 by a new tree t_2.
- $t_1 \oplus t_2$ constructs a new tree resulting from merging t_1 and t_2 by identifying nodes reachable via identical paths. E.g., {{name: {{first: "Kristen"}}}} $\oplus$ {{name: {{last: "Nygaard"}}}} returns {{name: {{first: "Kristen", last: "Nygaard"}}}}.
- $\mathsf{array}(F,p)$ constructs a new tree that is the array of all $\mathsf{subtree}(t,p)$ for $t \in F$, while $\mathsf{forest}(F,p)$ keeps all $\mathsf{subtree}(t,p)$ in a set.

Now, we are ready to define the semantics of the MQuery stages: specifically, given a

forest F and a stage s, we define the forest $F \triangleright s$ (for a lookup stage, we also require an additional forest F' as parameter), as shown in Figure 6. We observe that, for all operators except *group*, each tree in the input can be processed independently of the other trees, and gives rise to zero, one, or more trees in the output. Below, we provide some explanations:

Match. We just observe that *match* might produce an empty output.

Unwind. We say that a path p is a *first array* in t if $\mathsf{type}(p,t) = \mathsf{array}$ and $\mathsf{type}(p',t) \neq \mathsf{array}$, for each strict prefix p' of p. When p is a first array in t with value different from $[\,]$, then $\omega_p(t)$ contains one tree for each element in such an array, obtained by replacing in t the array by the element. In all other cases (i.e., when p is a first array in t and its value is $[\,]$, when p is missing in t, when $\mathsf{type}(p,t) \neq \mathsf{array}$, or when $\mathsf{type}(p,t) = \mathsf{array}$ but p is not a first array in t), we have that $\omega_p(t)$ is empty.

Project. ρ_P produces exactly one output tree from each input tree, obtained by applying to the input tree each of the elements in P, independently of the other elements. Note that, when $q/p \in P$ and p is missing in the input tree, then also q is missing in the output tree. Instead, for $q/[p] \in P$ with p missing, q is present in the output with value [**null**].

Group. In $\gamma_{G\,:\,A}$, when G is empty, i.e., $n = 0$, φ is the empty conjunction and hence true, so all input trees are grouped in one output tree where the value of `_id` is **null**. Instead, when $G = g_1/y_1, \ldots, g_n/y_n$ with $n \geq 1$, then the set of input trees is partitioned into "groups", where each group corresponds to a (possibly empty) subset Y of $\{y_1, \ldots, y_n\}$, so that the trees in the group agree not only on the respective values t_j reached through all the paths $y_j \in Y$, but also on the non-existence of paths not in Y. Each group $Y = \{y_{j_1}, \ldots, y_{j_k}\}$ gives rise to one output tree. In the case where $k > 0$ and all g_{j_i}s are keys, in the output tree the value of `_id` for that group is $\{\!\{g_{j_1}{:}t_{j_1}, \ldots, g_{j_k}{:}t_{j_k}\}\!\}$, and in the case where $k = 0$ (i.e., $Y = \emptyset$) the value of `_id` is **null**. Moreover, for each pair a_i/b_i in A, the values of b_i of all trees in a group are collected in an array, and such an array is inserted in the output tree for that group as the value of a_i.

Lookup. $\lambda_p^{p_1=C.p_2}[F']$ produces exactly one output tree from each input tree. Each such tree coincides with the input tree, except for one additional array containing all the trees of F' for which the value of p_2 coincides with the value of p_1 in the input tree.

We clarify what we mean by "employing set semantics". For every stage s and forest F, $F \triangleright s$ is a set of trees, i.e., contains no duplicates. Duplicates are detected by comparing trees using deep equality, where comparison of arrays ignores the element indexes. However trees might contain arrays with duplicates. Also, array indexes are sometimes important for merging trees correctly when computing the result of a project stage (see Example 24 in App. A.3).

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'.

Counter-intuitive Choices in the Semantics of MongoDB

We conclude this section by discussing some choices in the semantics of MongoDB that we consider counter-intuitive, and that could be considered as an inconsistency in the behavior of operators. Therefore, in MQuery, we have chosen a cleaner, more uniform semantics.

"Entering an array" when comparing value and path. In MongoDB, the satisfaction relation $t \models (p = v)$ behaves differently in *match* and in *project* when the type of p in t is array. In *match*, equality holds when v is *(1)* exactly the array value of p, or *(2)* an element inside the array value of p, while *project* checks only condition *(1)*. In MQuery, we take a uniform approach, in which $t \models (p = v)$ in *project* behaves as in *match*.

Null and missing values. In some cases, MongoDB does not distinguish *(a)* when the value of a path p is **null**, i.e., $[\![p]\!]^t = \{x\}$ and $\mathsf{value}(x,t) = \textbf{null}$, from *(b)* when p is missing in t. In particular, in *match* both *(a)* and *(b)* imply that $t \models (p = \textbf{null})$. Instead, in *project*, only *(a)* implies it. Similarly, in *group*, when grouping by one path (e.g., $\gamma_{g/p:A}$), MongoDB puts the trees satisfying *(a)* and *(b)* into the same group (having $\texttt{\_id} = \{\!\{g : \textbf{null}\}\!\}$). Instead, when grouping with multiple paths (e.g., $\gamma_{g_1/p_1,g_2/p_2,\ldots:A}$), the trees with all p_i missing are put into a separate group (having $\texttt{\_id} = \{\!\{\}\!\}$). In MQuery, instead, we systematically distinguish the cases *(a)* and *(b)*.

Comparison of values. In MongoDB, equality of non-literal values is determined by comparing their binary representation[5]. Hence, two objects with the same key-value pairs but in different orders, will not be considered the same, which might result in missed answers. In MQuery, we employ the classical semantics for "deep" equality of non-literal values.

5 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.

5.1 Nested Relational View of MongoDB

We start by defining a nested relational view of MongoDB instances. 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. We start by introducing the notion of *type* of a tree, which is analogous to complex object types [13], and similar to JSON schema [15].

▶ **Definition 2.** Consider JSON values constructed according to the following grammar:

TYPE ::= literal | {{LIST<KEY:TYPE>}} | [TYPE]

Given such a JSON value d, we call the tree $\mathsf{tree}(d)$ a *type*. We say that a tree t is *of type* τ if for every path p we have that $t \models \exists p$ implies *(i)* $\tau \models \exists p$, *(ii)* $\mathsf{type}(p,t) = \mathsf{type}(p,\tau)$, and *(iii)* $\mathsf{type}(p[\,],t) = \mathsf{type}(p[\,],\tau)$. A forest F is *of type* τ if all trees in F are of type τ. A forest (resp., tree) is *well-typed* if it is of some type.

We now 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. In the following definition, given paths p and q, we say that $p.q$ is a *simple extension* of p if there is no strict prefix q' of q such that $\mathsf{type}(p.q',\tau) = \mathsf{array}$.

▶ **Definition 3.** For a type τ, the *relation schema* $\mathsf{rschema}(\tau)$, is defined as $R_\tau(\mathsf{ratt}_\tau(\varepsilon))$, where, for a path p in τ, $\mathsf{ratt}_\tau(p)$ is the set of simple extensions p' of p such that p' is an *atomic attribute* if $\mathsf{type}(p',\tau) = \mathsf{literal}$, and p' is a *sub-relation* if $\mathsf{type}(p',\tau) = \mathsf{array}$. In the latter case, p' has attributes $\{p'.\texttt{lit}\}$ if $\mathsf{type}(p'[\,],\tau) = \mathsf{literal}$, and $\mathsf{ratt}_\tau(p')$ otherwise.

Observe that 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.

Next, we define the relational view of a well-typed forest. In this view, to capture the semantics of the missing paths, we introduce the new constant **missing**.

[5] https://docs.mongodb.org/manual/reference/bson-types/#comparison-sort-order

_id	awards		birth	contribs	name.first	name.last
	awards.award	awards.year		contribs.lit		
4	Rosing Prize	1999	1926-08-27	OOP	Kristen	Nygaard
	Turing Award	2001		Simula		
	IEEE John von Neumann Medal	2001				

Figure 7 Relational view of the document about Kristen Nygaard.

▶ **Definition 4.** The *relational view* of a well-typed forest F, denoted $\mathsf{rel}(F)$, is defined as $\{\mathsf{rtuple}_\tau(R_\tau, \varepsilon, t) \mid t \in F\}$, where τ is the type of F. For a relation name R in $\mathsf{rschema}(\tau)$ and a path p, $\mathsf{rtuple}_\tau(R, p, t)$ is the R-tuple $\{p.q : \mathsf{rval}(p.q, t)\}_{p.q \in \mathsf{ratt}_\tau(p)}$, where

$$\mathsf{rval}(p.q,t) = \begin{cases} \textbf{missing}, & \text{if } [\![q]\!]^t = \emptyset; \\ \mathsf{value}(\mathsf{subtree}(t,q)), & \text{if } p.q \text{ is atomic;} \\ \{(p.q.\mathtt{lit} : \mathsf{value}(\mathsf{subtree}(t,q.i))) \mid [\![q.i]\!]^t \neq \emptyset, \text{ for } i \in I\}, & \text{if } \mathsf{att}_\tau(p.q) = \{p.q.\mathtt{lit}\}; \\ \{\mathsf{rtuple}_\tau\,(p.q, p.q, \mathsf{subtree}(t,q.i)) \mid [\![q.i]\!]^t \neq \emptyset, \text{ for } i \in I\}, & \text{otherwise.} \end{cases}$$

▶ **Example 5.** Consider the type $\tau_{\mathtt{bios}}$ for `bios`:

```
{ "_id": "literal",    "awards": [ { "award": "literal", "year": "literal" } ],
  "birth": "literal",  "contribs": [ "literal" ],
  "name": { "first": "literal", "last": "literal" } }
```

Then, $\mathsf{rschema}(\tau_{\mathtt{bios}})$ is defined as `bios`(`_id`, awards(awards.award, awards.year), birth, contribs(contribs.`lit`), name.first, name.last). Moreover, for the tree t in Figure 4, the relational view $\mathsf{rel}(\{t\})$ is illustrated in Figure 7. ◀

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.

▶ **Definition 6.** Let $\mathcal{S}$ be a set of type constraints, and D a MongoDB instance 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.

▶ **Definition 7.** 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 $\mathcal{S}$.

Notice that the above definition of equivalence between well-typed forests and nested relations appears to be asymmetric, since it would in principle allow for nested relations that are not equivalent to any well-typed forest. We remark, however, that the MongoDB view of a nested relation always exists, is well-typed, and can be defined in a straightforward way. Therefore, we can consider both translations (from NRA to MQuery, and vice-versa), as defined on well-typed forests and their relational views.

5.2 From NRA to MQuery

We now show that $\mathcal{M}^{\mathrm{MUPGL}}$ captures NRA, while $\mathcal{M}^{\mathrm{MUPG}}$ 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

s	$\mathsf{subq}_j(s)$	s	$\mathsf{subq}_j(s)$
μ_φ	$\mu_{(\text{actRel}=3-j)\vee\varphi_{[p\to\text{rel}j.p]}}$	$\gamma_{g/y:a/b}$	$\gamma_{g/\text{rel}j.y,\,\text{actRel}:a/\text{rel}j.b,\,\text{rel}(3-j)}\triangleright$
ω_p	$\rho_{\text{actRel},\,\text{rel}(3-j),\,\text{rel}j/((\text{actRel}=j)?\,\text{rel}j:\,\{\!\{p:[0]\}\!\})}$ $\triangleright\,\omega_{\text{rel}j.p}\triangleright\,\mathsf{norm}$		$\rho_{\text{actRel}/\texttt{\_id}.\text{actRel},\,\text{rel}j.\texttt{\_id}.g/\texttt{\_id}.g,\,\text{rel}j.a/a,\,\text{rel}(3-j)}$ $\triangleright\rho_{\text{actRel},\,\text{rel}j,\,\text{rel}(3-j)/((\text{actRel}=3-j)?\,\text{rel}(3-j):[0])}$ $\triangleright\,\omega_{\text{rel}(3-j)}\triangleright\,\mathsf{norm}$
$\rho_{p,\,q/d}$	$\rho_{\text{actRel},\,\text{rel}(3-j),\,\text{rel}j.p,}$ $\text{rel}j.q/((\text{actRel}=j)?\,d_{[q'\to\text{rel}j.q']}:\text{dummy})$		

Figure 8 Subquery $\mathsf{subq}_j(s)$ for stage s, where we have detailed only the short forms for project and group stages. We use $e_{[p\to q]}$ to denote the expression e in which every occurrence of the path p is replaced by the path q, and use norm to abbreviate $\rho_{\text{actRel},\,\{\text{rel}i/((\text{actRel}=i)?\,\text{rel}i:\,\text{dummy})\}_{i=1,2}}$.

contains *one sequence* of stages. So, we first show how to "linearize" tree-shaped NRA expressions into a MongoDB pipeline. More precisely, we 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. We define $\mathsf{pipeline}(\boldsymbol{q}_1,\boldsymbol{q}_2)$ as $\mathsf{dup}\triangleright\mathsf{subq}_1(\boldsymbol{q}_1)\triangleright\mathsf{subq}_2(\boldsymbol{q}_2)$.

The idea of dup is to create for each tree t of the input forest two trees differentiated by an ad-hoc key-value pair $\texttt{actRel}:j$ and storing the original tree as $\texttt{rel}j:t$, for $j\in\{1,2\}$. More precisely, we want to obtain for each forest F that $F\triangleright\mathsf{dup}=F_1\cup F_2$, where $F_1=\{\!\{\text{actRel}:1,\,\text{rel1}:t\}\!\}_{t\in F}$ and $F_2=\{\!\{\text{actRel}:2,\,\text{rel2}:t\}\!\}_{t\in F}$. This is achieved by setting $\mathsf{dup}=\rho_{\text{origDoc}/\varepsilon,\,\text{actRel}/[1,2]}\triangleright\omega_{\text{actRel}}\triangleright\rho_{\text{actRel},\,\{\text{rel}j/((\text{actRel}=j)?\,\text{origDoc}:\,\text{dummy})\}_{j=1,2}}$, where $\texttt{dummy}$ is a path that does not exist in any collection.

The idea of $\mathsf{subq}_j(\boldsymbol{q}_j)$ is to execute $\boldsymbol{q}_j$ so that it affects the trees from F_j, but not from F_{3-j}, and to obtain that $(F_1\cup F_2)\triangleright\mathsf{subq}_1(\boldsymbol{q}_1)\triangleright\mathsf{subq}_2(\boldsymbol{q}_2)$ evaluates to the forest $\{\!\{\text{actRel}:1,\,\text{rel1}:t\}\!\}_{t\in(F\triangleright\boldsymbol{q}_1)}\cup\{\!\{\text{actRel}:2,\,\text{rel2}:t\}\!\}_{t\in(F\triangleright\boldsymbol{q}_2)}$. Before describing subq_j formally, we provide the intuition in an example.

▶ **Example 8.** Consider the sequences of stages $\boldsymbol{q}_1=\mu_{a=1}\triangleright\rho_{a,\,b}$ and $\boldsymbol{q}_2=\mu_{c=\text{"x"}}\triangleright\rho_{c,\,d}$, and the forest $F=\{t_1,t_2,t_3,t_4\}$, for $t_1=\{\!\{\text{a:1, b:6, d:8}\}\!\}$, $t_2=\{\!\{\text{a:1, b:7, c:"x", d:9}\}\!\}$, $t_3=\{\!\{\text{a:2, b:6, c:"x", d:7}\}\!\}$, and $t_4=\{\!\{\text{a:3, b:8, c:"y", d:6}\}\!\}$.

Denote by t_i^{pq} the tree resulting from t_i by applying $\rho_{p,\,q}$ to it. Then $F\triangleright\boldsymbol{q}_1$ evaluates to $\{t_1^{ab},t_2^{ab}\}$, and $F\triangleright\boldsymbol{q}_2$ to $\{t_2^{cd},t_3^{cd}\}$. Thus, $(F_1\cup F_2)\triangleright\mathsf{subq}_1(\boldsymbol{q}_1)\triangleright\mathsf{subq}_2(\boldsymbol{q}_2)$ should be $\{\{\!\{\text{actRel}:1,\,\text{rel1}:t_1^{ab}\}\!\},\{\!\{\text{actRel}:1,\,\text{rel1}:t_2^{ab}\}\!\},\{\!\{\text{actRel}:2,\,\text{rel2}:t_2^{cd}\}\!\},\{\!\{\text{actRel}:2,\,\text{rel2}:t_3^{cd}\}\!\}\}$. We achieve this by setting $\mathsf{subq}_1(\boldsymbol{q}_1)=\mu_{(\text{actRel}=2)\vee(\text{rel1}.a=1)}\triangleright\rho_{\text{rel2},\,\text{actRel},\,\text{rel1}.a,\,\text{rel1}.b}$ and $\mathsf{subq}_2(\boldsymbol{q}_2)=\mu_{(\text{actRel}=1)\vee(\text{rel2}.c=\text{"x"})}\triangleright\rho_{\text{rel1},\,\text{actRel},\,\text{rel2}.c,\,\text{rel2}.d}$. Here, in the case of $\mathsf{subq}_1(\boldsymbol{q}_1)$, by transforming the criterion $(a=1)$ into $(\text{actRel}=2)\vee(\text{rel1}.a=1)$ we make sure to preserve the trees in F_2, and we check the condition on the correct path $\texttt{rel1.a}$ in the trees in F_1. Similarly, the project stage $\rho_{a,b}$ became $\rho_{\text{rel2},\,\text{actRel},\,\text{rel1}.a,\,\text{rel1}.b}$ in order to preserve the t_is in F_2 stored under $\texttt{rel2}$, to preserve the auxiliary key $\texttt{actRel}$, and to project the correct paths $\texttt{rel1.a}$ and $\texttt{rel1.b}$ in the trees from F_1. ◀

Formally, when $\boldsymbol{q}_j=s_1\triangleright\cdots\triangleright s_n$ then $\mathsf{subq}_j(\boldsymbol{q}_j)$ is defined as $\mathsf{subq}_j(s_1)\triangleright\cdots\triangleright\mathsf{subq}_j(s_n)$, for $j\in\{1,2\}$, where subq_j for single stages is defined in Figure 8. Recall that the idea of $\mathsf{subq}_j(s)$ is to affect only the trees in F_j, hence:

- $\mathsf{subq}_j(\mu_\varphi)$ selects all trees in F_{3-j} (those that satisfy $(\text{actRel}=3-j)$), while from F_j it selects the trees that satisfy φ, where all original paths p are replaced by $\texttt{rel}j.p$.
- The unwind stage ω_p cannot be implemented simply by $\omega_{\text{rel}j.p}$, since all trees in F_{3-j} would be lost (they do not contain the path $\texttt{rel}j.p$). Therefore we first create a temporary non-empty array ($[0]$) under the path $\texttt{rel}j.p$ in the trees from F_{3-j}, unwind the path $\texttt{rel}j.p$, and then in norm normalize the trees by making sure that the trees with $(\text{actRel}=i)$ contain only $\texttt{rel}i$ but not $\texttt{rel}(3-i)$.

Q	$\mathsf{nra2mq}(Q)$
C	$\rho_{\mathsf{att}(C)}$
$\sigma_\psi(Q_1)$	$\mathsf{nra2mq}(Q_1) \triangleright \rho_{\mathsf{att}(Q_1),\,\mathrm{cond}/\psi} \triangleright \mu_{\mathrm{cond}=\mathbf{true}} \triangleright \rho_{\mathsf{att}(Q_1)}$
$\pi_S(Q_1)$	$\mathsf{nra2mq}(Q_1) \triangleright \rho_S$
$\nu_{S\to b}(Q_1)$	$\mathsf{nra2mq}(Q_1) \triangleright \rho_{(\mathsf{att}(Q_1)\setminus S),\,\{b.p/p \mid p\in S\}} \triangleright \gamma_{(\mathsf{att}(Q_1)\setminus S)\,:\,b} \triangleright \rho_{b,\,\{p/\texttt{\_id}.p \mid p\in\mathsf{att}(Q_1)\setminus S\}}$
$\chi_a(Q_1)$	$\mathsf{nra2mq}(Q_1) \triangleright \omega_a$
$Q_1 \times Q_2$	$\mathsf{pipeline}(\mathsf{nra2mq}(Q_1), \mathsf{nra2mq}(Q_2)) \triangleright \gamma_{\,:\,\mathrm{rel1},\,\mathrm{rel2}} \triangleright \omega_{\mathrm{rel1}} \triangleright \omega_{\mathrm{rel2}}$
$Q_1 \cup Q_2$	$\mathsf{pipeline}(\mathsf{nra2mq}(Q_1), \mathsf{nra2mq}(Q_2)) \triangleright \rho_{\{\mathrm{p}i/((\mathrm{actRel}=1)?\,\mathrm{rel1}.\mathrm{p}i:\,\mathrm{rel2}.\mathrm{p}i)\}_{i=1}^n} \triangleright \gamma_{\mathrm{p1},\ldots,\mathrm{p}n\,:} \triangleright \rho_{\{\mathrm{p}i/\texttt{\_id}.\mathrm{p}i\}_{i=1}^n}$
$Q_1 \setminus Q_2$	$\mathsf{pipeline}(\mathsf{nra2mq}(Q_1), \mathsf{nra2mq}(Q_2)) \triangleright \rho_{\mathrm{rel2},\,\{\mathrm{p}i/((\mathrm{actRel}=1)?\,\mathrm{rel1}.\mathrm{p}i:\,\mathrm{rel2}.\mathrm{p}i)\}_{i=1}^n} \triangleright \gamma_{\mathrm{p1},\ldots,\mathrm{p}n\,:\,\mathrm{rel2}} \triangleright \mu_{\mathrm{rel2}=[]} \triangleright \rho_{\{\mathrm{p}i/\texttt{\_id}.\mathrm{p}i\}_{i=1}^n}$

Figure 9 Translation from NRA to $\mathcal{M}^{\mathrm{MUPG}}$. We extend the function att from schema names to NRA queries such that $\mathsf{att}(Q)$ is the attribute set of the schema implied by an NRA query Q.

- The encoding of the project stage $\rho_{p,\,q/d}$ makes sure that $\texttt{rel}(3-j)$ and $\texttt{actRel}$ are not lost, and that the path $\texttt{rel}j.q$ is not created in the trees from F_{3-j} (guaranteed by the conditional expression for q/d).
- The encoding of the group stage $\gamma_{g/y\,:\,a/b}$ adds $\texttt{actRel}$ to the grouping condition and aggregates $\texttt{rel}(3-j)$ so as to group all trees from F_{3-j} in one tree, and then renames appropriately the paths $\texttt{\_id.actRel}$, $\texttt{\_id.g}$, and $\texttt{a}$. It is concluded similarly to $\mathsf{subq}_j(\omega_p)$ in order to flatten the array $\texttt{rel}(3-j)$ where all trees from F_{3-j} have been aggregated.

Now, having defined $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$, we are ready to show how to translate NRA to MQuery. We start with a singleton set $\mathcal{S} = \{(C, \tau_C)\}$ of type constraints for a collection name C, and consider an NRA query Q over the relation name C (with schema $\mathsf{rschema}(\tau_C)$). The translation of Q is the $\mathcal{M}^{\mathrm{MUPG}}$ query $C \triangleright \mathsf{nra2mq}(Q)$, where $\mathsf{nra2mq}(Q)$ is defined inductively in Figure 9. To encode select, the filter is translated as a Boolean value definition, except that atoms of the form $\neg(a = \mathbf{missing})$ become $\exists a$. The translation of $Q_1 \times Q_2$ first groups all input trees in one tree, where the answer trees t_i to Q_i are aggregated in arrays $\texttt{rel}i$, $i \in \{1, 2\}$, and then unwinds these two arrays, thus producing all possible pairs (t_1, t_2). The translations of $Q_1 \cup Q_2$ and $Q_1 \setminus Q_2$, where we assume that $\mathsf{att}(Q_i) = \{\mathrm{p1}, \ldots, \mathrm{p}n\}$, first create fresh paths $\mathrm{p}i$ in each tree to be used in the grouping condition. Then, in the case of union it only remains to rename the paths $\texttt{\_id.}\mathrm{p}i$ back to $\mathrm{p}i$, while in the case of difference, we also select only those "tuples" that were not present in the answer to Q_2.

▶ **Example 9.** Consider the forest F from Example 8 stored under a collection name C, and the type $\tau_C = \{\!\{\mathrm{a}\colon \mathrm{literal},\ \mathrm{b}\colon \mathrm{literal},\ \mathrm{c}\colon \mathrm{literal},\ \mathrm{d}\colon \mathrm{literal}\}\!\}$. Then $\mathsf{rschema}(\tau_C)$ is defined as $C(\mathrm{a},\mathrm{b},\mathrm{c},\mathrm{d})$, and $\mathsf{rel}(F)$ is the relation $\{(\mathrm{a}\colon 1,\ \mathrm{b}\colon 6,\ \mathrm{c}\colon \mathbf{missing},\ \mathrm{d}\colon 8),\ (\mathrm{a}\colon 1,\ \mathrm{b}\colon 7,\ \mathrm{c}\colon \text{"x"},\ \mathrm{d}\colon 9),\ (\mathrm{a}\colon 2,\ \mathrm{b}\colon 6,\ \mathrm{c}\colon \text{"x"},\ \mathrm{d}\colon 7),\ (\mathrm{a}\colon 3,\ \mathrm{b}\colon 8,\ \mathrm{c}\colon \text{"y"},\ \mathrm{d}\colon 6)\}$. Let Q be the NRA query $\sigma_{\mathrm{rel1}.b=\mathrm{rel2}.d}\big(Q_1 \times Q_2\big)$, where $Q_1 = \pi_{a,b}(\sigma_{a=1}(C))$ and $Q_2 = \pi_{c,d}(\sigma_{c=\text{"x"}}(C))$. Then Q evaluated over $\mathsf{rel}(F)$ returns $\{(\mathrm{rel1.a}\colon 1,\ \mathrm{rel1.b}\colon 7,\ \mathrm{rel2.c}\colon \text{"x"},\ \mathrm{rel2.d}\colon 7)\}$.

Now, $\mathsf{nra2mq}(Q_j) = \rho_{a,\,b,\,c,\,d} \triangleright \boldsymbol{q}_j$, for $j = 1, 2$, where $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ are as in Example 8. Since $\rho_{a,\,b,\,c,\,d} \triangleright \boldsymbol{q}_j$ and $\boldsymbol{q}_j$ are equivalent (return the same answers over all forests), we have that $F \triangleright \mathsf{pipeline}(\mathsf{nra2mq}(Q_1), \mathsf{nra2mq}(Q_2)) = F \triangleright \mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$. Denote by F' the result of $F \triangleright \mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$, see Example 8. Then

- $F'' = F' \triangleright \gamma_{\,:\,\mathrm{rel1},\,\mathrm{rel2}}$ is the forest $\{\{\!\{\mathrm{rel1}\colon [t_1^{ab}, t_2^{ab}],\ \mathrm{rel2}\colon [t_2^{cd}, t_3^{cd}]\}\!\}\}$.
- $F''' = F'' \triangleright \omega_{\mathrm{rel1}} \triangleright \omega_{\mathrm{rel2}}$ is the forest $\{\{\!\{\mathrm{rel1}\colon t_1^{ab},\ \mathrm{rel2}\colon t_2^{cd}\}\!\},\ \{\!\{\mathrm{rel1}\colon t_2^{ab},\ \mathrm{rel2}\colon t_2^{cd}\}\!\},\ \{\!\{\mathrm{rel1}\colon t_1^{ab},\ \mathrm{rel2}\colon t_3^{cd}\}\!\},\ \{\!\{\mathrm{rel1}\colon t_2^{ab},\ \mathrm{rel2}\colon t_3^{cd}\}\!\}\}$.
- Finally, $F''' \triangleright \rho_{\mathrm{cond}/(\mathrm{rel1.b}=\mathrm{rel2.d})} \triangleright \mu_{\mathrm{cond}=\mathbf{true}} \triangleright \rho_{\mathrm{rel1.a},\,\mathrm{rel1.b},\,\mathrm{rel2.c},\,\mathrm{rel2.d}}$ is the forest $\{\{\!\{\mathrm{rel1}\colon t_2^{ab},\ \mathrm{rel2}\colon t_3^{cd}\}\!\}\}$, or equivalently $\{\{\!\{\mathrm{rel1}\colon \{\!\{\mathrm{a}\colon 1,\ \mathrm{b}\colon 7\}\!\},\ \mathrm{rel2}\colon \{\!\{\mathrm{c}\colon \text{"x"},\ \mathrm{d}\colon 7\}\!\}\}\!\}\}$. ◀

▶ **Theorem 10.** *Let Q be a NRA query over C. Then $C \triangleright \mathsf{nra2mq}(Q) \equiv_{\mathcal{S}} Q$.*

Next, we consider NRA queries across several collections, and show how to translate them to $\mathcal{M}^{\text{MUPGL}}$. Let $\mathcal{S}$ be a set of type constraints for collection names $C_1, \ldots, C_n$, with $n \geq 2$, Q an NRA query over $C_1, \ldots, C_n$, and C_1 the collection over which we evaluate the generated MQuery. The translation of Q is the $\mathcal{M}^{\text{MUPGL}}$ query $C_1 \triangleright \mathsf{bring}(C_2, \ldots, C_n) \triangleright \mathsf{nra2mq}^\star(Q)$, where intuitively (1) the phase $\mathsf{bring}(C_2, \ldots, C_n)$ "brings in" the trees from the collections $C_2, \ldots, C_n$, and (2) the function $\mathsf{nra2mq}^\star(Q)$, adapted from $\mathsf{nra2mq}(Q)$, simulates the NRA operators in Q. More precisely, we want that if $F_1, \ldots, F_n$ are collections for $C_1, \ldots, C_n$, the result of $F_1 \triangleright \mathsf{bring}(C_2, \ldots, C_n)[F_2, \ldots, F_n]$ is the forest $\bigcup_{i=1}^{n} \{\!\{\text{actColl}\colon i,\ \text{coll}i\colon t\}\!\}_{t \in F_i}$. This is done by setting $\mathsf{bring}(C_2, \ldots, C_n)$ as

$$\gamma_{\colon \text{coll1}/\varepsilon} \triangleright \lambda^{\text{dummy}=C_2.\text{dummy}}_{\text{coll2}} \triangleright \cdots \triangleright \lambda^{\text{dummy}=C_n.\text{dummy}}_{\text{colln}} \triangleright \rho_{\text{coll1},..,\text{colln},\,\text{actColl}/[1..n]} \triangleright \omega_{\text{actColl}} \triangleright$$
$$\rho_{\text{actColl},\,\{\text{coll}i/((\text{actColl}=i)?\,\text{coll}i\colon [0])\}_{i=1}^{n}} \triangleright \omega_{\text{coll1}} \triangleright \cdots \triangleright \omega_{\text{colln}} \triangleright \rho_{\text{actColl},\,\{\text{coll}i/((\text{actColl}=i)?\,\text{coll}i\colon \text{dummy})\}_{i=1}^{n}}$$

Moreover, we define the function $\mathsf{nra2mq}^\star(Q)$ that differs from $\mathsf{nra2mq}(Q)$ in the translation of the collection names as $\mathsf{nra2mq}^\star(C_i) = \mu_{\text{actColl}=i} \triangleright \rho_{\{p/\text{coll}i.p \,\mid\, p \in \mathsf{att}(C_i)\}}$.

▶ **Theorem 11.** *Let Q be an NRA query over $C_1, \ldots, C_n$, and $\boldsymbol{q} = C_1 \triangleright \mathsf{bring}(C_2, \ldots, C_n) \triangleright \mathsf{nra2mq}^\star(Q)$. Then $\boldsymbol{q} \equiv_{\mathcal{S}} Q$. Moreover, the size of $\boldsymbol{q}$ is polynomial in the size of Q.*

Thus, we obtain that $\mathcal{M}^{\text{MUPGL}}$ captures full NRA, and that $\mathcal{M}^{\text{MUPG}}$ captures NRA over a single collection. We observe that the above translation serves the purpose of understanding the expressive power of MQuery, but is likely to produce queries that MongoDB will not be able to efficiently execute in practice, even on relatively small database instances. We also note that the translation from NRA to MQuery works even if we allow for database instances D such that $D.C$ is not strictly of type τ_C, but may also contain other paths not in τ_C.

5.3 From MQuery to NRA

In this section, we aim at defining a translation from MQuery to NRA, and for this 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 $C \circ \mathsf{mq2nra}(s_1) \circ \cdots \circ \mathsf{mq2nra}(s_n)$[6], 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. This capability is due to value definitions in a project operator: already a query as simple as $\rho_{\text{a}/(\texttt{\_id}=1?\,[0,1]\colon \texttt{"s"})}$ produces from the well-typed forest $\{\{\!\{\texttt{\_id}\colon 1\}\!\},\ \{\!\{\texttt{\_id}\colon 2\}\!\}\}$ a non well-typed one: $\{\{\!\{\texttt{\_id}\colon 1,\ \text{a}\colon [0,1]\}\!\},\ \{\!\{\texttt{\_id}\colon 2,\ \text{a}\colon \texttt{"s"}\}\!\}\}$. Therefore, in order to derive such a translation $\mathsf{mq2nra}(s)$, we restrict our attention to MQueries with stages preserving well-typedness.

▶ **Definition 12.** Given a type τ (and a type τ'), a stage s is *well-typed* for τ (and τ'), if for each forest F of type τ (and each forest F' of type τ'), $F \triangleright s$ (resp., $F \triangleright s[F']$ when s is a lookup stage) is a well-typed forest.

We observe that the match, unwind, group and lookup stages are always well-typed, and, given such a stage s and input types τ, τ', we can compute the output type τ_o of s: *(i)* match does not change the input type, i.e., $\tau_o = \tau$, *(ii)* for unwind and group stages s, τ_o is obtained by evaluating s over $\{\tau\}$, i.e., $\{\tau_o\} = \{\tau\} \triangleright s$, and *(iii)* similarly, the output type for a lookup stage is the single tree in $(\{\tau\} \triangleright \lambda^{p_1=C.p_2}_{p}[\{\tau'\}])$. As for a project stage $s = \rho_P$ and an input type τ, we can check whether s is well-typed for τ, and if yes, we can compute the output

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

type τ_o of s, as follows. For each $p/d \in P$, we compute the type τ_d of d with respect to τ; if all τ_d are defined, then s is well-typed and τ_o is the type where $\mathsf{subtree}(\tau_o, p)$ coincides with τ_d for each $p/d \in P$, and that agrees with τ on all $p \in P$; otherwise s is not well-typed. The *type* τ_d of a value definition d with respect to a type τ is defined inductively as follows: $\tau_v = \tau'$ for a value v, if v is of type τ', and undefined otherwise; $\tau_\beta = \mathsf{literal}$ for a Boolean value definition β; $\tau_p = \mathsf{subtree}(\tau, p)$, for a path p; $\tau_{[d_1,\ldots,d_n]} = [\tau_{d_1}]$ if $\tau_{d_1} = \cdots = \tau_{d_n}$, and undefined otherwise; $\tau_{(c?\, d_1 :\, d_2)}$ is τ_{d_1} if c is valid, τ_{d_2} if c is unsatisfiable, τ_{d_1} if c is satisfiable and not valid and $\tau_{d_1} = \tau_{d_2}$, and undefined otherwise.

Then, given a set $\mathcal{S}$ of type constraints and an MQuery $\boldsymbol{q} = C \triangleright s_1 \triangleright \cdots \triangleright s_n$, we can check whether each stage in $\boldsymbol{q}$ is well-typed for its input type determined by $\boldsymbol{q}$ and $\mathcal{S}$. To do so, we take the input type for s_1 to be τ_0, where $(C, \tau_0) \in \mathcal{S}$, and we compute sequentially the input type for each stage s_i, as long as this is possible, i.e., all stages preceding it are well-typed.

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 reported in [3].

▶ **Theorem 13.** *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 $\mathcal{S}$.*

A natural question is when an MQuery can be translated to NRA even if it contains non well-typed stages. E.g., in the example above, this can happen when the path `a` is projected away in the subsequent stages without being actually used. We leave this for future work.

6 Complexity of MQuery

In this section we report results on the complexity of different fragments of MQuery. Specifically, we are concerned with the combined and query complexity of the *Boolean query evaluation* problem (i.e., the problem of checking non-emptiness of query answers).

We first establish that $\mathcal{M}^{\text{MUPGL}}$ and $\mathcal{M}^{\text{MUPG}}$ are complete for exponential time with a polynomial number of alternations under LogSpace reductions [6, 12][7]. That is, they have the same complexity as monad algebra with atomic equality and negation [13], which however is strictly less expressive than NRA. As a corollary, we obtain a tight bound for NRA.

▶ **Theorem 14.** *$\mathcal{M}^{\text{MUPG}}$ and $\mathcal{M}^{\text{MUPGL}}$ are $\text{TA}[2^{n^{O(1)}}, n^{O(1)}]$-complete in combined complexity, and in AC^0 in data complexity.*

▶ **Corollary 15.** *NRA is $\text{TA}[2^{n^{O(1)}}, n^{O(1)}]$-complete in combined complexity.*

Next, we study some of the less expressive fragments of MQuery. We consider match to be an essential operator, and we start with the minimal fragment $\mathcal{M}^{\text{M}}$, for which we show that query answering is tractable and very efficient.

▶ **Theorem 16.** *$\mathcal{M}^{\text{M}}$ is LogSpace-complete in combined complexity.*

The project and group operators allow one to create exponentially large values by duplicating the existing ones. For instance, the result of $\{\{\!\{a{:}1\}\!\}\} \triangleright s_1 \triangleright \cdots \triangleright s_n$, for $s_1 = \cdots = s_n = \rho_{a.\ell/a,\, a.r/a}$ consists of a full binary tree of depth n. Nevertheless, without the unwind operator it is still possible to maintain tractability.

▶ **Theorem 17.** *$\mathcal{M}^{\text{MP}}$ is PTime-hard in query complexity and $\mathcal{M}^{\text{MPGL}}$ is in PTime in combined complexity.*

[7] We observe that $\text{TA}[2^{n^{O(1)}}, n^{O(1)}]$ lies between NExpTime and ExpSpace, hence is provably intractable.

We can identify the unwind operator as one of the sources of complexity, as it allows one to multiply the number of trees each time it is used in the pipeline. Indeed, adding the unwind operator alone causes already loss of tractability, provided the input tree contains multiple arrays (hence in combined complexity).

▶ **Theorem 18.** *$\mathcal{M}^{\text{MU}}$ is* LogSpace*-complete in query complexity and* NP*-complete in combined complexity.*

Adding project and lookup does not increase the combined complexity, but does increase the query complexity, since they allow for creating multiple arrays from a fixed input tree.

▶ **Theorem 19.** *$\mathcal{M}^{\text{MUP}}$ and $\mathcal{M}^{\text{MUL}}$ are* NP*-hard in query complexity, and $\mathcal{M}^{\text{MUPL}}$ is in* NP *in combined complexity.*

In the presence of unwind, group provides another source of complexity, since in $\mathcal{M}^{\text{MUG}}$ we can generate doubly exponentially large trees, analogously to monad algebra [13]. Let $t_0 = \{\!\{\texttt{\_id}\colon \{\!\{\text{x}\colon 0\}\!\}\}\!\}$ and $t_1 = \{\!\{\texttt{\_id}\colon \{\!\{\text{x}\colon 1\}\!\}\}\!\}$. The result of applying the $\mathcal{M}^{\text{MUG}}$ query $s_1 \triangleright \cdots \triangleright s_n$, where $s_i = \gamma_{\texttt{:}x/\texttt{\_id}.x} \triangleright \gamma_{x.l/x,\, x.r/x\texttt{:}} \triangleright \omega_{\texttt{\_id}.x.l} \triangleright \omega_{\texttt{\_id}.x.r}$, to $\{t_0, t_1\}$ is a forest containing 2^{2^n} trees, each encoding one 2^n-bit value. Below we show that already $\mathcal{M}^{\text{MUG}}$ queries are PSpace-hard.

▶ **Theorem 20.** *$\mathcal{M}^{\text{MUG}}$ is* PSpace*-hard in query complexity.*

7 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 expressive query language for JSON. We proposed a clean abstraction for its five main operators, which we called MQuery. Our formalization focuses on set semantics and, similarly to [10], ignores ordering; bag and list semantics are left for future work. MQuery also "polishes" some counter-intuitive aspects in the syntax and semantics of the actual aggregation framework, which are inherited from its ad-hoc development. We believe that these last changes, which are independent of our simplifying assumptions, make the framework more uniform, and we consequently encourage the designers of MongoDB to adopt them.

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, and hence is worth attention from the DB theory community. We hope that our study will also clarify the apparent confusion among practitioners about its capabilities to perform joins, in particular in the absence of lookup. 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.

With version v3.4, MongoDB has been extended with a *graph-lookup* stage in a pipeline, allowing for a recursive search on a collection, and it is of interest to understand how this affects formal and computational properties. We also propose to investigate the properties of MQuery when the well-typedness restrictions are lifted, and to compare it to JLogic [10], which is likewise able to handle flexible types. We are currently working on applying the results presented here, to provide high-level access to MongoDB data sources by relying on the standard ontology-based data access (OBDA) paradigm [16]. For this, we build on the translation from NRA to MQuery presented in Section 5.2 [2].

Acknowledgements. We thank Christoph Koch, Dan Suciu, Henrik Ingo, and Martin Rezk for helpful discussions. 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. Serge Abiteboul and Catriel Beeri. The power of languages for the manipulation of complex values. *Very Large Database Journal*, 4(4):727–794, 1995. URL: `http://www.vldb.org/journal/VLDBJ4/P727.pdf`.
2. Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Martin Rezk, and Guohui Xiao. OBDA beyond relational DBs: A study for MongoDB. In *Proc. of the 29th Int. Workshop on Description Logics (DL)*, volume 1577 of *CEUR Workshop Proceedings,* `http://ceur-ws.org/`, 2016.
3. Elena Botoeva, Diego Calvanese, Benjamin Cogrel, and Guohui 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. Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoč. JSON: Data model, query languages and schema specification. In *Proc. of the 36th ACM Symp. on Principles of Database Systems (PODS)*, pages 123–135, 2017. `doi:10.1145/3034786.3056120`.
5. Peter Buneman, Shamim Naqvi, Val Tannen, and Limsson Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995. `doi:10.1016/0304-3975(95)00024-Q`.
6. Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981. `doi:10.1145/322234.322243`.
7. Evgeny Dantsin and Andrei 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. `doi:10.1007/3-540-63045-7_7`.
8. Daniela Florescu and Ghislain Fourny. JSONiq: The history of a query language. *IEEE Internet Computing*, 17(5):86–90, 2013. `doi:10.1109/MIC.2013.97`.
9. Stéphane Grumbach and Victor Vianu. Tractable query languages for complex object databases. In *Proc. of the 10th ACM Symp. on Principles of Database Systems (PODS)*, pages 315–327, 1991. `doi:10.1145/113413.113442`.
10. Jan Hidders, Jan Paredaens, and Jan Van den Bussche. J-Logic: Logical foundations for JSON querying. In *Proc. of the 36th ACM Symp. on Principles of Database Systems (PODS)*, pages 137–149, 2017. `doi:10.1145/3034786.3056106`.
11. Gerhard Jaeschke and Hans-Jörg Schek. Remarks on the algebra of non first normal form relations. In *Proc. of the 1st ACM Symp. on Principles of Database Systems (PODS)*, pages 124–138, 1982. `doi:10.1145/588111.588133`.
12. David S. Johnson. A catalog of complexity classes. In *Handbook of Theoretical Computer Science*, volume A, chapter 2, pages 67–161. Elsevier Science Publishers, 1990.
13. Christoph Koch. On the complexity of nonrecursive XQuery and functional query languages on complex values. *ACM Trans. on Database Systems*, 31(4):1215–1256, 2006. `doi:10.1145/1189769.1189771`.
14. Kian Win Ong, Yannis Papakonstantinou, and Romain 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, arXiv.org e-Print archive, 2017. Available at `http://arxiv.org/abs/1405.3631`.
15. Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of JSON schema. In *Proc. of the 25th Int. World Wide Web Conf. (WWW)*, pages 263–273, 2016. `doi:10.1145/2872427.2883029`.
16. Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. *Journal on Data Semantics*, X:133–173, 2008. `doi:10.1007/978-3-540-77688-8_5`.
17. Stan J. Thomas and Patrick C. Fischer. Nested relational structures. *Advances in Computing Research*, 3:269–307, 1986.

18 Jan 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. doi:10.1016/S0304-3975(99)00301-1.

19 Jan Van den Bussche and Jan Paredaens. The expressive power of complex values in object-based data models. *Information and Computation*, 120(2):220–236, 1995. doi:10.1006/inco.1995.1110.

A Examples and Syntax of the MongoDB Aggregation Framework

The MongoDB *aggregation framework* provides a powerful querying mechanism, in which a query consists of a pipeline of *stages*, each transforming a forest into a new forest. We formalized a core part of this query language consisting of five stages as *MQuery*. In the examples below, we provide all queries both as MQueries and in the actual MongoDB syntax. We assume to have a second document in the `bios` collection as follows:

```
{ "_id": 6,
  "awards": [
    { "award": "Award for the Advancement of Free Software", "year": 2001, "by": "FSF" },
    { "award": "NLUUG Award", "year": 2003, "by": "NLUUG" } ],
  "birth": "1956-01-31",
  "contribs": [ "Python" ],
  "name": { "first": "Guido", "last": "van Rossum" } }
```

A.1 Match

The match operator takes as input a criterion, a Boolean condition on the trees, and returns the trees that satisfy that condition.

▶ **Example 21.** The following MQuery selects trees where the value of the path name.first is Kristen, and there exists an awards path:

$$\texttt{bios} \triangleright \mu_{\text{name.first}=\text{“Kristen”} \wedge \exists \text{awards}}$$

where the corresponding MongoDB query is:

```
db.bios.aggregate([
    {$match: {"name.first": {$eq: "Kristen"},
              "awards": {$exists: true} }} ])
```

This query returns the document about Kristen Nygaard. ◀

▶ **Example 22.** Consider the following query consisting of a match stage with two conditions on keys inside the `awards` array:

$$\texttt{bios} \triangleright \mu_{\text{awards.year}=1999 \wedge \text{awards.award}=\text{“Turing Award”}}$$

The corresponding MongoDB query is:

```
db.bios.aggregate([
    {$match: {"awards.year": {$eq: 1999},
              "awards.award": {$eq: "Turing Award"} }} ])
```

The query returns all persons that have received an award in 1999, and the Turing award in a possibly different year. Observe that it does not impose that one array element must satisfy all the conditions. This query retrieves the document about Kristen Nygaard because he received an award (the Rosing Prize) in 1999 in addition to the Turing Award (in 2001). ◀

A.2 Unwind

The unwind operator creates a new document for every element in an array.

▶ **Example 23.** The following MQuery unwinds path `awards`:

$$\texttt{bios} \triangleright \omega_{\text{awards}}$$

The corresponding MongoDB query is:

```
db.bios.aggregate([
    {$unwind: "$awards"} ])
```

When applied to the document about Guido van Rossum, it outputs 2 documents:

```
{ "_id": 6,
  "awards": { "award": "Award for the Advancement of Free Software", "year": 2001, "by": "FSF" },
  "birth": "1956-01-31",
  "contribs": [ "Python" ],
  "name": { "first": "Guido", "last": "van Rossum" } },
{ "_id": 6,
  "awards": { "award": "NLUUG Award", "year": 2003, "by": "NLUUG" },
  "birth": "1956-01-31",
  "contribs": [ "Python" ],
  "name": { "first": "Guido", "last": "van Rossum" } }
```

However, unwinding path `birth` in the same document gives the empty result, since the value of this path is not an array. ◀

A.3 Project

The project stage is similar to the extended projection from relational algebra.

▶ **Example 24.** The following query preserves the paths starting with `_id`, `name`, `awards.award` and `awards.year`:

$$\texttt{bios} \triangleright \rho_{\texttt{\_id}, \text{name}, \text{awards.award}, \text{awards.year}}$$

The corresponding MongoDB query is:

```
db.bios.aggregate([
   {$project: { "name": true, "awards.award": true, "awards.year": true}} ])
```

The document about Kristen Nygaard is then transformed into the document:

```
{ "_id": 4,
  "name": { "first": "Kristen", "last": "Nygaard" },
  "awards": [ { "award": "Rosing Prize", "year": 1999 },
              { "award": "Turing Award", "year": 2001 },
              { "award": "IEEE John von Neumann Medal", "year": 2001 } ] }
```

Observe that `_id` is preserved by MongoDB by default. In our formalization, though, the behaviour of project is the same for all paths. Note also that the information by whom the awards were given is lost as the path `awards.by` was not passed as a parameter. ◀

▶ **Example 25.** Project allows for renaming paths. The following query renames `name.first` to `firstName`, `awards.award` and `awards.year` to `awardsName` and `awardsYear`, respectively, and a non-existing path `abc` to `invisible`:

$$\texttt{bios} \triangleright \rho_{\texttt{\_id}, \text{firstName/name.first}, \text{awardsName/awards.award}, \text{awardsYear/awards.year}, \text{invisible/abc}}$$

The corresponding MongoDB query is:

```
db.bios.aggregate([
   {$project: {"firstName": "$name.first",
               "awardsName": "$awards.award", "awardsYear": "$awards.year",
               "invisible": "$abc" }} ])
```

It produces from the document about Kristen Nygaard:

```
{ "_id": 4,
  "firstName": "Kristen",
  "awardsName": [ "Rosing Prize", "Turing Award", "IEEE John von Neumann Medal" ],
  "awardsYear": [ 1999, 2001, 2001 ] }
```

Note that in the resulting document `awardsName` and `awardsYear` are two separate arrays unlike in the previous example, where keeping `awards.award` and `awards.year` without renaming them does not create two arrays. Also note that since there is no path `abc` in the input document, the result does not contain `invisible` key. ◀

▶ **Example 26.** Project also allows for creating new values, either fresh or from the existing ones. The following query introduces new keys `occupation` with value "Computer

Scientist", `fields` whose value is array consisting of the name, birth date and contributions, `sameFirstAndLastNames` whose value is the Boolean value of a comparison, and `condValue` whose value is calculated based on a condition:

$$\texttt{bios} \triangleright \rho_{\texttt{\_id},\,\text{occupation}/\text{“Computer Scientist”},\,\text{fields}/[\text{name},\,\text{birth},\,\text{contribs}],\,\text{sameFirstAndLastNames}/(\text{name.first}=\text{name.last}),\,\text{condValue}/((\texttt{\_id}=4)?\,\text{contribs}:\,\text{name})}$$

The corresponding MongoDB query is:

```
db.bios.aggregate([
    {$project: { "occupation": {$literal: "Computer Scientist"},
                 "fields": ["$name", "$birth", "$contribs"],
                 "sameFirstAndLastNames": {$eq: ["$name.first", "$name.last"]},
                 "condValue": {$cond: {
                     if: {$eq: ["$_id", 4]}, then: "$contribs", else: "$name" }} ])
```

It produces from the documents in the `bios` collection:

```
{ "_id": 4,                          "occupation": "Computer Scientist",
  "fields": [ {"first": "Kristen", "last": "Nygaard"}, "1926-08-27", [ "OOP", "Simula" ] ],
  "sameFirstAndLastNames": false,  "condValue": ["OOP", "Simula" ],
{ "_id": 6,                          "occupation": "Computer Scientist",
  "fields": [ {"first": "Guido", "last": "van Rossum"}, "1956-01-31", [ "Python" ] ],
  "sameFirstAndLastNames": false,  "condValue": {"first": "Guido", "last": "van Rossum"} }
```

Note that this project stage is a non-well-typed one. First, the array `fields` is not a well-typed array. Second, the types of `condValue` in the two resulting trees do not coincide. This demonstrates that project is a very powerful stage and can produce from a well-typed input forest a non-well-typed one. ◀

A.4 Group

The group stage allows to combine different trees into one. More specifically, the set of input trees is partitioned into several according to a grouping condition, and for each group a single output tree is produced. The documents are grouped together according to the grouping condition, and only the values of paths specified in the aggregation expression are included in the output, combined into an array for each group. Notice that, in the MongoDB syntax, the grouping condition G is specified through the key-value pair `_id` : G.

Let us consider an additional collection, called `awards`, focusing on the award information:

```
{ "_id": 1, "person_id": 4, "name": "Rosing Prize", "in": 1999 },
{ "_id": 2, "person_id": 4, "name": "Turing Award", "in": 2001 },
{ "_id": 3, "person_id": 4, "name": "IEEE John von Neumann Medal", "in": 2001 },
{ "_id": 4, "person_id": 6, "name": "Award for the Advancement of Free Software",
                                                                  "in": 2001 },
{ "_id": 5, "person_id": 6, "name": "NLUUG Award", "in": 2003 }
```

▶ **Example 27.** The following query returns for each year the identifiers of scientists that received an award in that year:

$$\texttt{awards} \triangleright \gamma_{\text{year}/\text{in}\,:\,\text{scientists}/\text{person\_id}}$$

The corresponding MongoDB query is:

```
db.awards.aggregate([
    {$group: { "_id": {"year": "$in"}, "scientists": {$addToSet: "$person_id"} }} ])
```

Running this query over the `awards` collection produces the following output:

```
{ "_id": { "year": 2001 }, "scientists": [4, 6] },
{ "_id": { "year": 1999 }, "scientists": [4] },
{ "_id": { "year": 2003 }, "scientists": [6] }
```

▶ **Example 28.** The following group stage has no aggregation condition, so all input documents are aggregated into one. It returns the names of all the scientists in the `bios` collection:

$$\texttt{bios} \triangleright \gamma_{:\,\text{names}/\text{name}}$$

The corresponding MongoDB query is:

```
db.bios.aggregate([
    {$group: { "_id": null, "names": {$addToSet: "$name"} }} ])
```

Running this query over the `bios` collection produces the following output:

```
{ "_id": null,
  "names": [ { "first": "Kristen", "last": "Nygaard" },
             { "first": "Guido", "last": "van Rossum" } ] }
```

▶ **Example 29.** The following query groups persons according to their date of death:

$$\texttt{bios} \triangleright \gamma_{\text{death}\,:\,\text{names/name}}$$

The corresponding MongoDB query is:

```
db.bios.aggregate([
    {$group: { "_id": "$death", "names": {$addToSet: "$name"} }} ])
```

When executing over the `bios` collection, it produces the following output:

```
{ "_id": "2002-08-10",
  "names" : [ { "first": "Kristen", "last": "Nygaard" } ] },
{ "_id": null,
  "names": [ { "first": "Guido", "last": "van Rossum" } ] }
```

Since the `death` path is not present in the document about Guido van Rossum, the latter is grouped in the document where `_id` is **null**. ◀

A.5 Lookup

The lookup stage joins input documents with documents in an external collection, using a local path and a path in the external collection to express the join condition, and stores the matching external documents in an array.

▶ **Example 30.** For each document in the `bios` collection, the following query collects information about the awards received by the scientist from the `awards` collection and stores it in the `awards_info` array:

$$\texttt{bios} \triangleright \lambda^{\text{\_id=awards.person\_id}}_{\text{awards\_info}}$$

The corresponding MongoDB query is:

```
db.bios.aggregate([
 {$lookup: {
     from: "awards", localField: "_id", foreignField: "person_id", as: "awards_info" }} ])
```

Executing this query over the `bios` collection produces the following result:

```
{ "_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" },
  "awards_info": [
    { "_id": 1, "person_id": 4, "name": "Rosing Prize", "in": 1999 },
    { "_id": 2, "person_id": 4, "name": "Turing Award", "in": 2001 },
    { "_id": 3, "person_id": 4, "name": "IEEE John von Neumann Medal", "in": 2001 } ] },
{ "_id": 6,
  "awards": [
    { "award": "Award for the Advancement of Free Software", "year": 2001, "by": "FSF" },
    { "award": "NLUUG Award", "year": 2003, "by": "NLUUG" } ],
  "birth": "1956-01-31",
  "contribs": [ "Python" ],
  "name": { "first": "Guido", "last": "van Rossum" },
  "awards_info": [
    { "_id": 4, "person_id": 6, "name": "Award for the Advancement of Free Software",
                                                          "in": 2001 },
    { "_id": 5, "person_id": 6, "name": "NLUUG Award", "in": 2003 } ] }
```

B Details on the Semantics of tree operations in MQuery

In the following, let $t = (N, E, L_\mathsf{n}, L_\mathsf{e})$ be a tree. Below, when we mention reachability, we mean reachability along the edge relation.

subtree: *The subtree of t rooted at x and induced by M*, for $x \in M$ and $M \subseteq N$, denoted $\mathsf{subtree}(t, x, M)$, is defined as $(N', E|_{N'\times N'}, L_\mathsf{n}|_{N'}, L_\mathsf{e}|_{E'})$ where N' is the subset of nodes in M reachable from x by traversing only nodes in M. Note that $\mathsf{subtree}(t, x, M)$ might be the empty tree, e.g., when M is a set of nodes disconnected from x. We write $\mathsf{subtree}(t, M)$ as abbreviation for $\mathsf{subtree}(t, \mathsf{root}(t), M)$.

For a path p with $|[\![p]\!]^t| = 1$, *the subtree* $\mathsf{subtree}(t, p)$ *of t hanging from p* is defined as $\mathsf{subtree}(t, r_p, N')$ where $\{r_p\} = [\![p]\!]^t$, and N' are the nodes reachable from r_p via E. For a path p with $|[\![p]\!]^t| = 0$, $\mathsf{subtree}(t, p)$ is defined as $\mathsf{tree}(\mathbf{null})$.

attach: The *tree* $\mathsf{attach}(k_1 \dots k_n, t)$ *constructed by inserting the path $k_1 \dots k_n$ on top of the tree t*, for $n \geq 1$, is defined as $(N', E', L'_\mathsf{n}, L'_\mathsf{e})$, where *(i)* $N' = N \cup \{x_0, x_1, \dots, x_{n-1}\}$, for fresh $x_0, \dots, x_{n-1}$, *(ii)* $E' = E \cup \{(x_0, x_1), (x_1, x_2), \dots, (x_{n-1}, \mathsf{root}(t))\}$, *(iii)* $L'_\mathsf{n} = L_\mathsf{n} \cup \{(x_0, \text{`}\{\}\text{'}), \dots, (x_{n-1}, \text{`}\{\}\text{'})\}$, and *(iv)* $L'_\mathsf{e} = L_\mathsf{e} \cup \{((x_0, x_1), k_1), \dots, ((x_{n-2}, x_{n-1}), k_{n-1}), ((x_{n-1}, \mathsf{root}(t)), k_n)\}$.

intersection: Let $t_j = (N^j, E^j, L^j_\mathsf{n}, L^j_\mathsf{e})$, $j = 1, 2$, be trees. The function $t_1 \cap t_2$ returns the set of pairs of nodes $(x_n, y_n) \in N^1 \times N^2$ reachable along identical paths in t_1 and t_2, that is, such that there exist $(x_0, x_1), \dots, (x_{n-1}, x_n)$ in E^1, for $x_0 = \mathsf{root}(t_1)$, and $(y_0, y_1), \dots, (y_{n-1}, y_n)$ in E^2, for $y_0 = \mathsf{root}(t_2)$, with $L^1_\mathsf{n}(x_i) = L^2_\mathsf{n}(y_i)$ and $L^1_\mathsf{e}(x_{i-1}, x_i) = L^2_\mathsf{e}(y_{i-1}, y_i)$, for $1 \leq i \leq n$.

merge: Let $t_j = (N^j, E^j, L^j_\mathsf{n}, L^j_\mathsf{e})$, $j = 1, 2$, be trees such that $N^1 \cap N^2 = \emptyset$, and for each path p leading to a leaf in t_2, i.e., $t_2 \models (p = v)$ for some literal value v, we have that $t_1 \not\models \exists p$ and the other way around. Then the *tree $t_1 \oplus t_2$ resulting from merging t_1 and t_2* is defined as $(N, E, L_\mathsf{n}, L_\mathsf{e})$, where *(i)* $N = N^1 \cup N^{2'}$, for $N^{2'} = N^2 \setminus \{x_2 \mid (x_1, x_2) \in t_1 \cap t_2\}$, *(ii)* $E = E^1 \cup (E^2 \cap (N^{2'} \times N^{2'})) \cup ((t_1 \cap t_2) \circ E^2)$, *(iii)* $L_\mathsf{n} = L^1_\mathsf{n} \cup L^2_\mathsf{n}|_{N^{2'}}$, and *(iv)* $L_\mathsf{e} = L^1_\mathsf{e} \cup L^2_\mathsf{e}|_{N^{2'}\times N^{2'}} \cup \{((x_1, y_2), \ell) \mid L^2_\mathsf{e}(y_1, y_2) = \ell, (x_1, y_1) \in t_1 \cap t_2\}$

replace: Let $t = (N, E, L_\mathsf{n}, L_\mathsf{e})$ and $t_j = (N^j, E^j, L^j_\mathsf{n}, L^j_\mathsf{e})$, $j = 1, 2$, be trees such that t_1 is a subtree of t with $\mathsf{root}(t_1) \neq \mathsf{root}(t)$ and N^2 is disjoint from N. Further, let x be the parent of $\mathsf{root}(t_1)$ in t, i.e., $(x, \mathsf{root}(t_1)) \in E$, with $L_\mathsf{n}(x, \mathsf{root}(t_1)) = \ell$. Then the tree $\mathsf{replace}(t, t_1, t_2)$ resulting from replacing t_1 by t_2 in t is defined as $(N', E', L'_\mathsf{n}, L'_\mathsf{e})$, where *(i)* $N' = N \setminus N^1 \cup N^2$, *(ii)* $E' = E \cap (N' \times N') \cup E^2 \cup \{(x, \mathsf{root}(t_2))\}$, *(iii)* $L'_\mathsf{n} = L_\mathsf{n} \setminus L^1_\mathsf{n} \cup L^2_\mathsf{n}$, and *(iv)* $L'_\mathsf{e} = L_\mathsf{e}|_{E'} \cup L^2_\mathsf{e} \cup \{((x, \mathsf{root}(t_2)), \ell)\}$.

array: Let $\{t_1, \dots, t_n\}$, $n \geq 0$, be a forest and p a path. The operator $\mathsf{array}(\{t_1, \dots, t_n\}, p)$ creates the tree encoding the array of the values of the path p in the trees $t_1, \dots, t_n$. Let $t^p_j = \mathsf{subtree}(t_j, p)$ with $(N^j, E^j, L^j_\mathsf{n}, L^j_\mathsf{e})$ where all N^j are mutually disjoint, $t_{j_1} \neq t_{j_2}$ for $j_1 \neq j_2$, and $r_j = \mathsf{root}(t^p_j)$, where $1 \leq j \leq m \leq n$ (without loss of generality, we may assume that $t_1, \dots, t_n$ are ordered accordingly). Then, $\mathsf{array}(\{t_1, \dots, t_n\}, p)$ is the tree $(N, E, L_\mathsf{n}, L_\mathsf{e})$ where *(i)* $N = \left(\bigcup_{j=1}^n N^j\right) \cup \{v_0\}$, *(ii)* $E = \left(\bigcup_{j=1}^n E^j\right) \cup \{(v_0, r_1), \dots, (v_0, r_n)\}$, *(iii)* $L_\mathsf{n} = \left(\bigcup_{j=1}^n L^j_\mathsf{n}\right) \cup \{(v_0, \text{`}[]\text{'})\}$, and *(iv)* $L_\mathsf{e} = \left(\bigcup_{j=1}^n L^j_\mathsf{e}\right) \cup \{((v_0, r_1), 0), \dots, ((v_0, r_n), n-1)\}$.

We also define $\mathsf{subtree}(t, p)$ for paths p such that $|[\![p]\!]^t| > 1$. In this case it returns the tree encoding the array of all subtrees hanging from p. Formally, $\mathsf{subtree}(t, p) = \mathsf{array}(\{t_1, \dots, t_n\}, \varepsilon)$, where $\{r_1, \dots, r_n\} = [\![p]\!]^t$, N_j the set of nodes reachable from r_j via E, and $t_j = \mathsf{subtree}(t, r_j, N_j)$. We observe that the definition of the array operator is recursive as it uses the generalized $\mathsf{subtree}$ operator.