A Formal Presentation of MongoDB (Extended Version)

Elena Botoeva[1], Diego Calvanese[1], Benjamin Cogrel[1], Martin Rezk[2], and Guohui Xiao[1]

1 **Free University of Bozen-Bolzano** `lastname@inf.unibz.it`

2 **Rakuten, Inc.** `martin.rezk@rakuten.com`

Abstract

A significant number of database architectures and data models have been proposed during the last decade. While some of these new systems have gained in popularity, their formal semantics are generally still missing. In this paper, we consider the symptomatic case of MongoDB, a widely adopted document database, in which roughly speaking relational tables correspond to collections, and tuples to documents. We provide a formalization of the JSON-based data model adopted by MongoDB, and of a core fragment of the MongoDB aggregation query language, MUPGL, which includes the match, unwind, project, group, and lookup operators. We study the expressiveness of MUPGL by defining a relational view of MongoDB databases and developing a translation from relational algebra to MUPGL. Notably, we show that the MUPG fragment is already at least as expressive as full relational algebra over (the relational view of) a single collection, and in particular able to express arbitrary joins. We further investigate the computational complexity of MUPGL and of significant fragments of it.

1 Introduction

As envisioned by Stonebraker and Cetintemel [9], during the last ten years a diversity of new database (DB) architectures and data models has emerged, driven by the goal of better addressing the widely varying characteristics of modern data-intensive applications. Notably, many of these new systems do not rely on the relational model, hence the emergence of the term *NoSQL* (not only SQL) [5, 6]. While some of these *non-relational* DBs have gained in popularity and have been deployed in large-scale applications, there have been only some attempts at formally capturing their query languages, e.g., through a calculus [2], and in general a thorough understanding of their formal and computational properties is still missing.

In this paper, we consider the symptomatic case of MongoDB[1], a widely adopted document database with rich querying capabilities that is still lacking a proper formalization. MongoDB organizes data in collections of semi-structured tree-shaped documents in the BSON format, a variant of the JavaScript Object Notation (JSON) commonly used as an alternative to XML. A key characteristic of the tree structure of documents is the high degree of locality it offers, which is not provided by standard relational data in first normal form. As an example, consider the document in Figure 1, containing not only standard personal information about Kristen Nygaard, which could be common to (most) other persons (such as name and birthdate), but also describes very specific information (such as the awards he received). MongoDB provides rich querying capabilities by means of the *aggregation framework*, which is modelled on the notion of data processing pipelines. In this framework, each query is a

[1] `https://docs.mongodb.org/manual/`

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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

multi-stage pipeline, where each stage defines a transformation on a set of documents using a MongoDB-specific operator. We call such queries *MongoDB aggregate queries (*MAQ*s)*.

▶ **Example 1.** Consider a collection `bios` of documents as the one in Figure 1, each storing information about prominent computer scientists, such as their names and received awards. Then we can retrieve all persons who received two awards in the same year with the following pipeline:

```
db.bios.aggregate([
  {$project : {
    "name": true, "award1": "$awards", "award2": "$awards" } },
  {$unwind: "$award1"},
  {$unwind: "$award2"},
  {$project: {
    "name": true, "award1": true, "award2": true,
    "twoInOneYear": { $and: [
      {$eq: ["$award1.year", "$award2.year"]},
      {$ne: ["$award1.award", "$award2.award"]} ]} } },
  {$match: {"twoInOneYear": true} },
  {$project : {
    "firstName": "$name.first",    "lastName": "$name.last" ,
    "awardName1": "$award1.award", "awardName2": "$award2.award",
    "year": "$award1.year" } },
])
```

This MAQ is a sequence of six stages using the *match*, *unwind*, and *project* operators. For each document in `bios`, it performs a join over the awards contained in the document. ◁

Understanding and learning the MAQ language requires a significant effort because of its unconventional nature and expressiveness, while its specificity to MongoDB makes such an investment hard to justify. To circumvent this difficulty, some attempts (SQL^{++} [8], Apache Drill[2], and Teiid[3]) have been made to let end-users query MongoDB using a possibly extended version of SQL instead of MAQ. Notably, to the best of our knowledge, these systems only use a very limited subset of the MongoDB query capabilities, and therefore have to set up compensating post-processing techniques in which arbitrary programming code might be used, thus completely losing the declarative nature of a query language. These observations motivate us to study the expressiveness of MAQ in terms of well-known query constructs, such as join. Understanding what can be expressed by MAQ, on the one hand, helps MongoDB end-users in formulating MAQs, and on the other hand, guides the developers of the mentioned systems to determine to which extent query processing could be delegated to MongoDB.

Specifically, we conduct the first major investigation into the formal foundations and properties of MongoDB. Our main contribution is a formalization of the MongoDB data model and of the MUPGL fragment of MAQ, which includes the match, unwind, project, group,

[2] `https://drill.apache.org/`
[3] `http://teiid.jboss.org/`

and lookup operators. In particular, we study what can be expressed by MUPGL, and discover a strong connection between this fragment and relational algebra (RA). To this end, we define a relational view of MongoDB databases, and, inspired by [10], which provide a translation of first order queries into XPath 2.0, we show an encoding of relational algebra (RA) into MUPGL. Specifically, we show that the MUPG fragment is already as expressive as full RA over (the relational view of) a single collection, and in particular able to express arbitrary joins. We further discuss some notable features of MongoDB that we have encountered in our investigation as a result of our attempts to understand the semantics of its query language. Since such features are to some degree counterintuitive, and show even some inconsistent behaviors of the current version of MongoDB (v3.2), we consider it important to make the MongoDB community aware of them, so that users can properly make use of the query language.

Further, we carry out a preliminary investigation of the computational complexity of MUPGL and of significant fragments of it. In particular, we establish a number of lower bounds, and identify some tractable fragments.

2 Preliminaries

We recap the basics of relational algebra, mainly to fix notation. We consider the named perspective, in which a relation is characterized by its *signature* S, which is a name with an associated finite set $\mathsf{att}(S)$ of attributes. The number of elements of $\mathsf{att}(S)$ is the *arity* of S.

A *tuple* t over a signature S with $\mathsf{att}(S) = \{a_1, \dots, a_n\}$, also called an S*-tuple*, is a set $\{a_1{:}v_1, \dots, a_n{:}v_n\}$ consisting of one attribute-value pair $a_i{:}v_i$ for each attribute $a_i \in \mathsf{att}(S)$, where each value v_i is an element of an underlying domain Δ. A *relation over* S is a set of S-tuples. A *relational schema* RS is a finite set of signatures, and a *(relational database) instance* of RS is a set of relations, one over each $S \in RS$. A *filter* ψ over a set A of attributes is a Boolean formula constructed from atoms $(a \ \mathbf{op}\ v)$ and $(a\ \mathbf{op}\ a')$, where $a, a' \in A$, $v \in \Delta$, and $\mathbf{op}$ is one of $=$, $\neq$, $<$, $\leq$, $>$, and $\geq$.

Let S and S' be relational signatures. We recall the following relational algebra operators: *(i) set union* $S \cup S'$ and *set difference* $S \setminus S'$, for $\mathsf{att}(S) = \mathsf{att}(S')$; *(ii) cross-product* $S \times S'$; *(iii) join* $S \bowtie_\psi S'$, where ψ is a filter over $\mathsf{att}(S) \cup \mathsf{att}(S')$; if ψ is missing, then it denotes *natural join*; *(iv) left outer join* $S \mathbin{⟕}_\psi S'$; *(v) selection* $\sigma_\psi(S)$, where ψ is a filter over $\mathsf{att}(S)$; *(vi) projection* $\pi_A(S)$, for $A \subseteq \mathsf{att}(S)$; *(vii) extended projection* $\pi_A(S)$, where A may also contain elements of the form $b/f(a_1, \dots, a_n)$, for a computable function f, $a_i \in \mathsf{att}(S)$, and b a new attribute name not in $\mathsf{att}(S)$; *(viii) renaming* $\rho_{b/a}(S)$, where $a \in \mathsf{att}(S)$ and $b \notin \mathsf{att}(S)$.

3 Syntax and Semantics of MongoDB Documents

In this section we propose a formalization of the syntax and the semantics of MongoDB documents. As mentioned in the introduction, a MongoDB database stores collections of JSON-style documents, where each document is an object consisting of key-value pairs, where a value can itself be a nested object. Roughly, a collection corresponds to a table in a relational database, and a document corresponds to a tuple (a row in a table). Notice that, in MongoDB, the term 'key' is used with the meaning of 'attribute' in relational databases, hence it should not be confused with the traditional notion in 'key constraints'. Here we adopt the same terminology, thus, a key-value pair can be seen as an attribute-value pair.

We start by defining the syntax of MongoDB documents in the BSON (for Binary JSON) format. Let *literals* be atomic values, such as strings, numbers, and booleans. A *BSON*

```
 Value  ::= Literal | Object | Array
Object  ::= ⦃ List<Key : Value> ⦄
 Array  ::= [ List<Value> ]

where

  List<T>  ::= ε | List+<T>
 List+<T>  ::= T | T , List+<T>
```

Figure 2 Syntax of BSON objects.

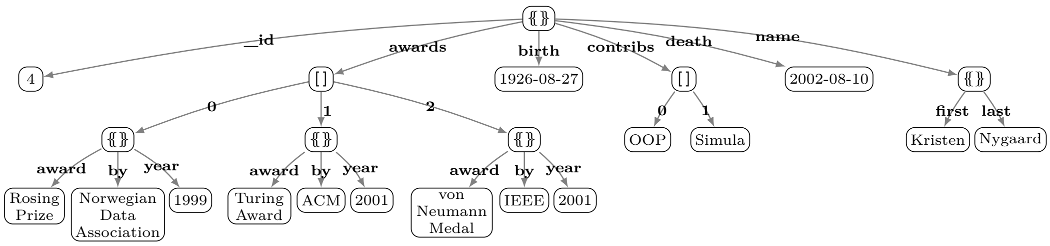

Figure 3 The tree representation of the MongoDB document in Figure 1.

object o is a finite ordered 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 according to the grammar in Figure 2. We require that the set of key-value pairs constituting a BSON object does not contain the same key twice. A *(MongoDB) document* is a BSON object (not nested within any other object) with a special key '`_id`', which is used to identify the document. Figure 1 shows a MongoDB document in which, apart from `_id`, the keys are `birth`, `name`, `awards`, etc. For that document, the value of `_id` is 4, the value of `birth` is "1926-08-27", the value of `name` is an object consisting of two key-value pairs, and the value of `awards` is an array of objects, each describing an award. Given a collection name C, a *(MongoDB) collection for* C is a finite set F_C of documents, such that each document is identified by the value of `_id`, i.e., the 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.

In the following, we formalize MongoDB documents as finite *ordered unranked node and edge-labeled trees* satisfying specific conditions. We assume three disjoint sets of labels: the sets K of *keys* and I of *indexes*, which are used as edge labels, and the set V of *literals*, which are used as node labels. The indexes are non-negative integers, and V contains the special elements **null**, **true**, and **false**.

A *(valid) tree* is a tuple $(N, E, \prec, L_n, L_e)$, where N is a set of nodes, E is a successor relation, $\prec$ is a partial order on N that imposes a total order on siblings, $L_n : N \to V \cup \{`⦃⦄\text{'}, `[\,]\text{'}\}$ is a node labeling function, and $L_e : E \to K \cup I$ is an edge labeling functions, 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 consecutive indexes starting from 0, and respecting the sibling order $\prec$. The fact that $\prec$ is a total order on siblings formally means that, for every node $x \in N$, if N_x is the set of children of x, then there is an enumeration $\{x_1, \dots, x_m\}$ of the nodes in N_x such that $x_1 \prec \cdots \prec x_m$. Given a tree t and a node x, the *type* of x in t, denoted $\mathsf{type}(x, t)$, is $\mathsf{literal}$ if $L_n(x) \in V$, object if $L_n(x) =$ '⦃⦄', and array if $L_n(x) =$ '[ ]'. The root of t is denoted by $\mathsf{root}(t)$. A *forest* is a set of trees. If $\mathsf{root}(t)$ has an outgoing edge labeled with `_id`, we call the tree t a *document*.

Given a tree t, we define inductively for each node x in t, the *value represented by* x *in* t, denoted $\mathsf{value}(x, t)$, as follows: *(i)* if x is a leaf in t, then $\mathsf{value}(x, t) = L_n(x)$; *(ii)* let $x_1, \dots, x_m$, with $x_1 \prec \cdots \prec x_m$, be all the children of x with the corresponding edges labeled

```
MFQ ::= Collection.find({Criterion}, { List<Path: true> })    MAQ ::= Collection.aggregate([ List+<Stage> ])
Stage ::= {$match: {Criterion}} | {$unwind: {UnwindExpr}} | {$project: {Projection}}
        | {$group: {GroupExpr}} | {$lookup: {LookupExpr}}
-----------------------------------------------------------------------------------------
         Path ::= Key | Path.Key
      PathRef ::= $Path | $$ROOT
          Lop ::= $and | $or | $nor
          Cop ::= $eq | $ne
                | $gt | $gte | $lt | $lte
          Aop ::= $in | $nin
          Bop ::= Lop | Cop
      Boolean ::= true | false
    Criterion ::= Path: Condition
                | Lop: [ List+<{Criterion}> ]
    Condition ::= {Cop: Value}
                | {Aop: [ List<Value> ]}
                | {$not: Condition}
                | {$exists: Boolean}
   Projection ::= List+<ProjectionElem>
ProjectionElem ::= _id: false | Path: true
                | Path: ValueDef

       ValueDef ::= PathRef
                  | {$literal: Value} | [ List<ValueDef> ]
                  | {Bop: [ List<ValueDef> ]}
                  | {$not: ValueDef}
                  | {$cond: { if: ValueDef,
                              then: ValueDef,
                              else: ValueDef }}
      GroupExpr ::= _id: GroupCondition,
                    List<Path: { $addToSet: PathRef }>
 GroupCondition ::= null | _id: { List<Path: PathRef> }
     UnwindExpr ::= path: PathRef,
                    includeArrayIndex: Path,
                    preserveNullAndEmptyArrays: Boolean
     LookupExpr ::= from: Collection,
                    localField: Path,
                    foreignField: Path,
                    as: Path
```

Figure 4 The grammar of MongoDB find and aggregate queries.

by $k_1, \ldots, k_m$. If $\mathsf{type}(x,t) = \mathsf{object}$, then $\mathsf{value}(x,t) = \{\!\{k_1{:}\mathsf{value}(x_1,t), \ldots, k_m{:}\mathsf{value}(x_m,t)\}\!\}$, and if $\mathsf{type}(x,t) = \mathsf{array}$, then $\mathsf{value}(x,t) = [\mathsf{value}(x_1,t), \ldots, \mathsf{value}(x_m,t)]$. The *BSON document represented by* t, denoted $\mathsf{value}(t)$, is then $\mathsf{value}(\mathsf{root}(t),t)$.

The tree corresponding to a value u, denoted $\mathsf{tree}(u)$, is defined as $(N, E, \prec, L_n, L_e)$, where $N = \{x_v \mid v$ is an object, array, or literal value appearing in $u\}$, and for $x_v \in N$: *(i)* if v is a literal, then $L_n(x_v) = v$ and x_v is a leaf; *(ii)* if $v = \{\!\{k_1 : v_1, \ldots, k_m : v_m\}\!\}$, for $m \geq 0$, then $L_n(x_v) =$ '$\{\!\{\}\!\}$', x_v has m children $x_{v_1}, \ldots, x_{v_m}$ with $L_e(x_v, x_{v_i}) = k_i$ and $x_{v_1} \prec \cdots \prec x_{v_m}$; *(iii)* if $v = [v_1, \ldots, v_m]$, for $m \geq 0$, then $L_n(x_v) =$ '$[\,]$', x_v has m children $x_{v_1}, \ldots, x_{v_m}$ with $L_e(x_v, x_{v_i}) = i - 1$ and $x_{v_1} \prec \cdots \prec x_{v_m}$. Observe that a literal v can be seen as a tree consisting of a single node whose label is v. Then, the tree corresponding to a BSON document d is defined as $\mathsf{tree}(d)$, where d is viewed as a value. The tree representation of the document in Figure 1 is depicted in Figure 3.

4 Syntax and Semantics of MongoDB Queries

MongoDB provides two main query mechanisms. The basic form of query is a *find* query, which allows one to filter out documents according to some (Boolean) criteria and to return, for each document passing the filter, a tree containing a subset of the key-value pairs in the document. With a find query we cannot change the shape of the individual pairs. A more powerful querying mechanism is provided by the *aggregation framework*, in which a query consists of a pipeline of *stages*, each transforming a forest into a new forest, and with the possibility of manipulating the shape of the trees. We call this transformation pipeline an *aggregate query*. Some examples of queries can be found in Section A.1.

4.1 Syntax of MongoDB Aggregate Queries

We consider fragments of MongoDB find and aggregate queries as of the latest version (v3.2). The grammar for the considered fragments of *MongoDB aggregate queries* (MAQ) and *MongoDB find queries* (MFQ), which are a special case of MAQ, is presented in Figure 4 (for readability, we use single curly brackets in queries). An MFQ consists of a criteria part, selecting documents of interest, and a projection part, specifying which paths should be kept

$$
\begin{array}{rcl}
\mathbf{op} & ::= & = \mid \neq \mid < \mid \leq \mid > \mid \geq \\
\varphi & ::= & p\ \mathbf{op}\ v \mid \exists p \mid \varphi \lor \varphi \mid \varphi \land \varphi \mid \neg \varphi \\
d & ::= & p \mid v \mid [d, \ldots, d] \mid \beta \mid \kappa \\
\beta & ::= & \mathbf{op}(p, p) \mid \mathbf{op}(p, v) \mid d \lor d \mid d \land d \mid \neg d \\
\kappa & ::= & d/d/d \\
\hline
S & ::= & \mu_\varphi \mid \omega_p^{(q,n)} \mid \theta^{p_1,\ldots,p_n}_{q_1=d_1,\ldots,q_m=d_m} \mid \gamma^{g_1:g'_1,\ldots,g_n:g'_n}_{a_1:a'_1,\ldots,a_m:a'_m} \mid \lambda_p^{p_1=C.p_2} \\
\textsc{mupgl} & ::= & S \triangleright \cdots \triangleright S \\
\textsc{mfq} & ::= & \mu_\varphi \triangleright \theta^{p_1,\ldots,p_n}
\end{array}
$$

Figure 5 Algebra for MUPGL queries.

in the output documents and which not. An MAQ instead, is a sequence of stages, each of which transforms a forest into another forest. We consider a fragment of MAQ, which we call MUPGL (for match, unwind, project, group, lookup), that allows for five types of stages: *(i) match*, which selects trees of interest, *(ii) unwind*, which flattens an array from the input trees to output a tree for each element of the array, *(iii) project*, which modifies trees by projecting away paths, renaming paths, or introducing new paths, *(iv) group*, which groups trees according to the values of a set of paths, and *(v) lookup*, which joins trees in the local collection with trees in an external collection C, using a local path and a path in C to express the join condition, and an additional path to store the matching trees. We consider also various fragments of MAQ, and we denote each fragment by including in the name the first letter of those stages that can be used in queries. For example, MUPG denotes the fragment of MUPGL that does not use lookup, and MUP the fragment of MUPG that does not use group. Since a query in MFQ is a special case of query in MP, in the following, we consider only MAQ.

We provide some comments and additional requirements on the grammar in Figure 4. A `Path` (which in MongoDB terminology is actually called a "field"), is a non-empty concatenation of `Key`s, where elements for `Key` are from the set K. Elements for `Value` are defined according to the grammar in Figure 2. `Collection` is a collection name, that is, a non-empty string. We use ε to denote the empty string and the empty path. The empty path can be used in a path reference and is implemented in MongoDB by the string `$$ROOT`. In the following, a *path* is either the empty path or an element constructed according to `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''. Also, p' is a *prefix* of p if p' is either a strict prefix of p or equal to p. We assume that a projection $p_1{:}d_1, \ldots, p_n{:}d_n$ is such that there are no $i \neq j$ where p_i is a prefix of p_j. The comparison operators used in a value definition `ValueDef` accept only arrays of length 2. With respect to the official MongoDB syntax, we have removed/introduced some syntactic sugar. In particular, for `Criterion` we disallow expressions of the form `"name.first": "john"`. Instead they can be expressed as `"name.first": {$eq: "john"}`. Moreover, we allow the use of `$nor` in `ValueDef`, as it can be expressed using `$not` and `$and`.

4.2 Semantics of MongoDB Aggregate Queries

To abstract away syntactic aspects of MongoDB queries, and allow us to formalize their semantics, we first propose an algebra for them. It is shown in Figure 5, where **op** stands for a comparison operator, φ for a criterion, p, p' for paths, v for a value, d for a value definition, β for a Boolean value definition, κ for a conditional value definition, C for a collection name, and S for a stage. Moreover, we denote the query stages as follows: *(i)* μ_φ for a match stage; *(ii)* $\omega_p^{(q,n)}$ for an unwind stage, where the optional q corresponds to the path where to store the array index, and the presence of the optional n indicates to preserve null and empty arrays; *(iii)* $\theta^{p_1,\ldots,p_n}_{q_1=d_1,\ldots,q_m=d_m}$ for a project stage, where $p_1,\ldots,p_n$ are paths to be kept, and $q_1,\ldots,q_m$ are new paths with value definitions $d_1,\ldots,d_m$, respectively; *(iv)* $\gamma^{g_1:g'_1,\ldots,g_n:g'_n}_{a_1:a'_1,\ldots,a_m:a'_m}$ for a group stage, where $g_1 : g'_1,\ldots,g_n : g'_n$ provide the group condition, and $a_1 : a'_1,\ldots,a_m : a'_m$ are the aggregation paths; and *(v)* $\lambda^{p_1=C.p_2}_{p}$ for a lookup stage, where C is the name of the external collection, p_1 is the local path, p_2 is the path from collection C, and p is the path to store the matching trees.

To introduce the formal set semantics of the MongoDB algebra, we specify the semantics of each stage over a forest, and then obtain the semantics of a query by simply composing (via $\triangleright$) the answers of its stages. First, we show how to interpret paths over trees.

▶ **Definition 2.** Given a tree $t = (N, E, \prec, L_n, L_e)$, for a node $x \in N$, we define $\mathsf{ipath}(x,t)$ as the concatenation of the edge labels on the path from $\mathsf{root}(t)$ to x, and $\mathsf{path}(x,t)$ as the result of eliminating all indexes from $\mathsf{ipath}(x,t)$. Then, we interpret a (possibly empty) path p, and its concatenation $p.i$ with an index i as sets of nodes as follows, where k is a key:

$$\begin{array}{rcl} [\![\varepsilon]\!]^t & = & \{\mathsf{root}(t)\} \\ [\![p.k]\!]^t & = & \{x \in N \mid \mathsf{path}(x,t) = p.k \text{ and the incoming edge of } x \text{ is labeled by } k\} \\ [\![p.i]\!]^t & = & \{y \in N \mid \text{there exists } x \text{ s.t. } \mathsf{path}(x,t) = p,\ (x,y) \in E \text{ and } L_e(x,y) = i\} \end{array}$$

When $[\![p]\!]^t = \emptyset$, we say that the path p is *missing* in t. ◁

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

We are ready to define the semantics of the match stage that filters out the trees that do not satisfy the criterion. In this definition we assume that for each comparison operator **op** and pair of values v_1 and v_2, the comparison (v_1 **op** v_2) evaluates to a boolean value. We say (v_1 **op** v_2) *holds* when it evaluates to true. We observe that when v_2 is **null**, then (v_1 **op** v_2) holds iff **op** is one of "=", "≤", "≥" and v_1 is **null**, or **op** is "≠" and v_1 is not **null** (similarly when v_1 is **null**). The comparison of non-atomic values is defined by the BSON specification, which roughly follows the lexical order of the binary representation of values[4].

▶ **Definition 3** (Match μ)**.** Given a criterion φ and a tree $t = (N, E, \prec, L_n, L_e)$, we define when t *satisfies* φ, denoted $t \models \varphi$, by treating Booleans as usual, and defining:

$t \models (p\ \mathbf{op}\ v)$ for $v \neq \mathbf{null}$, if there is $x \in [\![p]\!]^t$ or $x \in [\![p.i]\!]^t$ for some $i \in I$, such that ($\mathsf{value}(x,t)$ $\mathbf{op}$ v) holds.

$t \models (p\ \mathbf{op}\ \mathbf{null})$, if *(i)* **op** is one of "=", "≤", "≥", and $[\![p]\!]^t = \emptyset$, or *(ii)* there is $x \in [\![p]\!]^t$ or $x \in [\![p.i]\!]^t$ for some $i \in I$, such that ($\mathsf{value}(x,t)$ $\mathbf{op}$ $\mathbf{null}$) holds.

$t \models (\exists p)$, if $[\![p]\!]^t \neq \emptyset$.

Let F be a forest. Then $F \triangleright \mu_\varphi = \{t \in F \mid t \models \varphi\}$. ◁

We also define when a value definition d evaluates to true in a tree t, denoted $t \models d$. It is used for evaluation of conditional and Boolean value definitions:

$t \models \mathbf{op}(p,v)$, if there is $x \in [\![p]\!]^t$ such that ($\mathsf{value}(x,t)$ $\mathbf{op}$ v) holds.
$t \models \mathbf{op}(p_1,p_2)$ if there are $x_1 \in [\![p_1]\!]^t$ and $x_2 \in [\![p_2]\!]^t$ such that ($\mathsf{value}(x_1,t)$ $\mathbf{op}$ $\mathsf{value}(x_2,t)$) holds.
$t \models v$, for an atomic value v, if $v \notin \{\mathbf{null}, \mathbf{false}, 0\}$.
$t \models p$, for a path p, if $t \not\models (p = v)$ for $v \in \{\mathbf{null}, \mathbf{false}, 0\}$.
$t \models d/d_1/d_2$, if $t \models d$ and $t \models d_1$, or if $t \not\models d$ and $t \models d_2$.
$t \models [d_1,\ldots,d_n]$, always.

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

The semantics of arbitrary Boolean value definitions is then obtained straightforwardly.

Note that comparisons are evaluated differently in criteria conditions and in value definitions: in $(p\ \mathbf{op}\ v)$, when $\mathsf{type}(x,t) = \mathsf{array}$ for some $x \in [\![p]\!]^t$, the comparison might hold due to a value inside the array, while in evaluating $\mathbf{op}(p,v)$ the array is not entered; moreover, $=(p, \mathbf{null})$ does not hold if $[\![p]\!]^t = \emptyset$.

To define the semantics of the *unwind*, *project*, and *group* operators, we make use of a number of auxiliary operators over trees, which we informally introduce here (a formal definition is given in Appendix A.2). Let t, t_1, t_2 be trees, F a forest, p a path, N a set of nodes, and x a node. Then: *(i)* $\mathsf{subtree}(t,x,N)$ returns the subtree of t rooted in x and induced by N; *(ii)* $\mathsf{subtree}(t,p)$ returns the subtree of t hanging from a path p. In the case where $|[\![p]\!]^t| > 1$, it returns the array of single subtrees; *(iii)* $\mathsf{attach}(p,t)$ constructs a new tree by attaching a path p on top of the root of t; *(iv)* $t_1 \setminus t_2$ returns the tree resulting from removing a subtree t_2 from a tree t_1; *(v)* $t_1 \oplus t_2$ constructs a new tree resulting from merging the two trees t_1 and t_2 by identifying nodes reachable via identical paths; and *(vi)* $\mathsf{array}(F,p)$ constructs a new tree that encodes 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. If $p = \varepsilon$, we write $\mathsf{array}(F)$.

Given a path referring to an array, unwind flattens it by creating a new tree for each element in the array. Unwinding non-arrays has no effect.

▶ **Definition 4** (Unwind ω)**.** Let p and q be paths and t a tree. For $i \in I$, denote the tree $\big(t \setminus \mathsf{subtree}(t,p)\big) \oplus \mathsf{attach}\big(p, \mathsf{subtree}(t,p.i)\big)$ by $\mathit{tree}_{t,p,i}$. We say that p is *flat* in t if $\mathsf{type}(p',t) \neq \mathsf{array}$, for each strict prefix p' of p. Below we use square brackets to indicate that merge is optional, and should be performed if the includeArrayIndex path q is defined.

$$\omega_p^{(q)}(t) = \begin{cases} \{\mathit{tree}_{t,p,i}\, [\oplus \mathsf{attach}(q,i)]\}_{i\in I,\ [\![p.i]\!]^t \neq \emptyset}, & \text{if } p \text{ is flat in } t \text{ and } \mathsf{type}(p,t) = \mathsf{array},\\ \{t\, [\oplus \mathsf{attach}(q, \mathbf{null})]\}, & \text{if } p \text{ is flat in } t,\ \mathsf{type}(p,t) \neq \mathsf{array}, \text{ and}\\ & t \not\models (p = \mathbf{null}),\\ \emptyset, & \text{otherwise.} \end{cases}$$

$$\omega_p^{(q,n)}(t) = \begin{cases} \omega_p^{(q)}(t), & \text{if } \omega_p^{(q)}(t) \neq \emptyset,\\ \{t\, [\oplus \mathsf{attach}(q, \mathbf{null})]\}, & \text{otherwise.} \end{cases}$$

Let F be a forest. Then $F \triangleright \omega_p^{(q,n)} = \bigcup_{t\in F} \omega_p^{(q,n)}(t)$. ◁

Project is similar to the extended projection of relational algebra.

▶ **Definition 5** (Project θ)**.** Let p, p' be paths, $c, d_1, \ldots, d_m$ value definitions, and t a tree. For a value definition d, denote by v_d the value associated to d in t, defined as d if $d \in V$, as $\mathsf{value}(\mathsf{subtree}(t,d))$ if d is a path, and as the value of $(t \models d)$ if d is a Boolean value definition.

$$\theta^p(t) = \mathsf{subtree}(t, \texttt{\_id}) \oplus \mathsf{subtree}(t, N_p), \text{ where } N_p \text{ are the nodes in } t \text{ that are on the path from } \mathsf{root}(t) \text{ to } x, \text{ or reachable from } x, \text{ for some } x \in [\![p]\!]^t.$$

$$\theta_{p=p'}(t) = \begin{cases} \mathsf{subtree}(t, \texttt{\_id}) \oplus \mathsf{attach}(p, \mathsf{tree}(v_{p'})), & \text{if } t \models \exists p'\\ \mathsf{subtree}(t, \texttt{\_id}), & \text{otherwise} \end{cases}$$

$$\theta_{p=d}(t) = \mathsf{subtree}(t, \texttt{\_id}) \oplus \mathsf{attach}(p, \mathsf{tree}(v_d))$$

$$\theta_{p=[d_1,\ldots,d_m]}(t) = \mathsf{subtree}(t, \texttt{\_id}) \oplus \mathsf{attach}(p, \mathsf{tree}([v_{d_1}, \ldots, v_{d_m}])),$$

$$\theta_{p=(c/d_1/d_2)}(t) = \theta_{p=d}(t) \text{ where } d = d_1 \text{ if } t \models c, \text{ and } d = d_2 \text{ otherwise}$$

Let $p_1, \ldots, p_n, q_1, \ldots, q_m$ be distinct paths none of which is a sub-path of another path.

$$\theta^{p_1,\ldots,p_n}_{q_1=d_1,\ldots,q_m=d_m}(t) = \bigoplus_{i=1}^n \theta^{p_i}(t) \ \oplus\ \bigoplus_{j=1}^m \theta_{q_j=d_j}(t)$$

Let F be a forest. Then $F \triangleright \theta^P_{VD} = \{\theta^P_{VD}(t) \mid t \in F\}$. $\triangleleft$

Group combines several trees in one tree according to the grouping condition g' stored in $\texttt{\_id}.g$, and stores the aggregation paths a' in the arrays a.

▶ **Definition 6** (Group γ)**.** Let F be a forest. Then,

$$\begin{aligned}
F \triangleright \gamma^{\mathbf{null}}_{a_1:b_1,\ldots,a_m:b_m} &= \big\{\mathsf{attach}(\texttt{_id}, \mathbf{null}) \oplus \textstyle\bigoplus_{i=1}^m \mathsf{attach}(a_i, \mathsf{array}(F, b_i))\big\}\\
F \triangleright \gamma^{g:y}_{a_1:b_1,\ldots,a_m:b_m} &= \big\{\mathsf{attach}(\texttt{_id}.g, t) \oplus \textstyle\bigoplus_{i=1}^m \mathsf{attach}(a_i, \mathsf{array}(F \triangleright \mu_{y=t}, b_i)) \mid t \in \mathsf{forest}(F, y)\big\}\\
F \triangleright \gamma^{g_1:y_1,\ldots,g_n:y_n}_{a_1:b_1,\ldots,a_m:b_m} &= \Big\{ \textstyle\bigoplus_{e\in E} \mathsf{attach}(\texttt{_id}.g_e, t_e) \oplus \bigoplus_{i=1}^m \mathsf{attach}\big(a_i, \mathsf{array}(F \triangleright \mu_\varphi, b_i)\big) \mid \\
&\qquad E \cup \bar{E} = \{1, \ldots, n\},\ E \neq \emptyset,\ E \cap \bar{E} = \emptyset,\\
&\qquad t_e \in \mathsf{forest}(F, y_e),\ \text{for each } e \in E,\\
&\qquad \varphi = \textstyle\bigwedge_{e\in E}((y_e = t_e) \wedge \exists y_e) \wedge \bigwedge_{e\in\bar{E}}(\neg\exists y_e),\ (F \triangleright \mu_\varphi) \neq \emptyset \Big\}\\
&\quad \cup \Big\{\mathsf{attach}(\texttt{_id}, \{\!\!\{\}\!\!\}) \oplus \textstyle\bigoplus_{i=1}^m \mathsf{attach}\big(a_i, \mathsf{array}(F \triangleright \mu_\varphi, b_i)\big) \mid \\
&\qquad\qquad \varphi = \textstyle\bigwedge_{i=1}^n(\neg\exists y_i),\ (F \triangleright \mu_\varphi) \neq \emptyset \Big\}
\end{aligned}$$

Here, in $\bigoplus_{e\in E}$ we assume that the elements in E are enumerated in the increasing order. $\triangleleft$

Lookup performs an outer left join with an external forest where the joining condition is $p_1 = C.p_2$ and the matching trees are stored in the array p.

▶ **Definition 7** (Lookup λ)**.** Let t be a tree, C a collection name, and F_2 a collection for C. Moreover, let p, p_1, p_2 be paths.

$$\lambda_p^{p_1=C.p_2}(t, F_2) = t \oplus \mathsf{attach}(p, \mathsf{array}(F_2 \triangleright \mu_{p_2=v_1})),\ \text{where } v_1 = \mathsf{value}(\mathsf{subtree}(t, p_1)).$$

Let F_1 be a forest. Then $F_1 \triangleright \lambda_p^{p_1=C.p_2}[F_2] = \{\lambda_p^{p_1=C.p_2}(t, F_2) \mid t \in F_1\}$. $\triangleleft$

Finally, we are ready to define the semantics of MUPGLs.

▶ **Definition 8.** Let $\boldsymbol{q} = C \triangleright s_1 \triangleright \cdots \triangleright s_n$ be an MUPG query, where C is a collection name and each s_i is a stage. The *result of evaluating* $\boldsymbol{q}$ *over* a MongoDB instance D, denoted $ans_{\mathsf{mo}}(\boldsymbol{q}, D)$, and if $\boldsymbol{q}$ contains no lookup operator, also denoted $D.C \triangleright s_1 \triangleright \cdots \triangleright s_n$, is defined as F_n, where $F_0 = D.C$, and for $i \in [1..n]$, $F_i = (F_{i-1} \triangleright s_i)$ 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 from a collection name C'. $\triangleleft$

5 What can be expressed by MUPGL

In this section we characterize the expressiveness of MUPG in terms of the relational algebra.

We start with a discussion of the abstract notion of *join*, whose goal is to combine information from two entities that share some values. The way values are shared is referred to as the *joining condition*. Apart from the newly added lookup feature, there is no straightforward way to perform joins in MongoDB. It is known that in relational algebra, joins constitute a source of complexity: SPJ queries are already NP-hard in combined complexity. Therefore, it is natural to ask whether MongoDB queries can express joins. Below we discuss three different types of joins relevant to the way MongoDB structures data, and then show on an example how to join information from different documents.

Since in MongoDB entities can be both documents and collections, we distinguish three types of joins: *inner-document*, *cross-document*, and *cross-collection* joins. An inner-document join combines information originating from the same document. A cross-document join combines information from several (possibly distinct) documents from the same collection. And finally, a cross-collection join combines information from arbitrary documents, i.e., possibly different documents from possibly different collections. Below we demonstrate how to express cross-document joins in MUPG by extending the technique used in Example 1.

▶ **Example 9.** Suppose we want to retrieve all pairs of scientists that received the same award in the same year. Since in our `bios` collection, each document stores information about one scientist, this query requires a cross-document join. This can be expressed by the following MUPG query:

```
db.bios.aggregate([
    {$unwind: "$awards"},
    {$project: { "awards": true, "doc._id": "$_id", "doc.name": "$name" }},
    {$group: {
        _id: { "awardYear": "$awards.year", "awardName": "$awards.award" },
        "docs": { $addToSet: "$doc" }}},
    {$project : { "doc1": "$docs", "doc2": "$docs" }},
    {$unwind: "$doc1"},
    {$unwind: "$doc2"},
    {$project: {
        "lastName1": "$doc1.name.last", "lastName2": "$doc2.name.last",
        "awardName": "$_id.awardName", "awardYear": "$_id.awardYear",
        "toJoin": { $ne: ["$doc1._id", "$doc2._id"] }}},
    {$match: { "toJoin": true }},
    {$project : { "_id": false,
        "lastName1": true, "lastName2": true, "awardName": true, "awardYear": true }}
])
```

With the evidence that we can express joins (even without lookup), we can ask ourselves a natural question: can we capture full relational algebra by MongoDB queries? In the rest of this section, we answer to it positively for the class of MUPGL queries, by developing a translation from relational algebra to MUPGL.

5.1 Relational view of MongoDB databases

Before developing the correspondence between relational algebra and MongoDB queries, it is necessary to define the relational database corresponding to a MongoDB database. To this purpose, we define a relational view of MongoDB databases.

In the context of MongoDB, a path corresponds to a relational attribute. Therefore, the attributes of a *MongoDB relational signature*, or simply *signature*, is a set of paths $\{p_1, \dots, p_m\}$. We illustrate a relational view over such a signature in the example below.

▶ **Example 10.** Consider the document in Figure 1. Then we can naturally view it as the following relation $R_{\texttt{bios}}$:

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

Note that $R_{\texttt{bios}}$ consists of 6 tuples as this relational view implicitly "unwinds" the two arrays `award` and `contribs`, containing 3 and 2 elements, respectively. ◁

In the general case, if a document t contained n arrays with $k_1, \dots, k_n$ elements, respectively, the relation obtained from the document according to this principle would contain $k_1 \cdot k_2 \cdots k_n$ tuples. Hence, this natural relational view might be exponential in the size of t, and in general cannot be computed efficiently in the size of the data. To define a compact relational view, we need to detect arrays that could interact, and take them into account when constructing the relational view. To do this independently of the actual database instance, we introduce the notion of type constraint specifying the type of a path as one of array, $\mathsf{literal}$, and object.

▶ **Definition 11.** A *(MongoDB) type constraint* is a triple (C, p, type) or $(C, p.\#, \mathit{type})$, where C is a collection name, p a path, and type is one of object, array, or $\mathsf{literal}$. A database instance D *satisfies* the type constraint (C, p, type), if for each document $t \in D.C$, we have that $\mathsf{type}(p, t) = \mathit{type}$, and it satisfies $(C, p.\#, \mathit{type})$, if it satisfies (C, p, array) and for each each document $t \in D.C$, index $i \in I$, and $x \in [\![p.i]\!]^t$, we have that $\mathsf{type}(x, t) = \mathit{type}$. ◁

For the rest of this section, we fix a set $\mathcal{S}$ of MongoDB type constraints. For simplicity, for a type constraint $(C, p.\#, \mathit{type}) \in \mathcal{S}$, we assume that $\mathit{type} \neq \mathsf{array}$, and thus, we rule out the case of arrays of arrays. For each collection name C, we now define the corresponding signature and relational schema with respect to $\mathcal{S}$, where intuitively each relation signature in the schema corresponds to one constraint in $\mathcal{S}$ of type array for C.

▶ **Definition 12.** Let C be a collection name appearing in $\mathcal{S}$. The *signature* $\mathsf{sig}_{\mathcal{S}}(C)$ *of* C *with respect to* $\mathcal{S}$ is defined as having attributes $\{p \mid (C, p, \mathsf{literal}) \in \mathcal{S} \text{ or } (C, p.\#, \mathsf{literal}) \in \mathcal{S}\}$.

Let $\mathsf{arr}_{\mathcal{S}}(C)$ be the set $\{p_1, \ldots, p_n\}$ of all paths such that $(C, p_i, \mathsf{array}) \in \mathcal{S}$ and p_i is a prefix of some path in $\mathsf{sig}_{\mathcal{S}}(C)$. We partition $\mathsf{sig}_{\mathcal{S}}(C)$ into $n+1$ signatures $P_0, P_1, \ldots, P_n$, where the P_is are defined as follows:

- $\mathsf{att}(P_0) = \{\texttt{\_id}\} \cup \{p \in \mathsf{sig}_{\mathcal{S}}(C) \mid \text{none of } p_j, \text{ for } j \in [1..n], \text{ is a prefix of } p\}$.
- $\mathsf{att}(P_i) = \{\texttt{\_id}\} \cup \{p_j.\text{index} \mid j \in [1..n] \text{ and } p_j \text{ is a prefix of } p_i\}$
  $\cup \{p \in \mathsf{sig}_{\mathcal{S}}(C) \mid p_i \text{ is the longest prefix of } p \text{ among } p_1, \ldots, p_n\}$,
  for $i \in [1..n]$. We call P_i a *signature with indexes*.

Then the *relational schema* $\mathsf{rschema}_{\mathcal{S}}(C)$ of C with respect to $\mathcal{S}$ is defined as $\{P_0, \ldots, P_n\}$. The *relational schema* of $\mathcal{S}$, denoted $\mathsf{rschema}_{\mathcal{S}}$, is defined as $\bigcup_{C \text{ in } \mathcal{S}} \mathsf{rschema}_{\mathcal{S}}(C)$. ◁

▶ **Example 13.** Consider the following set $\mathcal{S}_b$ of type constraints for the `bios` collection:

(`bios`, `awards`, array)	(`bios`, `birth`, literal)	(`bios`, `name`, object)
(`bios`, `awards.award`, literal)	(`bios`, `contribs`, array)	(`bios`, `name.first`, literal)
(`bios`, `awards.year`, literal)	(`bios`, `contribs.#`, literal)	(`bios`, `name.last`, literal)

Then, the relational schema $\mathsf{rschema}_{\mathcal{S}_b}(\texttt{bios})$ consists of P_0, P_1, and P_2 defined as follows:

$\mathsf{att}(P_0) = \{\texttt{\_id}, \texttt{birth}, \texttt{name.first}, \texttt{name.last}\}$
$\mathsf{att}(P_1) = \{\texttt{\_id}, \texttt{contribs.index}, \texttt{contribs}\}$
$\mathsf{att}(P_2) = \{\texttt{\_id}, \texttt{awards.index}, \texttt{awards.award}, \texttt{awards.year}\}$ ◁

Next, we show how to compute the relational view of a MongoDB collection/database with respect to the relational signatures and schemas defined above. In this view, we distinguish between existing paths with **null** value, and missing paths. To this purpose, we introduce a new constant **missing**.

▶ **Definition 14.** Let F be a forest satisfying $\mathcal{S}$ and P a signature. Then, the *relational view of* F *with respect to* P, denoted $\mathsf{rel}_P(F)$, is the relation defined as follows:

- When P is a signature (without indexes) with $\mathsf{att}(P) = \{p_1, \ldots, p_m\}$:

$$\mathsf{rel}_P(F) = \Big\{ \{p_\ell : v_\ell\}_{\ell=1}^m \mid t \in F \text{ and for } \ell \in [1..m],\ t \models (\exists p_\ell) \wedge (p_\ell = v_\ell),\ \text{for } v_\ell \in V \text{ or } t \not\models \exists p_\ell \text{ and } v_\ell = \textbf{missing} \Big\}.$$

- When P is a signature with indexes with $\mathsf{att}(P) = \{\mathtt{\_id}, a_1.\mathrm{index}, \ldots, a_k.\mathrm{index}, a_k.p_1, \ldots, a_k.p_m\}$, where a_i is of the form $a_1.b_2 \ldots b_i$, for $2 \leq i \leq k$, we first define two auxiliary functions. Given a tree t and a node x in t, we have that $\mathsf{index}(x,t)$ is the sequence of indexes $i_1 \ldots i_k$ obtained by removing the keys from $\mathsf{ipath}(x,t)$. Given indexes $i_1, \ldots, i_k$ and a path $p \in \{p_1, \ldots, p_m\}$, the function $val(i_1, \ldots, i_k, p, t)$ is defined as:
$$\begin{cases} v \in V, & \text{if there is } x \in [\![a_k.p]\!]^t \cup [\![a_k.i_k]\!]^t, \text{ with } L_n(x) = v \text{ and } \mathsf{index}(x,t) = (i_1, \ldots, i_k), \\ \mathbf{missing}, & \text{if there does not exist } x \text{ such that } \mathsf{ipath}(x,t) = a_1.i_1.b_2.i_2 \ldots b_k.i_k.p, \end{cases}$$
Then $\mathsf{rel}_P(F)$ is defined as the set of all tuples
$$\{\mathtt{\_id} : id,\ a_1.\mathrm{index} : i_1,\ \ldots,\ a_k.\mathrm{index} : i_k,\ a_k.p_1 : v_1,\ \ldots,\ a_k.p_m : v_m\}$$
such that there exists $t \in F$ with $t \models (\mathtt{\_id} = id)$ and *(i)* if there exist $\{i_1, \ldots, i_k\} \subseteq I$ such that $\mathsf{index}(x,t) = (i_1, \ldots, i_k)$ for some $x \in [\![a_k.i_k]\!]^t$, then $v_\ell = val(i_1, \ldots, i_k, p_\ell, t)$, for $\ell \in [1..m]$; *(ii)* if there exist $\{i_1, \ldots, i_n\} \subseteq I$, for $0 \leq n < k$, such that $\mathsf{index}(x,t) = (i_1, \ldots, i_n)$ for some $x \in [\![a_n.i_n]\!]^t$, and there does not exist $y \in [\![a_{n+1}.0]\!]^t$ such that $\mathsf{index}(y,t) = (i_1, \ldots, i_n, 0)$, then $i_{n+1} = \cdots = i_k = v_1 = \cdots = v_m = \mathbf{missing}$. ◁

Notice that, in the above definition, $\mathsf{rel}_P(F)$ for P a signature without indexes, is indeed well defined for arbitrary forests F.

We also observe that, in order to correctly capture the MongoDB semantics of missing paths and of **null** (different occurrences of which *do* join, and behave like missing paths, cf. also Section 7), we cannot use "`NULL`" of SQL, but need to introduce the special constant **missing**, which we assume does not belong to the keys K and literals V. Therefore, the relational view we obtain is always a *complete* database.

▶ **Definition 15.** Let D be a MongoDB database instance satisfying $\mathcal{S}$. The *relational view* of D with respect to $\mathcal{S}$ and a collection name C, denoted $\mathsf{rdb}_{\mathcal{S}}(D,C)$, is the relational database instance $\{\mathsf{rel}_P(D.C) \mid P \in \mathsf{rschema}_{\mathcal{S}}(C)\}$. The *relational view* $\mathsf{rdb}_{\mathcal{S}}(D)$ of D with respect to $\mathcal{S}$ is the instance $\bigcup_{C \text{ in } \mathcal{S}} \mathsf{rdb}_{\mathcal{S}}(D,C)$. ◁

For each collection name C (appearing) in $\mathcal{S}$, we can also define a *virtual relational view* $\mathsf{vrel}_{\mathcal{S}}(D,C)$ that is a single relation. It is obtained by (naturally) joining the relations in $\mathsf{rdb}_{\mathcal{S}}(D,C)$, i.e., $\mathsf{vrel}_{\mathcal{S}}(D,C) = \pi_{\mathsf{sig}_{\mathcal{S}}(C)}(R_0 \bowtie \cdots \bowtie R_n)$, where $\{R_0, \ldots, R_n\} = \mathsf{rdb}_{\mathcal{S}}(D,C)$.

▶ **Example 16.** Let D be a database instance in which the collection `bios` contains the single document t shown in Figure 1. The relational view $\mathsf{rdb}_{\mathcal{S}_b}(D, \mathtt{bios})$ is a database consisting of the following three relations:

$R_0 =$

`_id`	birth	name.first	name.last
4	1926-08-27	Kristen	Nygaard

$R_1 =$

`_id`	contribs.index	contribs
4	0	OOP
4	1	Simula

$R_2 =$

`_id`	awards.index	awards.award	awards.year
4	0	Rosing Prize	1999
4	1	Turing Award	2001
4	2	IEEE John von Neumann Medal	2001

◁

Finally, given a MongoDB query and a relational algebra query, we define when the two can be considered as equivalent. To this purpose, we define equivalence between two kinds of answers: trees in the former case, and named tuples in the latter case.

▶ **Definition 17.** Let P be a signature. A tree t is P-*equivalent* to a P-tuple a, denoted $t \simeq_P a$, if $\mathsf{rel}_P(\{t\}) = \{a\}$. ◁

▶ **Definition 18.** Let Q be a relational query over $\mathsf{rschema}_{\mathcal{S}}$ with output signature P. Then a MongoDB query $\boldsymbol{q}$ is *equivalent to Q w.r.t. $\mathcal{S}$*, denoted by $\boldsymbol{q} \equiv_{\mathcal{S}} Q$, if for each database instance D satisfying $\mathcal{S}$, we have that *(i)* for each $a \in ans_{\mathsf{ra}}(Q, \mathsf{rdb}_{\mathcal{S}}(D))$ there is $t \in ans_{\mathsf{mo}}(\boldsymbol{q}, D)$ s.t. $t \simeq_P a$, and *(ii)* for each $t \in ans_{\mathsf{mo}}(\boldsymbol{q}, D)$ there is $a \in ans_{\mathsf{ra}}(Q, \mathsf{rdb}_{\mathcal{S}}(D))$ s.t. $t \simeq_P a$. ◁

5.2 Relational algebra to MongoDB queries

We now show that relational algebra can be fully captured by MUPGL, while MUPG captures relational algebra over (the relational view of) a single collection. Due to space limitations, we omit the actual encoding from the main text and describe only the structure of the translation. The detailed translation and its description are reported in Appendix A.3.

- Given as input a set $\mathcal{S}$ of type constraints, our translation ra2maq is such that the result of each obtained MUPG(L) query over a database instance satisfying $\mathcal{S}$ is a forest, where each tree is equivalent to a P-tuple and P is the signature of the relational algebra result.
- First, we translate the SPJ algebra to MUPG queries, essentially generalizing the technique illustrated in Examples 1 and 9 (cf. Section A.3.1).
- Then we extend the translation to arbitrary RA expressions over (the relational view of) a single collection, thus showing that MUPG queries (over a single collection) are at least as expressive as full relational algebra (cf. Section A.3.2). This extension is concerned with translating set union and set difference (which is relatively easy), and with nesting arbitrary relational algebra expressions. To deal with arbitrary nesting, we develop a general approach to translate MUPG subqueries, and use an encoding where the input relations are stored in arrays. Namely, if R_1 and R_2 are two relations over a signature S, then we assume to have as input a single tree with two key-array pairs, where one array contains R_1 and the other array contains R_2. The subquery technique then combines two MUPG queries into a single MUPG query in such a way that its result is a single tree that contains the results of the original two queries in two arrays.
- Finally, we show how to express cross-document joins, and thus obtain the complete translation for RA over the relational view of multiple collections (cf. Section A.3.3).

The following theorem establishes the correctness of the translation ra2maq.

▶ **Theorem 19.** *The translation* ra2maq *is correct. That is, for each relational algebra query* Q *over* $\mathsf{rschema}_{\mathcal{S}}$, $\mathsf{ra2maq}(Q) \equiv_{\mathcal{S}} Q$.

▶ **Theorem 20.** MUPGL *captures full relational algebra, and* MUPG *captures relational algebra over a single collection. Moreover, inner-document joins can be expressed in* MUP.

We observe that the goal of ra2maq is to provide a conceptually simple translation, at the cost of sacrificing the scalability of the translation with query size. Moreover the resulting queries might not be executable in practice, due to the assumption of storing input relations in one tree, which might lead to violation of the maximum document size constraint (16MB). However, the translation can be implemented differently in practice, and we have developed also an alternative, more involved, translation of binary RA constructs. We have also devised optimization techniques that allow us to produce queries that execute more efficiently than the ones obtained with the more direct translation. These techniques, and evidence about their effectiveness are reported in Section A.3.6.

6 The Complexity of MongoDB Queries

In this section we report on a preliminary study of the complexity of different fragments of MUPGL queries. Specifically, we are concerned with the combined and query complexity of the *Boolean query evaluation* problem, which is the problem of checking whether the answer to a given query over a given database instance is non-empty. Our first result is that MFQs (and hence match queries) are tractable and very efficient.

▶ **Lemma 21.** *Boolean query evaluation for* MFQ *queries is in* LogSpace.

Adding the unwind operator causes loss of tractability.

▶ **Lemma 22.** *Boolean query evaluation for* MU *queries is* NP*-complete.*

As a corollary, we obtain that query evaluation for MUP and MUL queries is NP-hard already in query complexity.

It follows from the translation from RA to MUPG that MUPG queries are PSpace-hard. The translation however uses quite powerful project operators such as conditional value definitions, or the introduction of new arrays. Here, we show that MUPG queries are PSpace-hard even without the project operator.

▶ **Lemma 23.** *Boolean query evaluation for* MUG *queries is* PSpace*-hard.*

We can identify the unwind operator as one of the sources of complexity, as it allows one to generate an exponential number of trees in the pipeline. The project operator turns out to be also quite powerful as it allows one to create new values by duplicating the existing ones; hence, it can make trees grow exponentially in the size of the query. Next, we show that evaluation of MP queries with additional array operators *filter* and *map*, which allow for filtering out and for transformation of the elements inside an array, respectively, is NP-hard already in query complexity.

▶ **Lemma 24.** *Boolean query evaluation for* MP *queries with filter and map operators is NP-hard in query complexity.*

However, if we restrict the project operator so as to disallow duplication of existing paths (and hence disallow creation of exponentially large arrays or objects), and similarly with the group operator (in principle, value duplication can also be done by group), then the size of the trees can grow only polynomially in the size of the query. Such restricted MPG queries, which we denote with MPG$^-$, turn out to be PTime-complete.

▶ **Lemma 25.** *Query evaluation for* MPG$^-$ *queries is* PTime*-complete.*

7 Lessons to Be Learned

We discuss now some features of MongoDB that emerged in our investigation, and that is worth pointing out. Some of these might be considered as counterintuitive, at least to users familiar with relational databases and SQL, or could even appear as inconsistencies in the semantics of operators.

Comparison of null values. SQL employs three-valued semantics, where each occurrence of `NULL` is treated as a *fresh* unknown value, and the expression (`NULL` = `NULL`) evaluates to `NULL` (hence is not true). On the other hand, MongoDB works under two-valued semantics, where **null** is treated as a constant and (**null** = **null**) evaluates to true. Strangely, in comparisons done within **$project** (but not within **$match**), **null** is considered less than any constant, in particular (**null** $< -\infty$). Since there is no rationale for this, we consider this as a bug.

Group. The group operator behaves differently for grouping by one path and grouping by multiple paths, as shown in Definition 6. Namely, in the former case **missing** is treated as **null**, while in the latter case it is treated differently. More specifically, when grouping by one path (e.g. $\gamma^{g:y}_{\dots}$), MongoDB puts the trees with $y =$ **null** or y missing into the same group with `_id` $= \{\!\{g : \textbf{null}\}\!\}$. On the contrary, when grouping with multiple paths (e.g., $\gamma^{g_1:y_1,\ g_2:y_2}_{\dots}$), the trees with all y_i missing are put into a separate group with `_id` $= \{\!\{\}\!\}$.

Comparing value and array path. The criteria in match and Boolean value definitions in project behave differently. For instance, when comparing a path p of type array with a value v using equality, match checks (1) if v is exactly the array value of p; or (2) if v is an element inside the array value of p; instead, project only checks condition (1). Moreover, for match ($p = \mathbf{null}$) holds (a) when p exists and its value is **null**, or (b) when p is missing; instead, for project $=(p, \mathbf{null})$ holds only for (a).

Construction of exponentially large objects and arrays. The project and group operators have the ability to rename and to duplicate the existing values. This feature can easily lead to the creation of trees that are exponentially large in the number of repetitions of such operators (see for instance Lemma 24), which might not be expected or wanted by users.

8 Conclusions

In this work we have carried out a first formal investigation of MongoDB, a widely used noSQL database/document management system, with the aim of understanding its query capabilities and expressiveness, and have obtained preliminary results of the complexity of various fragments of its query language.

We are extending our work in the following directions:

- Establishing tight complexity bounds for MUPGL and its fragments.
- Devising a translation from MUPGL to relational algebra to better understand the relationship between these two query languages. We have so far devised an exponential reduction in general, due to the possibility of generating exponentially large objects, and hence, exponentially many distinct paths which have to appear as relational attributes.
- Applying the results of this paper, and specifically the translation from RA to MUPGL, for accessing MongoDB using a language that is user-friendly [1], thus avoiding hard-coded post-processing transformations. Our aim is to extend the ontology-based data access so as to include also MongoDB data sources [3].

References

1 Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. Uniform access to non-relational database systems: The SOS platform. In *Proc. of the 24th Int. Conf. on Advanced Information Systems Engineering (CAiSE)*, volume 7328 of *Lecture Notes in Computer Science*, pages 160–174. Springer, 2012.

2 Véronique Benzaken, Giuseppe Castagna, Kim Nguyen, and Jérôme Siméon. Static and dynamic semantics of NoSQL languages. *SIGPLAN Notices*, 48(1):101–114, January 2013.

3 Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Martin Rezk, and Guohui Xiao. OBDA beyond relational DBs: A study for MongoDB. In *Proc. of the 2016 Int. Workshop on Description Logics (DL)*, CEUR Workshop Proceedings, 2016. To appear.

4 Samuel R. Buss. The boolean formula value problem is in ALOGTIME. In *Proc. of the 19th ACM SIGACT Symp. on Theory of Computing (STOC)*, pages 123–131. ACM Press, 1987.

5 Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, May 2011.

6 Neal Leavitt. Will NoSQL databases live up to their promise? *Computer*, 43(2):12–14, February 2010.

7 Frank Neven and Thomas Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proc. of the 19th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*, pages 145–156. ACM Press, 2000.

8 Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. The SQL++ query language: Configurable, unifying and semi-structured. CoRR Technical Report abs/1405.3631, arXiv.org e-Print archive, 2014. Available at `http://arxiv.org/abs/1405.3631`.

9 Michael Stonebraker and Ugur Cetintemel. "one size fits all": An idea whose time has come and gone. In *Proc. of the 21st IEEE Int. Conf. on Data Engineering (ICDE)*, pages 2–11, 2005.

10 Balder ten Cate and Maarten Marx. Axiomatizing the logical core of XPath 2.0. *Theoretical Computer Science*, 44(4):561–589, 2009.

A Appendix

A.1 Examples of MongoDB Queries

MongoDB provides two main query mechanisms. The basic form of query is a *find* query, which allows one to filter out documents according to some (Boolean) criteria and to return, for each document passing the filter, a tree containing a subset of the key-value pairs in the document. Specifically, a find query has two components, where the first one is a *criterion* for selecting documents, and the second one is a *projection condition*.

▶ **Example 26.** The following MongoDB find query selects from the `bios` collection the documents talking about scientists whose first name is Kristen, and for each document only returns the full name and the date of birth.

```
db.bios.find(
    {"name.first": {$eq: "Kristen"}},
    {"name" : 1, "birth" : 1}
)
```

When applied to the document in Example 1, it returns the following tree:

```
{   "_id": 4,
    "birth": "1926-08-27",
    "name": {
        "first": "Kristen", "last": "Nygaard" }
}
```

Observe that by default the document identifier is included in the answer of the query, hence by default the answer is a document. ◁

Note that with a find query we can either obtain the original documents as they are, or we can modify them by specifying in the projection condition only a subset of the keys, thus retaining in the answer only the corresponding key-value pairs. However, we cannot change the shape of the individual pairs.

A more powerful querying mechanism is provided by the *aggregation framework*, in which a query consists of a pipeline of *stages*, each transforming a forest into a new forest. We call this transformation pipeline an *aggregate query*. One of the main differences with find queries is that aggregate queries can manipulate the shape of the trees.

▶ **Example 27.** The following MongoDB aggregate query essentially does the same as the previous find query, but now it flattens the complex object `name` into two key-value pairs.

```
db.bios.aggregate([
    {$match: {"name.first": {$eq: "Kristen"}}},
    {$project: {
        "birth": true, "firstName": "$name.first", "lastName": "$name.last" } }
])
```

So the document from our running example will be transformed into the following tree:

```
{   "_id" : 4,
    "birth": "1926-08-27",
    "firstName": "Kristen",
    "lastName": "Nygaard"
}
```

◁

▶ **Example 28.** Consider the query in Example 1 which is an aggregate query consisting of 6 stages that retrieves all persons who received two awards in one year. The first stage flattens the complex object `name`, creates two copies of the array `awards`, and projects away all other fields. The second and third stages flatten (unwind) the two copies (`award1` and `award2`) of the array of awards (which intuitively creates a cross-product). The fourth step compares awards pairwise and creates a new key (`twoInOneYear`) whose value is true if the scientist has two awards in one year. The fifth one selects the documents of interests (those where `twoInOneYear` is true), and the final stage renames and projects keys.

By applying the query to the document in Example 1, we obtain:

```
{   "_id": 4,
    "firstName": "Kristen",
    "lastName": "Nygaard",
    "awardName1": "IEEE John von Neumann Medal",
    "awardName2": "Turing Award",
    "year": 2001
}
```

◁

We note that the unwind operator creates a new document for every element in the array. Thus, unwinding `awards` (once) in the document in our running example will output 3 documents, only one of which satisfies the subsequent selection stages. In the example below we illustrate the group stage, which combines different documents into one.

▶ **Example 29.** The following query returns for each year all scientists that received an award in that year.

```
db.bios.aggregate([
    {$unwind: "$awards"},
    {$group: {
        _id: {"year": "$awards.year"}, "names": {$push: "$name"} }},
])
```

Running this query over the database consisting of the document in Figure 1, produces the following output:

```
{   "_id": { "year": 2001 },
    "names": [
        { "first": "Kristen", "last": "Nygaard" },
        { "first": "Kristen", "last": "Nygaard" } ]
},
{   "_id": { "year": 1999 },
    "names": [
        { "first": "Kristen", "last": "Nygaard" } ]
}
```

◁

We note that in terms of the abstract tree query languages proposed in [7], MFQ corresponds to pattern matching (with projection), while MAQ goes beyond pattern matching allowing also for a "construct" phase.

A.2 Tree operations

In the following, let $t = (N, E, \prec, L_n, L_e)$ 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 $n \in M$ and $M \subseteq N$, denoted $\mathsf{subtree}(t, x, M)$, is defined as $(N', E|_{N' \times N'}, \prec|_{E'}, L_n|_{N'}, L_e|_{E'})$ where N' is the subset of nodes in M reachable from x through nodes in M. 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 $(N', E', \prec, L'_n, L'_e)$, where

- $N' = N \cup \{x_0, x_1, \ldots, x_{n-1}\}$, for fresh $x_0, \ldots, x_{n-1}$.
- $E' = E \cup \{(x_0, x_1), (x_1, x_2), \ldots, (x_{n-1}, \mathsf{root}(t))\}$,
- $L'_n = L_n \cup \{(x_0, `\{\!\{\}\!\}'), \ldots, (x_{n-1}, `\{\!\{\}\!\}')\}$,
- $L'_e = L_e \cup \{((x_0, x_1), k_1), \ldots, ((x_{n-2}, x_{n-1}), k_{n-1}), ((x_{n-1}, \mathsf{root}(t)), k_n)\}$.

intersection Let t_1 and t_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 $(x_0, x_1), \ldots, (x_{n-1}, x_n)$ in E^1, for $x_0 = \mathsf{root}(t_1)$, and $(y_0, y_1), \ldots, (y_{n-1}, y_n)$ in E^2, for $y_0 = \mathsf{root}(t_2)$, with $L^1_N(x_i) = L^2_N(y_i)$ and $L^1_E(x_{i-1}, x_i) = L^2_E(y_{i-1}, y_i)$, for $1 \leq i \leq n$.

merge Let t_1, t_2 be trees $(N^j, E^j, \prec^j, L^j_n, L^j_e)$, $j = 1, 2$, 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 $(N, E, \prec, L_n, L_e)$, where

- $N = N^1 \cup N^{2\prime}$, for $N^{2\prime} = N^2 \setminus \{x_2 \mid (x_1, x_2) \in t_1 \cap t_2\}$
- $E = E^1 \cup (E^2 \cap (N^{2\prime} \times N^{2\prime})) \cup ((t_1 \cap t_2) \circ E^2)$
- $\prec = \prec^1 \cup \prec^2 \cup \{(x_1, x_2) \mid x_1 \in N^1, x_2 \in N^{2\prime},\ x_1, x_2 \text{ are siblings in } (N, E)\}$
- $L_n = L^1_n \cup L^2_n|_{N^{2\prime}}$
- $L_e = L^1_e \cup L^2_e|_{N^{2\prime} \times N^{2\prime}} \cup \{((x_1, y_2), \ell) \mid L^2_e(y_1, y_2) = \ell, (x_1, y_1) \in t_1 \cap t_2\}$

minus $t_1 \setminus t_2$ is $\mathsf{subtree}(t_1, N')$ where $N' = N_1 \setminus N_2$.

array Let $\{t_1, \ldots, t_n\}$, $n \geq 0$, be a forest and p a path. The operator $\mathsf{array}(\{t_1, \ldots, t_n\}, p)$ creates the tree encoding the array of the values of the path p in the trees $t_1, \ldots, t_n$. Let $t^p_j = \mathsf{subtree}(t_j, p)$ with $(N^j, E^j, \prec^j, L^j_n, L^j_e)$ where all N^j are mutually disjoint, and $r_j = \mathsf{root}(t^p_j)$. Then, $\mathsf{array}(\{t_1, \ldots, t_n\}, p)$ is the tree $(N, E, \prec, L_n, L_e)$ where

- $N = \left(\bigcup_{j=1}^n N^j\right) \cup \{v_0\}$,
- $E = \left(\bigcup_{j=1}^n E^j\right) \cup \{(v_0, r_1), \ldots, (v_0, r_n)\}$,
- $\prec = \left(\bigcup_{j=1}^n \prec^j\right) \cup \{(r_1, r_2), (r_2, r_3), \ldots, (r_{n-1}, r_n)\}^\star$, where $\star$ is the transitive closure,
- $L_n = \left(\bigcup_{j=1}^n L^j_n\right) \cup \{(v_0, `[\,]')\}$,
- $L_e = \left(\bigcup_{j=1}^n L^j_e\right) \cup \{((v_0, r_1), 0), \ldots, ((v_0, r_n), n-1)\}$.

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, $\mathsf{subtree}(t, p) = \mathsf{array}(\{t_1, \ldots, t_n\}, \varepsilon)$, where $\{r_1, \ldots, r_n\} = [\![p]\!]^t$, 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.

A.3 Relational algebra to MongoDB queries

In this section, we first develop a translation from relational algebra expressions over the relational view of a single collection to MUPG queries. Then we show how to express cross-collection joins (and hence, relational algebra expressions over the relational view of multiple collections) when lookup operator is available. In this translation, the result of each obtained MUPG(L) query is a forest, where each tree is equivalent to a P-tuple, where P is the signature of the relational algebra result. We start by showing how to translate the basic SPJ algebra in Section A.3.1. Then we provide in Section A.3.2 the translation for arbitrary RA expressions, and in Section A.3.3 the translation for RA over the relational view of multiple collections.

A.3.1 Translation of SPJ algebra to MUPG queries

We fix a set $\mathcal{S}$ of MongoDB type constraints, and a collection name C.

We start by showing how to "compute" the virtual relational view, that is, we provide a translation of the expression $\pi_{\mathsf{sig}_{\mathcal{S}}(C)}(P_0 \bowtie \cdots \bowtie P_n)$ that corresponds to the virtual relational view $\mathsf{sig}_{\mathcal{S}}(C)$, where $\mathsf{rschema}_{\mathcal{S}}(C) = \{P_0, \ldots, P_n\}$. Assume that $\mathsf{arr}_{\mathcal{S}}(C) = \{p_1, \ldots, p_n\}$ and $p_1, \ldots, p_n$ are sorted by non-decreasing length. Then, given an input collection F for C satisfying $\mathcal{S}$, the following MUPG query transforms it into a forest, where each tree corresponds to a $\mathsf{sig}_{\mathcal{S}}(C)$-tuple:

$$\mathsf{ra2maq}(\mathsf{sig}_{\mathcal{S}}(C)) = \omega^n_{p_1} \triangleright \ldots \triangleright \omega^n_{p_n} \triangleright \theta^{\mathsf{sig}_{\mathcal{S}}(C)}. \qquad \text{(flatten)}$$

This query unwinds all arrays, and then projects away the paths that are not in $\mathsf{sig}_{\mathcal{S}}(C)$. In what follows, we assume that all relational algebra queries are defined over the single database relation $\mathsf{sig}_{\mathcal{S}}(C)$.

Next, assume that S_1 and S_2 are subsets of $\mathsf{sig}_{\mathcal{S}}(C)$ and we aim at joining relations over S_1 and S_2. In order to join such relations, we collect in one tree all S_i-tuples in two arrays. The following query $\mathsf{rel2array}(S_1, S_2)$ returns such a tree consisting of two arrays `rel`i.

$$\mathsf{rel2array}(S_1, S_2) = \mathsf{ra2maq}(\mathsf{sig}_{\mathcal{S}}(C)) \triangleright \theta_{\{\mathrm{doc}i.p=p \,\mid\, p\in S_i\}_{i=1,2}} \triangleright \gamma^{\mathbf{null}}_{\{\mathrm{rel}i\colon \mathrm{doc}i\}_{i=1,2}}. \qquad \text{(subrelations)}$$

It should be clear that this query can be easily extended to the case of k relations. Finally, a translation $\mathsf{ra2maq}$ from the cross-product, select and project operators over $\mathsf{sig}_{\mathcal{S}}(C)$ to MUPG is presented below.

$$\mathsf{ra2maq}(S_1 \times S_2) = \mathsf{rel2array}(S_1, S_2) \triangleright \omega_{\mathrm{rel1}} \triangleright \omega_{\mathrm{rel2}} \qquad \text{(cross-product)}$$

$$\mathsf{ra2maq}\big(\sigma_\psi(Q)\big) = \mathsf{ra2maq}(Q) \triangleright \theta^{\mathsf{sig}(Q)}_{\mathrm{satCond}=\psi} \triangleright \mu_{\mathrm{satCond}=\mathbf{true}} \triangleright \theta^{\mathsf{sig}(Q)} \qquad \text{(select)}$$

$$\mathsf{ra2maq}\big(\pi_{S'}(Q)\big) = \mathsf{ra2maq}(Q) \triangleright \theta^{S'} \qquad \text{(project)}$$

where $\mathsf{sig}(Q)$ denotes the output signature of a relational algebra query Q. The join of relations over S_1 and S_2 takes as input the result of the query $\mathsf{rel2array}(S_1, S_2)$, and consecutively unwinds all doc$i$'s. Hence, the result of the query $\mathsf{ra2maq}(S_1 \times S_2)$ is a forest containing one tree for each pair of S_1- and S_2-tuples. Finally, the select σ_ψ and project $\pi_{P'}$ operators are translated straightforwardly. Since match does not allow for comparing the values of two paths, the select operator is encoded using ψ as the Boolean value definition. It is straightforward to generalize the translation above to the case of k relations participating in the join. Since every SPJ expression involving a join can be equivalently represented by a query of the form $\pi_S(\sigma_\psi(S_1 \times \cdots \times S_k))$ the above translation is complete. We observe that the joining condition of the form $p_1 = p_2$ is translated as a criterion $(\exists p_1 \wedge \exists p_2 \wedge (p_1 = p_2)) \vee (\neg\exists p_1 \wedge \neg\exists p_2)$, and the joining condition of the form $p_1 \neq p_2$ is translated as a criterion $(\exists p_1 \wedge \exists p_2 \wedge (p_1 \neq p_2)) \vee (\exists p_1 \wedge \neg\exists p_2) \vee (\neg\exists p_1 \wedge \exists p_2)$.

A.3.2 Translation of Full Relational Algebra to MUPG

To obtain a translation of full relational algebra to MUPG, it sufficient to show how to translate set union and set difference, and then how to nest relational algebra operators arbitrarily. To this end, we use the encoding where the input relations are stored in arrays as it was done by the query $\mathsf{rel2array}(S_1, S_2)$. Namely, if R_1 and R_2 are two relations over a signature S, then we assume to have as input a single tree with two key-array pairs, where `rel1` is the array containing R_1 and `rel2` is the array containing R_2.

MongoDB provides the following `ValueDef` operators implementing standard set operations with straightforward semantics:

$$\begin{array}{lrl} \{\$\text{setUnion: } [\texttt{List<PathRef>}]\} & d ::= & p \cup p' \\ \{\$\text{setDifference: } [\texttt{PathRef}, \texttt{PathRef}]\} & \mid & p \setminus p' \end{array}$$

where `PathRef` references an array. Then, the union and the difference of R_1 and R_2 can be computed and stored in an array under the key `rel` as follows:

$$\text{rel1} \cup \text{rel2} = \theta_{\text{rel}=(\text{rel1}\cup\text{rel2})} \qquad \text{(union)}$$
$$\text{rel1} \setminus \text{rel2} = \theta_{\text{rel}=(\text{rel1}\setminus\text{rel2})} \qquad \text{(difference)}$$

Now, we develop the notion of MUPG subqueries, which allows us to translate arbitrary relational algebra expressions into a single MUPG query. More precisely, we show that it is possible to combine two MUPG queries into a single MUPG query so that its result (a single tree) contains the results of the original two queries in the form of two arrays. Let $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ be two MUPG queries. We construct an MUPG query $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$ such that the result of evaluating $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$ over a forest F is one tree with two arrays `rel1` and `rel2` such that `rel`i contains the result of evaluating $\boldsymbol{q}_i$ over F. The idea of $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$ is for each document t in F, to create two copies t_1 and t_2 accompanied by an auxiliary key `actDoc`, so that $t_i \models (\text{actDoc} = i)$, for $i = 1, 2$, the copy of t is stored in t_i under the key `doc`i, and later each $\boldsymbol{q}_i$ "affects" only t_i's, and not t_{3-i}'s. Specifically, $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$ is the following MUPG query:

$$\begin{array}{lll} \mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2) = & \theta_{\text{origDoc}=\varepsilon,\ \text{actDoc}=[1,2]} \triangleright \omega_{\text{actDoc}} \triangleright & \text{(duplication)} \\ & \theta^{\text{actDoc}}_{\{\text{doc}i=(\text{actDoc}=i)/\text{origDoc}/\text{dummy}\}_{i=1,2}} \triangleright & \text{(specialization)} \\ & \mathsf{subq}_1(\boldsymbol{q}_1) \triangleright \mathsf{subq}_2(\boldsymbol{q}_2) \triangleright & \text{(queries 1 and 2)} \\ & \gamma^{\mathbf{null}}_{\text{rel1:doc1},\ \text{rel2:doc2}} & \text{(normalization)} \end{array}$$

It consists of 4 logical subqueries: (duplication) creates two copies of each document by introducing an array `actDoc` containing 1 and 2, and then by unwinding it. The original document is stored under the key `origDoc`; (specialization) "specializes" each document, by storing the original document in the proper `doc`i. It is implemented using conditional value definition: e.g., if the value of `actDoc` is 1, then `doc1` is assigned the content of `origDoc` and `doc2` is assigned the content of the non-existing path `dummy`. By using the trick with `dummy` we achieve that in the trees with `actDoc` $= 1$, the path `doc2` does not exist, and the other way around. We refer to this property of the intermediate trees in the pipeline of $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$, the *clean specialization* property; (queries 1 and 2) individually encodes the input queries $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ as $\mathsf{subq}_j(q_j)$, which will be defined below. We note that subq_j are such that the clean specialization property holds after each of $\mathsf{subq}_j(\boldsymbol{q}_j)$; finally, (normalization) simply groups `doc1` and `doc2` in order to store the results of $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ in two arrays `rel1` and `rel2`, as required.

Now, for $\boldsymbol{q}_j = s_1 \triangleright \cdots \triangleright s_n$, $j = 1, 2$, the encoding $\mathsf{subq}_j(\boldsymbol{q}_j)$ is defined as $\mathsf{subq}_j(s_1) \triangleright \cdots \triangleright$

$\mathsf{subq}_j(s_n)$, where subq_j for single stages is defined as follows:

$$\begin{aligned}
\mathsf{subq}_j(\mu_\varphi) &= \mu_{(\text{actDoc}\neq j)\vee\varphi_{[p/\text{doc}j.p]}}\\
\mathsf{subq}_j(\omega_p^n) &= \omega^n_{\text{doc}j.p}\\
\mathsf{subq}_j(\omega_p) &= \mu_{(\text{actDoc}\neq j)\vee((\exists\text{doc}j.p)\wedge(\text{doc}j.p\neq[])\wedge(\text{doc}j.p\neq\mathbf{null}))} \triangleright \omega^n_{\text{doc}j.p}\\
\mathsf{subq}_j(\theta^q_{p=d}) &= \theta^{\text{doc}j.\texttt{_id},\ \text{doc}j.q,\ \text{doc}(3-j),\ \text{actDoc}}_{\text{doc}j.p=(\text{actDoc}=j)/d_{[p'/\text{doc}j.p']}/\text{dummy}}\\
\mathsf{subq}_j(\gamma^{g:g'}_{a:a'}) &= \gamma^{\text{doc}j.g:\text{doc}j.g',\ \text{actDoc}:\text{actDoc}}_{\text{doc}j.a:\text{doc}j.a',\ \text{doc}(3-j):\text{doc}(3-j)} \triangleright\\
&\quad\ \theta^{\text{doc}j.a,\ \text{doc}(3-j)}_{\texttt{_id}:0,\ \text{actDoc}=\texttt{_id}.\text{actDoc},\ \text{doc}j.\texttt{_id}.g=\texttt{_id}.\text{doc}j.g} \triangleright\\
&\quad\ \theta^{\text{actDoc}}_{\{\text{doc}i=(\text{actDoc}=i)/\text{doc}i/\text{dummy}\}_{i=1,2}} \triangleright\\
&\quad\ \omega^n_{\text{doc}(3-j)}
\end{aligned}$$

Here $e_{[p/q]}$ denotes the expression where every occurrence of the path p in the expression e is replaced by the path q. For a fixed j, we call the trees with $\texttt{actDoc} = j$, the ($\boldsymbol{q}_j$'s-)*own* trees, and the trees with $\texttt{actDoc} = (3-j)$, the *other* trees. The encodings of match and unwind stages are quite straightforward. We note that we need to use the unwind operator with the option of preserving nulls and empty arrays, as otherwise the other trees will be lost. The encoding of the project operator $\theta^q_{p=d}$ needs to take care not to lose other the paths $\texttt{doc}j.\texttt{\_id}$ (usually, `_id` is kept by default), $\texttt{doc}(3-j)$ and actDoc. Note that for each pair $p = d$, for a path p and a value definition d, we again use the trick with the non-existing path so as to avoid introducing the path $\texttt{doc}j.p$ in the other trees. The encoding of the group operator is the most involved one, and consists of 4 stages. The first one is grouping with respect to the grouping condition *and* the value of `actDoc`. The result is that all other trees are grouped in one tree and the own trees are spread over multiple trees, one for each value of $\texttt{doc}j.g'$. After this stage the clean specialization property is lost; the second and the third are utilities to rename the paths stored under `_id`, and to make sure that the clean specialization property holds after the forth stage, respectively; and the forth one unwinds $\texttt{doc}(3-j)$ since other trees should not be grouped. We note that for each stage s, the clean specialization property holds after $\mathsf{subq}_j(s)$.

We observe that the subquery mechanism can be easily extended to the case of k subqueries.

Now, let Q be an arbitrary relational algebra query over $\mathsf{sig}_{\mathcal{S}}(C)$. The translation $\mathsf{ra2maq}(Q)$ of Q is defined inductively:

- if $Q = Q_1 \times Q_2$, then $\mathsf{ra2maq}(Q) = \mathsf{pipeline}(\mathsf{ra2maq}(Q_1), \mathsf{ra2maq}(Q_2)) \triangleright \omega_{\text{rel1}} \triangleright \omega_{\text{rel2}}$;
- if $Q = Q_1$ **setop** Q_2, where **setop** $\in \{\setminus, \cup\}$, then

$$\mathsf{ra2maq}(Q) = \mathsf{pipeline}(\mathsf{ra2maq}(Q_1), \mathsf{ra2maq}(Q_2)) \triangleright \theta_{\text{rel}=(\text{rel1 }\mathbf{setop}\text{ rel2})} \triangleright \omega_{\text{rel}} \triangleright \theta_{\{p=\text{rel}.p\}_{p\in\mathsf{sig}(Q_1)}};$$

- if $Q = \sigma_\psi(Q_1)$, then $\mathsf{ra2maq}(Q)$ is defined according to (select);
- if $Q = \pi_S(Q_1)$, then $\mathsf{ra2maq}(Q)$ is defined according to (project);
- if $Q = \rho_{b/a}(Q_1)$, then $\mathsf{ra2maq}(Q) = \mathsf{ra2maq}(\pi_{\mathsf{sig}(Q_1)\setminus\{a\},b/a}(Q_1))$;
- if $Q = \mathsf{sig}_{\mathcal{S}}(C)$, then $\mathsf{ra2maq}(Q)$ is defined according to (flatten);

A.3.3 Translation of Full Relational Algebra to MUPGL

Let C_1 and C_2 be collection names. We show how to compute the cross product between $\mathsf{sig}_{\mathcal{S}}(C_1)$ and $\mathsf{sig}_{\mathcal{S}}(C_2)$.

$$
\begin{aligned}
\mathsf{ra2maq}(\mathsf{sig}_{\mathcal{S}}(C_1) \times \mathsf{sig}_{\mathcal{S}}(C_2)) = {} & \gamma^{\textbf{null}}_{\mathrm{rel1}:\epsilon} \triangleright \lambda^{\mathrm{dummy1}=C_2.\mathrm{dummy2}}_{\mathrm{rel2}} \triangleright && (\mathsf{subrelations2}) \\
& \omega_{\mathrm{rel2}} \triangleright \mathsf{ra2maq}^{\star}(\mathsf{sig}_{\mathcal{S}}(C_2)) \triangleright && (\mathsf{flatten2}) \\
& \omega_{\mathrm{rel1}}
\end{aligned}
$$

where $\mathsf{ra2maq}^{\star}(\mathsf{sig}_{\mathcal{S}}(C_2))$ modifies the query $\mathsf{ra2maq}(\mathsf{sig}_{\mathcal{S}}(C_2))$ by adding superscript `rel1` in the final project, and `dummy1` and `dummy2` are two paths that do not exist in C_1 and C_2, respectively. Here, (subrelations2) is similar to (subrelations) in that it produces one tree that gathers all $\mathsf{sig}_{\mathcal{S}}(C_1)$ tuples in the array `rel1`, and the whole collection for C_2 in `rel2`. Then, (flatten2) performs the preprocessing for the trees in C_2, and finally, by unwinding `rel1` we obtain the required cross-product.

The translation for the arbitrary relational algebra queries is then derived from combining the translation to MUPG queries and this operation. Note that lookup retrieves the collection for C_2 in the form it is stored in the database, so if the relation name $\mathsf{sig}_{\mathcal{S}}(C_2)$ is used multiple times in the input relational algebra query Q, it is convenient to keep the array of $\mathsf{sig}_{\mathcal{S}}(C_2)$-tuples (that is, preprocessed trees from C_2) through the pipeline, instead of performing lookup and then flattening each time $\mathsf{sig}_{\mathcal{S}}(C_2)$ is used.

A.3.4 Proof Theorem 19

▶ **Theorem 19.** *The translation* $\mathsf{ra2maq}$ *is correct. That is, for each relational algebra query* Q *over* $\mathsf{rschema}_{\mathcal{S}}$, $\mathsf{ra2maq}(Q) \equiv_{\mathcal{S}} Q$.

Here, we assume fixed a set $\mathcal{S}$ of type constraints.

▶ **Lemma 30.** *Let* C *be a collection name and* $\mathsf{rschema}_{\mathcal{S}}(C) = \{P_0, \ldots, P_n\}$. *Assume that* $\mathsf{arr}_{\mathcal{S}}(C) = \{p_1, \ldots, p_n\}$ *and* $p_1, \ldots, p_n$ *are ordered in such a way that if* p_i *is a prefix of* p_j, *then* $i < j$. *Then*

$$\omega^n_{p_1} \triangleright \ldots \triangleright \omega^n_{p_n} \triangleright \theta^{\mathsf{sig}_{\mathcal{S}}(C)} \equiv_{\mathcal{S}} \pi_{\mathsf{sig}_{\mathcal{S}}(C)}(P_0 \bowtie \cdots \bowtie P_n).$$

Proof. Consider a MongoDB instance D satisfying $\mathcal{S}$.

($\Rightarrow$) Let $F = D.C \triangleright \omega^n_{p_1} \triangleright \ldots \triangleright \omega^n_{p_n} \triangleright \theta^{\mathsf{sig}_{\mathcal{S}}(C)}$. Since D satisfies $\mathcal{S}$ and the type of each path in $\mathsf{sig}_{\mathcal{S}}(C)$ is literal, no tree in F contains arrays. Assume that $\mathsf{sig}_{\mathcal{S}}(C) = \{a_1, \ldots, a_m\}$, and let $t \in F$. Then $\mathsf{rel}_{\mathsf{sig}_{\mathcal{S}}(C)}(\{t\}) = \{\vec{w}\}$, for $\vec{w} = \{a_1 : v_1, \ldots, a_m : v_m\}$ and $v_i \in V$. First, there is a tree $t_0 \in D.C$ such that the value id of `_id` in t_0 coincides with the value of `_id` in t (the project operator keeps `_id` by default even if it is not included in $\mathsf{sig}_{\mathcal{S}}(C)$). By the semantics of unwind and project it follows that $t \in \big(\{t_0\} \triangleright \omega^n_{p_1} \triangleright \ldots \triangleright \omega^n_{p_n} \triangleright \theta^{\mathsf{sig}_{\mathcal{S}}(C)}\big)$. Second, let $ind_1, \ldots, ind_n$ be the (possibly undefined) indexes associated to t such that there exist trees $t_1, \ldots, t_n$, with $t_n = t$, $t_{j+1} = (t_j \setminus \mathsf{subtree}(t_j, p_{j+1})) \oplus \mathsf{attach}(p_{j+1}, \mathsf{subtree}(t_j, p_{j+1}.ind_{j+1}))$ if $[\![p_{j+1}.0]\!]^{t_j} \neq \emptyset$, and $t_{j+1} = t_j$, ind_{j+1} is undefined if $[\![p_{j+1}.0]\!]^{t_j} = \emptyset$. Note that, since D satisfies $\mathcal{S}$, we have that either p_j is not present in t_0, or the type of p_j in t_0 is array. Moreover, for $p_{j'}$ a strict prefix of p_j (hence, $j' < j$), if ind_j is defined, then also $ind_{j'}$ is defined, and if $ind_{j'}$ is undefined, then also ind_j is undefined.

We show that $\vec{w} \in ans\big(\pi_{\mathsf{sig}_{\mathcal{S}}(C)}(P_0 \bowtie \cdots \bowtie P_n), \{\mathsf{rel}_{P_0}(t_0), \mathsf{rel}_{P_1}(t_0), \ldots, \mathsf{rel}_{P_n}(t_0)\}\big)$, where $\mathsf{rel}_P(t_0)$ denotes $\mathsf{rel}_P(\{t_0\})$. More precisely, we show that for each $j = [0..n]$, there exists a tuple $\vec{w}_j$ in $\mathsf{rel}_{P_j}(t_0)$ with `_id` $: id \in \vec{w}_j$ and $a_i : v_i \in \vec{w}_j$ for $a_i \in P_j \cap \mathsf{sig}_{\mathcal{S}}(C)$, and for $j \geq 1$:

- if ind_j is defined, then $p_{j'}$.index : $ind_{j'} \in \vec{v_j}$ for each prefix $p_{j'}$ of p_j,
- otherwise, let $j_d < j$ be the biggest number such that ind_{j_d} is defined and p_{j_d} is a prefix of p_j, then for each $j' \leq j_d$ such that $p_{j'}$ is a prefix of p_j, we have that $p_{j'}$.index : $ind_{j'} \in \vec{w}_j$, and for each $j_d < j' \leq j$ such that $p_{j'}$ is a prefix of p_j, we have that $p_{j'}$.index : $\mathbf{missing} \in \vec{w}_j$. In this case, we also have that $v_i = \mathbf{missing}$ for each $a_i \in P_j \cap \mathsf{sig}_{\mathcal{S}}(C)$.

Consider the following cases:

- Let $\{a_{i_1}, \ldots, a_{i_k}\} = P_0 \cap \mathsf{sig}_{\mathcal{S}}(C)$, and note that $\mathsf{rel}_{P_0}(t_0)$ consists of the single tuple $\vec{w}_0$. It should be clear that $t_0 \models (a_i = v_i)$ for each $i \in \{i_1, \ldots, i_k\}$. Therefore we conclude that $a_i : v_i \in \vec{w}_0$ for each $i \in \{i_1, \ldots, i_k\}$.
- Let $j \geq 1$, $\{a_{i_1}, \ldots, a_{i_k}\} = P_j \cap \mathsf{sig}_{\mathcal{S}}(C)$, and

$$P_j = \{\texttt{\_id}, p_{j_1}.\text{index}, \ldots, p_{j_\ell}.\text{index}, p_j.\text{index}, p_j.b_{i_1}, \ldots, p_j.b_{i_k}\},$$

where $p_{j_1}, \ldots, p_{j_\ell}$ are the prefixes of p_j sorted by length, and $p_j.b_i = a_i$ for $i \in \{i_1, \ldots, i_k\}$. Note that here, ℓ is the level of nesting of the array under p_j.
Let $\ell = 0$ (i.e., p_j is not nested). Assume that ind_j is defined. Then the tuple $\vec{w}_j = \{\texttt{\_id} : id, p_j.\text{index} : ind_j, a_{i_1} : v_{i_1}, \ldots, a_{i_k} : v_{i_k}\}$ is in $\mathsf{rel}_{P_j}(t_0)$. If ind_j is undefined, then either p_j is an empty array, or does not exist in t_0. In both cases, $\mathsf{rel}_{P_j}(t_0) = \{\{\texttt{\_id} : id,\ p_j.\text{index} : \mathbf{missing},\ a_{i_1} : \mathbf{missing}, \ldots, a_{i_k} : \mathbf{missing}\}\}$.
Let $\ell \geq 1$. Assume that ind_j is defined (hence, so are $ind_{j_1}, \ldots, ind_{j_\ell}$). Then the tuple $\vec{w}_j = \{\texttt{\_id} : id, p_{j_1}.\text{index} : ind_{j_1}, \ldots, p_{j_\ell}.\text{index} : ind_{j_\ell}, p_j.\text{index} : ind_j, a_{i_1}{:}v_{i_1}, \ldots, a_{i_k}{:}v_{i_k}\}$ is in $\mathsf{rel}_{P_j}(t_0)$. If ind_j is undefined, then the paths $a_{i_1}, \ldots, a_{i_k}$ exist neither in t_{j-1}, nor in t, and $v_{i_1} = \cdots = v_{i_k} = \mathbf{missing}$. Let $d \in [1..\ell]$ be the biggest index such that ind_{j_d} is defined. Then we have that $\vec{w}_j = \{\texttt{\_id} : id, p_{j_1}.\text{index} : ind_{j_1}, \ldots, p_{j_d}.\text{index} : ind_{j_d}, p_{j_{d+1}}.\text{index} : \mathbf{missing}, \ldots, p_j.\text{index} : \mathbf{missing}, a_{i_1}{:}\mathbf{missing}, \ldots, a_{i_k}{:}\mathbf{missing}\}$ is in $\mathsf{rel}_{P_j}(t_0)$.
Now, by joining all $\vec{w}_j$ and projecting the indexes away, we obtain exactly $\vec{w}$. We conclude that $\vec{w} \in ans\left(\pi_{\mathsf{sig}_{\mathcal{S}}(C)}(P_0 \bowtie \cdots \bowtie P_n), \{\mathsf{rel}_{P_0}(t_0), \mathsf{rel}_{P_1}(t_0), \ldots, \mathsf{rel}_{P_n}(t_0)\}\right)$.

($\Leftarrow$) Let $R = ans\left(\pi_{\mathsf{sig}_{\mathcal{S}}(C)}(P_0 \bowtie \cdots \bowtie P_n), \{\mathsf{rel}_{P_0}(D.C), \mathsf{rel}_{P_1}(D.C), \ldots, \mathsf{rel}_{P_n}(D.C)\}\right)$, and $\vec{w} \in R$. Then, there exist $\vec{w}_0, \ldots, \vec{w}_n$ such that $\vec{w}_j \in \mathsf{rel}_{P_j}(D.C)$ and $\vec{w} = \pi_{\mathsf{sig}_{\mathcal{S}}(C)}(\vec{w}_0 \bowtie \cdots \bowtie \vec{w}_n)$. Let id be the value of `_id` in $\vec{w}_j$. For $j = [1..n]$, we set ind_j to be the value of p_j.index in $\vec{w}_j$. Then t is defined as t_n, where t_0 is the tree in $D.C$ with the value of `_id` equal id, $t_{j+1} = (t_j \setminus \mathsf{subtree}(t_j, p_{j+1})) \oplus \mathsf{attach}(p_{j+1}, \mathsf{subtree}(t_j, p_{j+1}.ind_{j+1}))$ if ind_{j+1} is distinct from $\mathbf{missing}$ (i.e., $[\![p_{j+1}.0]\!]^{t_j} \neq \emptyset$), and $t_{j+1} = t_j$, if $ind_{j+1} = \mathbf{missing}$ (i.e., $[\![p_{j+1}.0]\!]^{t_j} = \emptyset$). ◀

▶ **Lemma 31.** *The result of* $\mathsf{rel2array}(S_1, S_2)$ *contains* $\pi_{S_i}(\mathsf{sig}_{\mathcal{S}}(C))$ *in* `reli`.

Proof. Follows from the semantics of $\gamma^{\mathbf{null}}_{\ldots}$. ◀

▶ **Lemma 32.** $\mathsf{ra2maq}(S_1 \times S_2) \equiv_{\mathcal{S}} S_1 \times S_2$, where we assume that the output signature of $S_1 \times S_2$ is $\{\text{rel1}.p \mid p \in S_1\} \cup \{\text{rel2}.p \mid p \in S_2\}$.

Proof. Follows from the properties of $\mathsf{rel2array}(S_1, S_2)$ and from the semantics of unwind. ◀

▶ **Lemma 33.** *The result of* $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$ *contains the result of* $\boldsymbol{q}_i$ *in* `reli`.

Proof. Let F be a forest, and F_0 the result of evaluating the subqueries ($\mathsf{duplication}$) and ($\mathsf{specialization}$) over F. It should be clear that each tree in F_0 satisfies the clean specialization property: it follows from the semantics of conditional value definition and of $\theta_{p=p'}$ when p' is missing from the input trees. Moreover, the forest $(F_0 \triangleright \mu_{\text{actDoc}=j})$, for each $j = 1, 2$, coincides with F (up to the prefix $\texttt{doc}j$ and projecting away $\texttt{actDoc}$).

Let $F_1 = F_0 \triangleright \mathsf{subq}_1(\boldsymbol{q}_1)$. We prove that

(clean) F_1 satisfies the clean specialization property,

(own) $(F_1 \triangleright \mu_{\text{actDoc}=1})$, coincides with $F \triangleright \boldsymbol{q}_1$, and

(other) $(F_1 \triangleright \mu_{\text{actDoc}=2})$ coincides with $(F_0 \triangleright \mu_{\text{actDoc}=2})$, which coincides with F (i.e., the "other" trees are not affected).

It is sufficient to prove the above for the case of $\boldsymbol{q}_1$ being a single stage pipeline s. Consider the following cases:

- s is a match stage μ_φ. Then $\mathsf{subq}_1(\boldsymbol{q}_1) = \mu_{(\text{actDoc}\neq 1)\vee\varphi_{[p/\text{doc1}.p]}}$. Since match does not alter the structure of the trees, F_1 satisfies the clean specialization property.
  Let $t \in (F_0 \triangleright \mu_{(\text{actDoc}\neq 1)\vee\varphi_{[p/\text{doc1}.p]}} \triangleright \mu_{(\text{actDoc}=1)})$. Then by the properties of match, it follows that $t \in (F_0 \triangleright \mu_{(\text{actDoc}=1)} \triangleright \mu_{\varphi_{[p/\text{doc1}.p]}})$. By assumption, $(F_0 \triangleright \mu_{(\text{actDoc}=1)})$ coincides with F, therefore we obtain that t is in $F \triangleright \boldsymbol{q}_1$ (up to proper renaming). Similarly, in the other direction, when $t \in (F \triangleright \boldsymbol{q}_1)$, we derive that $t \in (F_1 \triangleright \mu_{(\text{actDoc}=1)})$.
  Since the query $\mu_{(\text{actDoc}\neq 1)\vee\varphi_{[p/\text{doc1}.p]}} \triangleright \mu_{(\text{actDoc}=2)}$ is equivalent to the query $\mu_{(\text{actDoc}=2)}$, we obtain that the forest $(F_0 \triangleright \mu_{(\text{actDoc}\neq 1)\vee\varphi_{[p/\text{doc1}.p]}} \triangleright \mu_{(\text{actDoc}=2)})$ coincides with $(F_0 \triangleright \mu_{(\text{actDoc}=2)})$.
- s is an unwind stage ω^n_p. Then $\mathsf{subq}_1(\boldsymbol{q}_1) = \omega^n_{\text{doc1}.p}$. First, $\mathsf{subq}_1(\boldsymbol{q}_1)$ does not affect the trees with $\text{actDoc} = 2$ because there does not exist the path $\texttt{doc1}.p$, and $\mathsf{subq}_1(\boldsymbol{q}_1)$ will preserve all such trees as they are. Second, the trees that contain the path $\texttt{doc1}.p$ (hence, with $\text{actDoc} = 1$), will be affected in exactly the same way as the trees in F would be affected by $\boldsymbol{q}_1$. Finally, since unwind does not affect other paths than p, we have that F_1 satisfies the clean specialization property.
- s is an unwind stage ω_p. Then $\mathsf{subq}_1(\boldsymbol{q}_1) = \mu_{(\text{actDoc}\neq 1)\vee((\exists\text{doc1}.p)\wedge(\text{doc1}.p\neq[])\wedge(\text{doc1}.p\neq\textbf{null}))} \triangleright \omega^n_{\text{doc1}.p}$. Again, $\mathsf{subq}_1(\boldsymbol{q}_1)$ does not affect the trees with $\text{actDoc} = 2$ because they will all pass the match stage and the subsequent unwind will preserve them as they are. Second, we note that evaluating $\boldsymbol{q}_1$ over F will remove trees where path p does not exist, or p exists and its value is **null**, or empty array. This is done by $\mathsf{subq}_1(\boldsymbol{q}_1)$ in the match stage. The subsequent unwind acts as the unwind above. Again, we have that F_1 satisfies the clean specialization property.
- s is a project stage $\theta^q_{p=d}$. Then, $\mathsf{subq}_1(\boldsymbol{q}_1) = \theta^{\text{doc1}.\texttt{\_id},\ \text{doc1}.q,\ \text{doc2},\ \text{actDoc}}_{\text{doc1}.p=(\text{actDoc}=1)/d_{[p'/\text{doc1}.p']}/\text{dummy}}$. It is easy to see that (clean) and (other) are satisfied. As for (own), the trees with $\text{actDoc} = 1$ will keep the paths doc1.`_id`, doc1.q and the value of the path doc1.p will be defined by d. Hence, (own) also holds.
- s is a group stage $\gamma^{g:g'}_{a:a'}$. Then $\mathsf{subq}_1(\boldsymbol{q}_1) = \gamma^{\text{doc1}.g:\text{doc1}.g',\ \text{actDoc}:\text{actDoc}}_{\text{doc1}.a:\text{doc1}.a',\ \text{doc2}:\text{doc2}} \triangleright$
  $$\theta^{\text{doc1}.a,\ \text{doc2}}_{\text{actDoc}=\texttt{\_id}.\text{actDoc},\ \text{doc1}.\texttt{\_id}.g=\texttt{\_id}.\text{doc1}.g} \triangleright$$
  $$\theta^{\text{actDoc}}_{\{\text{doc}i=(\text{actDoc}=i)/\text{doc}i/\text{dummy}\}_{i=1,2}} \triangleright$$
  $$\omega^n_{\text{doc2}}$$
  The result of the first stage is $n + 1$ trees where
  - one tree originates from all trees with $\text{actDoc} = 2$, the value of doc2 is the array of all such doc2 and doc1.a is an empty array.

- n is the number of different values $v_1, \ldots, v_n$ of doc1.g' in all trees with actDoc $= 1$, and each of the n trees originates from a subset of the trees with actDoc $= 1$ and doc1.$g' = v_i$, the value of doc2 is the empty array, the value of doc1.a is all doc1.a' in this subset of trees, and the value of doc1.g is v_i.

The result of the second stage is $n+1$ trees where some paths in `_id` are renamed. The result of the third stage is a forest satisfying the clean specialization property. In the forth stage, the array doc2 is unwinded, hence the trees with actDoc $= 2$ are brought in the original shape. It is easy to see that all properties are satisfied.

Since the translation is symmetric, we have also that $F_2 = F_1 \triangleright \mathsf{subq}_2(\boldsymbol{q}_2)$ satisfies the corresponding properties (clean), (own) and (other). The final stage of pipeline is a group by null stage that gathers all `doc1` in `rel1` and all `doc2` in `rel2`. Due to (own) and (other) we have that `rel`i contains $F \triangleright \boldsymbol{q}_i$. ◀

A.3.5 Translation of relational algebra to a fragment of MUPG without conditional value definition and standard set operations

Here, we show that is it possible to translate relational algebra (over a single collection) to a fragment of MUPG that does not use the powerful project operators such as conditional value definitions, and set operations such as set union and set difference. In fact, the set union and set difference can be translated by using an approach similar to the translation of SPJ, while the mechanism of subqueries needs a cleaning step to ensure the clean specialization property before the final grouping.

First, we can express set union and set difference without the standard array operators. Below we assume that `rel1` and `rel2` store relations over the signature $P = \{p_1, \ldots, p_n\}$.

$$
\begin{array}{rcl}
\text{rel1} \setminus \text{rel2} & = & \omega_{\text{rel1}} \triangleright \omega_{\text{rel2}} \triangleright \\
& & \theta^{\text{rel1}}_{\text{inIntersection}:\bigwedge_{i=1}^{n}(\text{rel1}.p_i=\text{rel2}.p_i)} \triangleright \\
& & \gamma^{\text{rel1:rel1}}_{\text{inIntersection:inIntersection}} \triangleright \\
& & \mu_{\text{inIntersection}\neq\textbf{true}} \triangleright \\
& & \theta_{\text{rel}=\texttt{\_id}.\text{rel1}} \\
\text{rel1} \cup \text{rel2} & = & \theta_{\text{rel}:[\text{rel1},\text{rel2}]} \triangleright \\
& & \omega_{\text{rel}} \triangleright \omega_{\text{rel}} \triangleright \\
& & \gamma^{\textbf{null}}_{\text{rel:rel}}
\end{array}
$$

Second, we eliminate the conditional value definitions from the encoding of the query $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$. The following query $\mathsf{pipeline}'(\boldsymbol{q}_1, \boldsymbol{q}_2)$ essentially does the same job as the query $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$:

$$
\begin{array}{rll}
\mathsf{pipeline}'(\boldsymbol{q}_1, \boldsymbol{q}_2) = & \theta_{\text{origDoc}=\$\$\text{ROOT},\ \text{actDoc}=[1,2]} \triangleright \omega_{\text{actDoc}} \triangleright & (\mathsf{duplication}) \\
& \theta^{\text{actDoc}}_{\{\text{doc}i=\text{origDoc}\}_{i=1,2}} \triangleright & (\mathsf{specialization'}) \\
& \mathsf{subq}'_1(\boldsymbol{q}_1) \triangleright \mathsf{subq}'_2(\boldsymbol{q}_2) \triangleright & (\mathsf{queries'\ 1\ and\ 2}) \\
& \boldsymbol{q}_{clean} \triangleright & (\mathsf{cleaning}) \\
& \gamma^{\textbf{null}}_{\text{rel1:doc1},\ \text{rel2:doc2}} \triangleright & (\mathsf{normalization}) \\
& \boldsymbol{q}_{null} & (\mathsf{nonull})
\end{array}
$$

With respect to $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$, $\mathsf{pipeline}'(\boldsymbol{q}_1, \boldsymbol{q}_2)$ contains two additional subqueries, (cleaning) and (nonull), immediately before and after (normalization). They are needed because without the conditional value definition, we cannot ensure the clean specialization property. Therefore trees after (queries' 1 and 2) contain "noise" either in the form of `doc1` for $\texttt{actDoc} = 2$ or in the form of `doc2` for $\texttt{actDoc} = 1$ that would consequently end up in $\texttt{rel}i$ if we do not perform any cleaning. Hence, the purpose of (cleaning) is to ensure a weaker version of the clean specialization property, where in the trees with $\texttt{actDoc} = 1$, the value of the path `doc2` is **null**, and symmetrically in the trees with $\texttt{actDoc} = 2$, the value of the path `doc1` is **null**. Then, after the (normalization) subquery, the arrays $\texttt{rel}i$ contain **null**: the purpose of subquery (nonull) is to remove it. The queries $\boldsymbol{q}_{clean}$ and $\boldsymbol{q}_{null}$ are defined as follows:

$$
\begin{aligned}
\boldsymbol{q}_{clean} = {}& \theta^{\mathrm{actDoc}}_{\{\mathrm{doc}i=[\mathbf{null},\mathrm{doc}i]\}_{i=1,2}} \triangleright \omega_{\mathrm{doc1}} \triangleright \omega_{\mathrm{doc2}} \\
& \triangleright \gamma^{\mathbf{null}}_{\mathrm{doc1:doc1,\ doc2:doc2}} \triangleright \omega_{\mathrm{doc1}} \triangleright \omega_{\mathrm{doc2}} \\
& \triangleright \mu_{(\mathrm{doc1.actDoc}=1)\vee(\mathrm{doc1}=\mathbf{null})} \\
& \triangleright \mu_{(\mathrm{doc2.actDoc}=2)\vee(\mathrm{doc2}=\mathbf{null})} \\
\boldsymbol{q}_{null} = {}& \theta^{\mathrm{doc1,doc2}}_{\{\mathrm{doc}i\mathrm{empty}=(\mathrm{doc}i=[\mathbf{null}])\}_{i=1,2}} \triangleright \omega_{\mathrm{doc1}} \triangleright \omega_{\mathrm{doc2}} \\
j = 1,2 \left\{ \begin{array}{l} \triangleright \mu_{(\mathrm{doc}j\neq\mathbf{null})\vee(\mathrm{doc}j\mathrm{empty}=\mathbf{true})} \\ \triangleright \gamma^{\mathrm{doc}(3-j):\mathrm{doc}(3-j),\ \{\mathrm{doc}i\mathrm{empty}:\mathrm{doc}i\mathrm{empty}\}_{i=1,2}}_{\mathrm{doc}j:\mathrm{doc}j} \\ \triangleright \theta^{\mathrm{doc}j}_{\mathrm{doc}(3-j)=\texttt{\_id}.\mathrm{doc}(3-j),\ \{\mathrm{doc}i\mathrm{empty}=\texttt{\_id}.\mathrm{doc}i\mathrm{empty}\}_{i=1,2}} \end{array} \right. & \\
& \triangleright \gamma^{\mathbf{null}}_{\mathrm{doc1:doc1,\ doc2:doc2}}
\end{aligned}
$$

We also need to update the encodings of stages. Thus for match and unwind stages s, $\mathsf{subq}'_j(s) = \mathsf{subq}_j(s)$, and

$$
\begin{aligned}
\mathsf{subq}'_j(\theta^{q}_{p=d}) &= \theta^{\mathrm{doc}j.\texttt{\_id},\ \mathrm{doc}j.q,\ \mathrm{doc}(3-j),\ \mathrm{actDoc}}_{\mathrm{doc}j.p=d_{[p'/\mathrm{doc}j.p']}} \\
\mathsf{subq}'_j(\gamma^{g:g'}_{a:a'}) &= \gamma^{\mathrm{doc}j.g:\mathrm{doc}j.g',\ \mathrm{actDoc:actDoc}}_{\mathrm{doc}j.a:\mathrm{doc}j.a',\ \mathrm{doc}(3-j):\mathrm{doc}(3-j)} \triangleright \\
& \quad \theta^{\mathrm{doc}j.a,\ \mathrm{doc}(3-j)}_{\mathrm{actDoc}=\texttt{\_id}.\mathrm{actDoc},\ \mathrm{doc}j.\texttt{\_id}.g=\texttt{\_id}.\mathrm{doc}j.g} \triangleright \\
& \quad \omega^{n}_{\mathrm{doc}(3-j)}
\end{aligned}
$$

This section should be viewed as a theoretical exercise: clearly, in practice it does not make sense to produce the $\mathsf{pipeline}'$ queries.

A.3.6 Optimizing the translation ra2maq

In this section we develop some optimization techniques for MUPG queries produced by our translation from relational algebra. In fact, such MUPG queries can be very inefficient and may easily cause violations of the limits imposed by MongoDB, basically, on the individual size of the documents (16 MB) and on the size of the intermediate results (100 MB for being kept in-memory).

Our first set of techniques aims at avoiding group by **null** ($\gamma^{\mathbf{null}}_{\dots}$), and results in an optimized translation $\mathsf{ra2maq}^{\star}$. We start by implementing the set union and set difference of queries Q_1 and Q_2 without grouping their results in two arrays in advance. Instead, we can keep the corresponding documents separately, and proceed as follows. Denote by $\mathsf{pipeline}^{\star}(\boldsymbol{q}_1, \boldsymbol{q}_2)$ the query $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$ without the last stage. Assume that $\mathsf{att}(\mathsf{sig}(Q_1)) =$

$\{p_1, \dots, p_n\}$, then:

$$
\begin{array}{rcl}
\mathsf{ra2maq}^\star(Q_1 \setminus Q_2) & = & \mathsf{pipeline}^\star(\mathsf{ra2maq}(Q_1), \mathsf{ra2maq}(Q_2)) \triangleright \\
 & & \theta^{\text{doc1, doc2}}_{\{\text{p}i=(\text{actDoc}=1)/\text{doc1.p}i/\text{doc2.p}i\}_{i=1}^{n}} \triangleright \\
 & & \gamma^{\text{p1:p1},\dots,\text{p}n:\text{p}n}_{\text{doc2s:doc2}} \triangleright \\
 & & \mu_{\text{doc2s}=[]} \triangleright \\
 & & \theta_{\{\text{p}i=\texttt{\_id}.\text{p}i\}_{i=1}^{n},\ \texttt{\_id}:\textbf{false}} \\
\mathsf{ra2maq}^\star(Q_1 \cup Q_2) & = & \mathsf{pipeline}^\star(\mathsf{ra2maq}(Q_1), \mathsf{ra2maq}(Q_2)) \triangleright \\
 & & \theta^{\text{doc1, doc2}}_{\{\text{p}i=(\text{actDoc}=1)/\text{doc1.p}i/\text{doc2.p}i\}_{i=1}^{n}} \triangleright \\
 & & \gamma^{\text{p1:p1},\dots,\text{p}n:\text{p}n} \triangleright \\
 & & \theta_{\{\text{p}i=\texttt{\_id}.\text{p}i\}_{i=1}^{n},\ \texttt{\_id}:\textbf{false}}
\end{array}
$$

Next, for cross-document joins grouping is necessary, moreover is some cases it might be that grouping by **null** is unavoidable. But in most of the practical cases, and specifically in the case of natural join, we can use some of the joining condition as the grouping condition. Let $Q_1 \bowtie_\varphi Q_2$, where for simplicity we assume that φ is a conjunction of equalities and inequalities between paths $p_i \in \mathsf{att}(\mathsf{sig}(Q_1))$ and $q_i \in \mathsf{att}(\mathsf{sig}(Q_2))$. (Here, equalities and inequalities between paths and constants are not considered to be real joining conditions. For their optimization, see below.) Let $\{p_1 = q_1, \dots, p_n = q_n\}$ be the set of equalities in φ, and NE the set of inequalities in φ. Then

$$
\begin{array}{rcl}
\mathsf{ra2maq}^\star(Q_1 \bowtie_\varphi Q_2) & = & \mathsf{pipeline}^\star(\mathsf{ra2maq}(Q_1), \mathsf{ra2maq}(Q_2)) \triangleright \\
 & & \theta^{\text{doc1, doc2}}_{\{\text{g}i=(\text{actDoc}=1)/\text{doc1.p}i/\text{doc2.q}i\}_{i=1}^{n}} \triangleright \\
 & & \gamma^{\text{g1:g1},\dots,\text{g}n:\text{g}n}_{\text{doc1:doc1, doc2:doc2}} \triangleright \\
 & & \omega_{\text{doc1}} \triangleright \omega_{\text{doc2}} \triangleright \\
 & & \theta_{\text{toJoin}=\bigwedge_{p \neq q \in NE} \neq(\text{doc1.}p,\text{doc2.}q)} \triangleright \mu_{\text{toJoin}=\textbf{true}}
\end{array}
$$

If the set of equalities in φ is empty, then we have to group by **null**.

Second, a fundamental property of our translation $\mathsf{ra2maq}$ is that it does not require any normal form for the inner-collection queries, and within the scope of subq it implements each RA construct as soon as it appears. For instance, for $Q = \pi_S(\sigma_{\psi_1}(R_1) \bowtie \sigma_{\psi_2}(R_2))$, the query $\mathsf{ra2maq}(Q)$ will first filter R_1 and R_2, and only then will join them. Therefore, knowing the statistics of the data, one can already optimize input RA queries using the existing RA optimization and planning techniques. As a result, the output MAQ will be also optimized to some degree. Then, we can optimize it further as follows. Due to the linear structure of the pipeline, even if subq_2 starts with a selective match stage, it will not be applied before subq_1 is finished, hence during subq_1 all the unfiltered documents of subq_2 will pollute the pipeline. Fortunately, the stages of subq_1 and subq_2 can be executed in an interleaved manner, so we can reorder them to make sure that the selective stages come first.

As for cross-collection joins, it is not possible to apply pre-filtering before the join. Hence, in such cases we have to assume some kind of normal form, and we cannot assure the best possible execution.

Finally, we propose some techniques that allow one to reduce the size of the intermediate results (the number of trees and their individual size) and the number of stages for arbitrary MAQs.

Array unwinding is an expensive operation that may produce large intermediate results due to the multiple copies of original trees (modulo the unwinded array) it creates.

Consecutive unwindings can even result in intermediate results of exponential size. When unwind is followed by a match stage, we can use the *filter* array operator to reduce the size of the array before unwinding. This operator $f_d(p)$ filters out the values inside an array p that do not satisfy d, and can be used in value definition:

{\$filter: { input: `PathRef`, as: `Path`, cond: `ValueDef` }} $d ::= \quad f_d(p)$

Below we assume that p is a path of type `array`, P is the set of paths we are interested in from the documents, and φ is a condition on p that we view both as a criterion and as a (Boolean) value definition. Unwind followed by match can be optimized as follows:

$$\omega_p^{(n)} \triangleright \mu_\varphi \equiv \theta^P_{p=f_\varphi(p)} \triangleright \omega_p^{(n)}$$

Using indexes As most databases, MongoDB provides primary (on `_id`) and secondary (on user-defined set of paths) indexing capabilities. Given that the match operation takes advantage of the indexes at the initial stage only, it is generally valuable to start the pipeline with a pre-filtering match stage to reduce the number of trees.

Early filtering A standard technique from RA is applying select as soon as possible to reduce the number of tuples in the intermediate relations. Here we can do the same by applying match as early as possible.

Other techniques that extend those that already exist in MongoDB[5]. The MongoDB engine already provides some optimization techniques for coalescing some stages, such as two consecutive match stages and a lookup-unwind sequence. To complement these techniques, we observe that:

1. Two consecutive project stages can also be coalesced by converting the first operation into a substitution and applying it to the second;
2. A project-match sequence can be replaced by a match stage when the project stage is only used for evaluating a variable-to-constant expression.

A.3.6.1 Evaluation

To show effectiveness of our techniques, we design an experiment based on translated queries from RA to MUPG. We created an extension of the `bios` collection that covers other awards (scientific, show business, humanitarian, etc.). This new collection contains 1287 documents and is called `awards1287`.

Let Q_1 be a RA query that retrieves all the pairs of persons that received the same award in the same year and where one of them is born before 1940:

$$\begin{aligned}
Q_1 &= \pi_{\mathrm{fn1,fn2,ln1,ln2,bd1,an1,ay1}}(\sigma_{\mathrm{bd1<1940}}(Q_{a1} \bowtie_{\mathrm{an1=an2 \wedge ay1=ay2 \wedge (fn1 \neq fn2 \vee ln1 \neq ln2)}} Q_{b1}))\\
Q_{a1} &= \pi_{\mathrm{fn1/name.first,ln1/name.last,an1/awards.award,ay1/awards.year,bd1/birth}}(\mathsf{sig}_{\mathcal{S}_b}(\texttt{awards1287}))\\
Q_{b1} &= \pi_{\mathrm{fn2/name.first,ln2/name.last,an2/awards.award,ay2/awards.year}}(\mathsf{sig}_{\mathcal{S}_b}(\texttt{awards1287}))
\end{aligned}$$

Let $Q_1^\star$ be an optimized version of Q_1 where the filter operation $\sigma_{\mathrm{bd1<1940}}$ is moved inside the subquery $Q_{a1}^\star$:

$$\begin{aligned}
Q_1^\star &= \pi_{\mathrm{fn1,fn2,ln1,ln2,bd1,an1,ay1}}(Q_{a1}^\star \bowtie_{\mathrm{an1=an2 \wedge ay1=ay2 \wedge (fn1 \neq fn2 \vee ln1 \neq ln2)}} Q_{b1})\\
Q_{a1}^\star &= \pi_{\mathrm{name.first/fn1,\ldots,birth/bd1}}(\sigma_{\mathrm{birth<1940}}(\mathsf{sig}_{\mathcal{S}_b}(\texttt{awards1287})))
\end{aligned}$$

[5] `https://docs.mongodb.org/manual/core/aggregation-pipeline-optimization/`

Let Q_2 be a RA query that retrieves all the awards received after 1999 and their recipients:

$$\begin{aligned}
Q_2 &= Q_{a2} \setminus Q_{b2}\\
Q_{a2} &= \pi_{\text{fn/name.first,ln/name.last,an/awards.award,ay/awards.year}}(\mathsf{sig}_{\mathcal{S}}(\texttt{awards1287}))\\
Q_{b2} &= \pi_{\text{fn/name.first,ln/name.last,an/awards.award,ay/awards.year}}(\sigma_{\text{awards.year}<2000}(\mathsf{sig}_{\mathcal{S}_b}(\texttt{awards1287})))
\end{aligned}$$

Let Q_3 be a RA query that retrieves all the awards and their recipients:

$$\begin{aligned}
Q_3 &= Q_{a3} \cup Q_{b3}\\
Q_{a3} &= \pi_{\text{fn/name.first,ln/name.last,an/awards.award,ay/awards.year}}(\sigma_{\text{awards.year}<2000}(\mathsf{sig}_{\mathcal{S}_b}(\texttt{awards1287})))\\
Q_{b3} &= \pi_{\text{fn/name.first,ln/name.last,an/awards.award,ay/awards.year}}(\sigma_{\text{awards.year}\geq 2000}(\mathsf{sig}_{\mathcal{S}_b}(\texttt{awards1287})))
\end{aligned}$$

We evaluated the execution times *(i)* of the ra2maq and ra2maq* translations of the queries Q_1, $Q_1^\star$, Q_2 and Q_3, and *(ii)* of the translations of Q_1 and $Q_1^\star$ to which we applied all the applicable optimization techniques mentioned above except early filtering. We run these translated queries 5 times on MongoDB 3.2.1 on a MacBookPro 8.1 having an SSD hard-drive and 8GB of RAM, and obtained the following results (standard deviation is given in parentheses):

Query	ra2maq	ra2maq*	All optimizations
$Q_1/Q_1^\star$	6.6 s (0.06 s)/4.4 s (0.07 s)	85 ms (2 ms)/76 ms (3 ms)	60 ms (3 ms)/57 ms (3 ms)
Q_2	52 ms (4 ms)	62 ms (3 ms)	-
Q_3	66 ms (3 ms)	72 ms (4 ms)	-

In the case of the translations of Q_1 and $Q_1^\star$, we observe that the optimization ra2maq* strongly improves the performance of the cross-document join performed by these queries. The impact of the early filtering technique introduced by $Q_1^\star$ is significant on the ra2maq translation but limited on the ra2maq* translation. The reduction of the number of stages performed by the additional optimization techniques has also a beneficial impact on the translations of Q_1 and $Q_1^\star$. Regarding the translations of Q_2 and Q_3, we observe that ra2maq* introduces a negligible additional cost compared to the ra2maq translation, which makes use of the standard MongoDB set union and set difference operators on arrays. This result favors the use of ra2maq* for handling minus and union operations because it better respects the maximum document size limitation imposed by MongoDB. By contrast, the ra2maq translation groups all results of the subqueries in one document, and may thus not be executable on larger datasets.

The exact queries and the `awards1287` collection can be found at `https://github.com/ontop/ontop-examples/tree/master/icdt-17`.

A.4 Proofs in Section 6

► **Lemma 21.** *Boolean query evaluation for* MFQ *queries is in* LOGSPACE *in combined complexity.*

Proof. Let D be a MongoDB database, and $\boldsymbol{q}$ an MFQ. Without loss of generality, we may assume that $\boldsymbol{q}$ is of the form $C \triangleright \mu_\varphi$, where φ is a criterion. We can view φ as a Boolean formula constructed using the connectors $\wedge$, $\vee$ and $\neg$ starting from the atoms of the form $(p\ \mathbf{op}\ v)$ and $\exists p$, where p is a path, v a literal value, and $\mathbf{op}$ is a comparison operator. Given a document d, we can construct the corresponding tree t in LOGSPACE. Then, given a tree

t and an atom α of the above form, we can check in LogSpace whether $t \models \alpha$: for each node x in t, we can check in LogSpace if $\mathsf{path}(x, t) = p$ and we can check in LogSpace if $L_n(x) = v$, or $L_n(x) =$ '[ ]' and for some child y of x $L_n(y) = v$.

Now, we define a LogSpace reduction from the problem of whether $ans_{\mathsf{mo}}(\boldsymbol{q}, D) \neq \emptyset$ to the problem of determining the truth value of a variable-free Boolean formula, known to be ALogTime-complete [4]. We construct a Boolean formula ψ as the disjunction of φ_t for each $t \in D.C$, where φ_t is a copy of φ, where each atom α is substituted with 1 if $t \models \alpha$ and with 0, otherwise. Then $ans_{\mathsf{mo}}(\boldsymbol{q}, D) \neq \emptyset$ iff the value of ψ is true. ◀

It is straightforward to show that Boolean query evaluation for MFQ queries in ALogTime-hard: for a given Boolean formula ψ, we construct a criterion φ by substituting in ψ each occurrence of 1 with (p1 = 1) and each occurrence of 0 with (p0 = 0); $\boldsymbol{q}$ is then the query $C \triangleright \mu_\varphi$, and the collection for C contains one document `{"p1": 1, "p0": 0}`. We leave it open whether MFQs are ALogTime-complete.

Next, we show that MU queries lose tractability.

▶ **Lemma 22.** *Boolean query evaluation for* MU *queries is* NP*-complete in combined complexity.*

Proof. We prove the lower bound by reduction from the Boolean satisfiability problem. Let φ be a Boolean formula over n variables x1, ..., xn. We fix a collection name C, and construct a collection F for C and an MU query $\boldsymbol{q}$ such that $ans_{\mathsf{mo}}(\boldsymbol{q}, F)$ is non-empty iff φ is satisfiable.

F contains a single document d of the form `{"x1": [true,false], ..., "xn": [true,false]}`, and $\boldsymbol{q}$ is the query: $C \triangleright \omega_{\mathrm{x1}} \triangleright \ldots \triangleright \omega_{\mathrm{xn}} \triangleright \mu_\varphi$, denoted $\boldsymbol{q}_{\mathrm{NP}}$, where φ can be viewed as a criterion.

For the upper bound we provide an NP algorithm. Let $\boldsymbol{q}$ be an MU query over a collection name C, and D a database instance. For each tree $t \in D.C$ we proceed as follows. For each match stage μ_φ in $\boldsymbol{q}$, for the paths $p_1, \ldots, p_n$ that appear in φ and are used for unwinding in the preceding stages, we guess the elements $v_1, \ldots, v_n$ in the corresponding arrays in t, and then check whether φ is satisfied. If yes, then we proceed to the next match stage until we reach the last one, and if it is successful, then t is in the answer. Otherwise, t in not in the answer. If at least one tree is in the answer, then $ans_{\mathsf{mo}}(\boldsymbol{q}, D)$ is non-empty. ◀

▶ **Corollary 34.** *The query emptiness problem for* MUP *queries is* NP*-hard in query complexity.*

Proof. Since it is possible to use project to create copies of arrays, we can modify the above reduction so that F contains a single document of the form `{"values": [true,false]}`, and $\boldsymbol{q} = C \triangleright \theta_{\mathrm{x1=values},\ \ldots,\ \mathrm{x}n\mathrm{=values}} \triangleright \boldsymbol{q}_{\mathrm{NP}}$. ◀

▶ **Corollary 35.** *The query emptiness problem for* MUL *queries is* NP*-hard in query complexity.*

Proof. Now, we can use lookup to create copies of arrays. In this case again, F contains two documents of the form `{"values": true}` and `{"values": false}`. The query is as follows: $\boldsymbol{q} = C \triangleright \lambda_{\mathrm{x1}}^{\mathrm{dummy}=C.\mathrm{dummy}} \triangleright \cdots \triangleright \lambda_{\mathrm{x}n}^{\mathrm{dummy}=C.\mathrm{dummy}} \triangleright \omega_{\mathrm{x}n} \triangleright \cdots \triangleright \omega_{\mathrm{x}n} \triangleright \mu_{\varphi'}$, where φ' is the variant of φ where each variable x is replaced by x.value. ◀

It follows from the translation from relational algebra to MUPG that MUPG queries are PSpace-hard in combined complexity. The translation however uses quite powerful project operators such as conditional value definition, or introducing new arrays. Here, we show that

MUPG queries are hard even for very restricted use of project, namely when project is used to compute Boolean value definitions.

▶ **Lemma 36.** *The query emptiness problem for* MUPG *queries is* PSPACE*-hard in combined complexity, even for project used only for Boolean value definitions.*

Proof. Proof by reduction from the validity problem of QBF. Let φ be a quantified Boolean formula over the variables $x_1, \ldots, x_n$ of the form $\mathsf{Q}_1 x_1 \mathsf{Q}_2 x_2 \ldots \mathsf{Q}_n x_n.\psi$, for $\mathsf{Q}_i \in \{\exists, \forall\}$. We construct a collection C and an MUPG query $\boldsymbol{q}$ such that $C \triangleright \boldsymbol{q}$ is non-empty iff φ is valid.

The collection C is the same as in the proof of Lemma 22. The query $\boldsymbol{q}$ can be seen as an extension of the query $\boldsymbol{q}_{\mathrm{NP}}$:

$$\boldsymbol{q} = \omega_{\mathrm{x1}} \triangleright \ldots \triangleright \omega_{\mathrm{x}n} \triangleright \theta^{\mathrm{x1},\ldots,\mathrm{x}n}_{\mathrm{phi}=\psi} \triangleright \qquad (\mathrm{s}_0)$$

$$\gamma^{\mathrm{x1{:}x1},\ \ldots,\ \mathrm{x}(n-1){:}\mathrm{x}(n-1)}_{\mathrm{values{:}phi}} \triangleright \theta_{\mathrm{phi}=qua_n(\mathrm{values})} \triangleright \qquad (\mathrm{s}_1)$$

$$\gamma^{\mathrm{x1{:}\_id.x1},\ \ldots,\ \mathrm{x}(n-2){:}\mathrm{\_id.x}(n-2)}_{\mathrm{values{:}phi}} \triangleright \theta_{\mathrm{phi}=qua_{n-1}(\mathrm{values})} \triangleright \qquad (\mathrm{s}_2)$$

$$\ldots$$

$$\gamma^{\mathrm{x1{:}\_id.x1}}_{\mathrm{values{:}phi}} \triangleright \theta_{\mathrm{phi}=qua_2(\mathrm{values})} \triangleright \qquad (\mathrm{s}_{n-1})$$

$$\gamma^{\mathbf{null}}_{\mathrm{values{:}phi}} \triangleright \theta_{\mathrm{phi}=qua_1(\mathrm{values})} \triangleright \qquad (\mathrm{s}_n)$$

$$\mu_{\mathrm{phi}=\mathbf{true}} \qquad (\mathrm{s}_{n+1})$$

where $qua_i(\mathrm{values})$ is the expression $(\mathrm{values} = [\mathbf{true}, \mathbf{true}])$ if Q_i is $\forall$ and the expression $(\mathrm{values} \neq [\mathbf{false}, \mathbf{false}])$ if Q_i is $\exists$.

The query $\boldsymbol{q}$ consists of $n+2$ subqueries $\mathrm{s}_0, \ldots, \mathrm{s}_{n+1}$. The first subquery s_0 creates all possible assignments to the variables $x_1, \ldots, x_n$ and then computes the value of ψ under each such assignment and stores it under the key `phi`. The subqueries s_i, for $i = 1, \ldots, n$, compute the value of the formula $\mathsf{Q}_{n-i+1} x_{n-i+1} \ldots \mathsf{Q}_n x_n.\psi$, by proper grouping, and then analyzing according Q_i the array `values` containing two Boolean values. Observe that after s_n the pipeline contains a single document. Finally, the subquery s_{n+1} checks if the value of `phi`, containing the value of φ, is true in that single document. ◀

Next, we modify the above reduction so as to avoid using project. We use match instead of Boolean value definitions.

▶ **Lemma 23.** *The query emptiness problem for* MUG *queries is* PSPACE*-hard in combined complexity.*

Proof. Proof by reduction from the validity problem of QBF. Let φ be a quantified Boolean formula over the variables $x_1, \ldots, x_n$ of the form $\mathsf{Q}_1 x_1 \mathsf{Q}_2 x_2 \ldots \mathsf{Q}_n x_n.\psi$, for $\mathsf{Q}_i \in \{\exists, \forall\}$. We construct a collection C and an MUPG query $\boldsymbol{q}$ such that $C \triangleright \boldsymbol{q}$ is non-empty iff φ is valid.

The collection C is the same as in the proof of Lemma 22, and $\boldsymbol{q}$ is as follows:

$$\boldsymbol{q} = \omega_{\mathrm{x1}} \triangleright \ldots \triangleright \omega_{\mathrm{x}n} \triangleright \mu_{\psi} \triangleright \qquad (\mathrm{s}_0)$$

$$\gamma^{\mathrm{x1{:}x1},\ \ldots,\ \mathrm{x}(n-1){:}\mathrm{x}(n-1)}_{\mathrm{values{:}x}n} \triangleright \mu_{qua_n(\mathrm{values})} \triangleright \qquad (\mathrm{s}_1)$$

$$\gamma^{\mathrm{x1{:}\_id.x1},\ \ldots,\ \mathrm{x}(n-2){:}\mathrm{\_id.x}(n-2)}_{\mathrm{values{:}x}n-1} \triangleright \mu_{qua_{n-1}(\mathrm{values})} \triangleright \qquad (\mathrm{s}_2)$$

$$\ldots$$

$$\gamma^{\mathrm{x1{:}\_id.x1}}_{\mathrm{values{:}x2}} \triangleright \mu_{qua_2(\mathrm{values})} \triangleright \qquad (\mathrm{s}_{n-1})$$

$$\gamma^{\mathbf{null}}_{\mathrm{values{:}x1}} \triangleright \mu_{qua_1(\mathrm{values})} \qquad (\mathrm{s}_n)$$

where $qua_i(\text{values})$ is the expression $(\text{values} = [0,1]) \vee (\text{values} = [1,0])$ if Q_i is $\forall$ and the expression $(\text{values} \neq [])$ if Q_i is $\exists$. ◀

We can identify the following sources of complexity. First, the unwind operator allows us to generate an exponential number of trees in the pipeline. Second, the project and the group operators allow us to create new objects and arrays by duplicating the existing ones. Hence, they can be used to create trees of exponential size (in the size of the query).

Next, we show that evaluation of MP queries with additional array operators *filter* and *map* is NP-hard in query complexity. The map operator $m_d(p)$ allows to transform each element inside an array p according to the new definition d:

{$map: { input: `PathRef`, as: `Path`, in: `ValueDef` }} $\quad d ::= \quad m_d(p)$

▶ **Lemma 24.** *The query emptiness problem for* MP *queries with filter and map operators is* NP*-hard in combined complexity.*

Proof. Proof by reduction from the Boolean satisfiability problem. Let φ be a Boolean formula over n variables x1, …, xn. We construct a query $\boldsymbol{q}$ such that for each non-empty forest F, $F \triangleright \boldsymbol{q}$ is non-empty iff φ is satisfiable.

$$\boldsymbol{q} = \theta_{\text{a0}=\{\text{x1}=0\},\ \text{a1}=\{\text{x1}=1\}} \triangleright \theta_{\text{a}=[\text{a0},\text{a1}]} \triangleright \qquad (\text{a}_1)$$

$$\theta_{\text{a0}=m_{\{\text{x1}=\text{a.x1},\ \text{x2}=0\}}(a),\ \text{a1}=m_{\{\text{x1}=\text{a.x1},\ \text{x2}=1\}}(a)} \triangleright \theta_{\text{a}=(\text{a0}\cup\text{a1})} \triangleright \qquad (\text{a}_2)$$

$$\ldots$$

$$\theta_{\text{a0}=m_{\{\text{x1}=\text{a.x1},\ldots,\text{x}(n-1)=\text{a.x}(n-1),\ \text{x}n=0\}}(\text{a}),\ \text{a1}=m_{\{\text{x1}=\text{a.x1},\ldots,\text{x}(n-1)=\text{a.x}(n-1),\ \text{x}n=1\}}(\text{a})} \triangleright \theta_{\text{a}=(\text{a0}\cup\text{a1})} \triangleright \qquad (\text{a}_n)$$

$$\theta_{\text{assignments}=f_\varphi(\text{a})} \triangleright \qquad (\text{filter})$$

$$\mu_{\text{assignments}\neq[]}$$

The stages (a_1) to (a_n) construct an array `a` of 2^n elements, where each element is an object encoding an assignment to the variables x1, …, xn. In the stage (a_i), the map operator is used to extend each current element with the an assignment to the variable xi. The (filter) stage then uses the filter operator to check for each element of the big array, whether it is a satisfying assignment, and if not, it is removed from the array. Finally, match will check that the resulting array is non-empty. If it is the case, then we have a satisfying assignment. All satisfying assignments will be stored in `a`. An actual query encoding the translation can be found in Section B.4.3. ◀

On the other hand, if we restrict the project operator so as to disallow duplication of existing paths (and hence to disallow creation of exponentially large arrays or objects), and similarly with the group operator (in principle, value duplication can also be done by group), then the size of the trees can grow only polynomially in the size of the query. Such restricted MPG queries are denoted MPG$^-$.

▶ **Lemma 25.** *The query emptiness problem for* MPG$^-$ *queries is* PTIME*-complete in combined complexity.*

Proof. First, we show the PTIME upper bound. Let F be a forest. Consider the following cases:

- $\boldsymbol{q} = \mu_\varphi$. It is clear that the result of $F \triangleright \boldsymbol{q}$ can be computed in PTime: for each $t \in F$, we check whether t satisfies φ: if it satisfies, then $t \in (F \triangleright \boldsymbol{q})$, otherwise it is not. The check $t \models \varphi$ can be done in polynomial time in the size of φ and t. The number of trees in the output is bounded by the number of trees in F: $(F \triangleright \boldsymbol{q}) \subseteq F$.
- $\boldsymbol{q} = \gamma^{g'_1:g_1,\ldots,g'_n:g_n}_{a'_1:a_1,\ldots,a'_m:a_m}$. Assume that $(F \triangleright q) = \{t_1, \ldots, t_k\}$.
  Then each t_i corresponds to a subset F_i of F such that $F_i \cap F_j = \emptyset$ for $i \neq j$. We have that $|t_i| \leq |F_i|$.
  It is clear that $F \triangleright \boldsymbol{q}$ can be computed in PTime in the size of F and $\boldsymbol{q}$, and the result is linear in the size of $\boldsymbol{q}$.
- $\boldsymbol{q} = \theta^{p_1,\ldots,p_m}_{p'_1=d_1,\ldots,p'_n=d_n}$. Then the number of trees in $F \triangleright \boldsymbol{q}$ is equal to the number of trees in F, and each tree $t \in F$ gives rise to a tree $t' \in (F \triangleright \boldsymbol{q})$, and the size of t' is linear in $m + n$ and polynomial in the size of t.

Now, let $\boldsymbol{q}$ be an arbitrary MPG$^-$ query. Then the number of trees in $F \triangleright \boldsymbol{q}$ is less than or equal to the number of trees in F, and each tree is polynomially large in the size of F and $\boldsymbol{q}$.

The PTime-lower bound is a straightforward reduction from the Circuit Value problem, known to be PTime-complete.For completeness, we provide the reduction. Given a monotone Boolean circuit $\mathcal{C}$ consisting of a finite set of assignments to Boolean variables $X_1, \ldots, X_n$ of the form $X_i = 0$, $X_i = 1$, $X_i = X_j \wedge X_k$, $j, k < i$, or $X_i = X_j \vee X_k$, $j, k < i$, where each X_i appears on the left-hand side of exactly one assignment, check whether the value X_n is 1 in $\mathcal{C}$.

We construct a query $\boldsymbol{q}$ such that on each non-empty forest F, $F \triangleright \boldsymbol{q}$ is non-empty iff the value X_n is 1 in $\mathcal{C}$. We set $\boldsymbol{q} = s_1 \triangleright \cdots \triangleright s_n \triangleright \mu_{\mathrm{x}n=1}$, where for $i = [1..n]$, $s_i = \theta^{\{\mathrm{x}j\}_{j=[1..i-1]}}_{\mathrm{x}i=ass_i}$, where $ass_i = b$, if $X_i = b$ for $b \in \{0, 1\}$, $ass_i = \mathrm{x}j \wedge \mathrm{x}k$, if $X_i = X_j \wedge X_k$, and $ass_i = \mathrm{x}j \vee \mathrm{x}k$, if $X_i = X_j \vee X_k$. ◀

B Appendix: Examples of MongoDB queries and low level details

B.1 Semantics of Comparisons

B.1.1 Comparison operators for null [6]

$\mathbf{null} = \mathbf{null}$	true	$\mathbf{null} \neq \mathbf{null}$	false	$\mathbf{null} < \mathbf{null}$	false	$\mathbf{null} > \mathbf{null}$	false
$\mathbf{null} = 5$	false	$\mathbf{null} \neq 5$	true	$\mathbf{null} < 5$	false	$\mathbf{null} > 5$	false
$\mathbf{null} = \infty$	false	$\mathbf{null} \neq \infty$	true	$\mathbf{null} < \infty$	false	$\mathbf{null} > \infty$	false
$\mathbf{null} = -5$	false	$\mathbf{null} \neq -5$	true	$\mathbf{null} < -5$	false	$\mathbf{null} > -5$	false
$\mathbf{null} = -\infty$	false	$\mathbf{null} \neq -\infty$	true	$\mathbf{null} < -\infty$	false	$\mathbf{null} > -\infty$	false
$\mathbf{null} = \mathbf{true}$	false	$\mathbf{null} \neq \mathbf{true}$	true	$\mathbf{null} < \mathbf{true}$	false	$\mathbf{null} > \mathbf{true}$	false
$\mathbf{null} = \mathbf{false}$	false	$\mathbf{null} \neq \mathbf{false}$	true	$\mathbf{null} < \mathbf{false}$	false	$\mathbf{null} > \mathbf{false}$	false

B.1.2 Comparison for objects

{abc: 3} < {abc: 4}	true
{abc: 3} < {abc: 4, def: 5}	true
{abc: 3} < {def: 5, abc: 4}	true
{abc: 3, def: 5} < {abc: 4}	true
{def: 5, abc: 3} < {abc: 4}	false
{abc: 3} < {abc: 2}	false
{abc: 3} < {abc: 2, def: 5}	false
{abc: 3, def: 5} < {abc: 2}	false

B.2 Syntax and Semantics Particulars

B.2.1 Difference between criterion and Boolean value definition

```
db.arrays_boolean = [
  { "_id" : 1, "arr" : [ true, true, true, true ] },
  { "_id" : 2, "arr" : [ true, true, false, true ] },
  { "_id" : 3, "arr" : [ false, true, false, false ] },
  { "_id" : 4, "arr" : [ false, false, false, false ] }
]

db.arrays_boolean.aggregate([
    // this match is different from the project below:
    {$match: {"arr": {$ne: false}}},
    //
    {$project: {"isModel": {$ne: ["$arr", false]} }}
])
```

B.2.2 Checking if a path exists in Boolean value definition

```
db.test_path_exists = [
  { "_id" : 1, "p" : null },
  { "_id" : 2, "p" : false },
  { "_id" : 3, "p" : 0 },
  { "_id" : 4 },
  { "_id" : 5, "p" : "abc" }
]

db.test_path_exists.aggregate([
    {$project: {"ptrue": {$cond: {
        if: "$p", then: true, else: false
      }}}}
```

[6] Actually, in comparisons done within **$project** (but not within **$match**), MongoDB considers **null** to be smaller than any other value. Since there is no rationale for this, we consider this as a bug.

```
])

"result" : [
  { "_id" : 1, "ptrue" : false },
  { "_id" : 2, "ptrue" : false },
  { "_id" : 3, "ptrue" : false },
  { "_id" : 4, "ptrue" : false },
  { "_id" : 5, "ptrue" : true }
]
```

```
db.test_path_exists.aggregate([
    {$project: {"pexists": {$cond: {
        if: "$p", then: true,
        else: {$cond: {
            if: {$or: [{$eq: ["$p", null]}, {$eq: ["$p", false]}, {$eq: ["$p", 0]}]},
            then: true,
            else: false
            }}}}}}
])
```

B.2.3 ifNull can be expressed as conditional value definition

The following two are equivalent:

```
db.test_path_exists.aggregate ([
    {$project : {
        "pathIsNull" : {$cond: {
            if: "$p",
            then: "$p",
            else: {$cond: {
                if: {$or: [{$eq: ["$p", false]}, {$eq: ["$p", 0]}]},
                then: "$p",
                else: true}}
            }},
        "p": 1
    }},
])
```

```
[
    {$project : {
        "pathIsNull": {$ifNull: ["$p",true]},
        "p": 1
    }},
])
```

B.3 Examples of Translation from Relational Algebra to MongoDB Queries

B.3.1 Sub-queries

Let $\boldsymbol{q}_1$ be the MUPG query defined in Example 9 that retrieves all pairs of scientists that received the same award in the same year, and $\boldsymbol{q}_2$ the following query that finds the persons born before 1950:

```
db.bios.aggregate([
  {$match: {
      "birth": {$lt: ISODate("1950-01-01")}} },
  {$project: {
      "_id": false,
      "firstName": "$name.first",
      "lastName":  "$name.last"
    }}
])
```

Then $\mathsf{pipeline}(\boldsymbol{q}_1, \boldsymbol{q}_2)$, as defined in A.3.2, is the following query:

```
db.bios.aggregate([
  // Duplication
  {$project: {"origDoc": "$$ROOT", "actDoc": [1, 2]}},
  {$unwind: "$actDoc"},
  // Specialization
  {$project: {"actDoc": 1,
      "doc1": {$cond: {if: {$eq: ["$actDoc", 1]}, then: "$origDoc", else: "$dummy"}},
```

```
      "doc2": {$cond: {if: {$eq: ["$actDoc", 2]}, then: "$origDoc", else: "$dummy"}}}},
  // Sub-query 1
  {$match: {$or: [
      {"actDoc": {$ne: 1}},
      {"doc1.awards": {$exists: true, $ne: null, $ne: []}}]}},
  {$unwind: {path: "$doc1.awards", preserveNullAndEmptyArrays: true}},
  {$project : {"actDoc": true, "doc2": true, "doc1._id": true, "doc1.awards": true,
      "doc1.doc._id": {$cond: {if: {$eq: ["$actDoc", 1]},
          then: "$doc1._id", else: "$dummy"}},
      "doc1.doc.name": {$cond: {if: {$eq: ["$actDoc", 1]},
          then: "$doc1.name", else: "$dummy"}}}},
  {$group: {
      _id: {"actDoc": "$actDoc", "doc1_awardYear": "$doc1.awards.year",
        "doc1_awardName": "$doc1.awards.award"},
      "doc1_docs": {$addToSet: "$doc1.doc"}, "doc2": {$push: "$doc2"}}},
  {$project: { _id: false,
      "doc1.docs": "$doc1_docs", "doc2": "$doc2", "actDoc": "$_id.actDoc",
      "doc1._id.awardYear": "$_id.doc1_awardYear", "doc1._id.awardName": "$_id.doc1_awardName"}},
  {$project: {"actDoc": true,
      "doc1": {$cond: {if: {$eq: ["$actDoc", 1]}, then: "$doc1", else: "$dummy"}},
      "doc2": {$cond: {if: {$eq: ["$actDoc", 2]}, then: "$doc2", else: "$dummy"}}}},
  {$unwind: {path: "$doc2", preserveNullAndEmptyArrays: true}},
  {$project : {"actDoc": true, "doc2": true, "doc1._id": true,
      "doc1.doc1": {$cond: {if: {$eq: ["$actDoc", 1]}, then: "$doc1.docs", else: "$dummy"}},
      "doc1.doc2": {$cond: {if: {$eq: ["$actDoc", 1]}, then: "$doc1.docs", else: "$dummy"}}}},
  {$match: {$or: [
      {"actDoc": {$ne: 1}},
      {"doc1.doc1": {$exists: true, $ne: null, $ne: []}}]}},
  {$unwind: {path: "$doc1.doc1", preserveNullAndEmptyArrays: true}},
  {$match: {$or: [
      {"actDoc": {$ne: 1}},
      {"doc1.doc2": {$exists: true, $ne: null, $ne: []}}]}},
  {$unwind: {path: "$doc1.doc2", preserveNullAndEmptyArrays: true}},
  {$project: {"actDoc": true, "doc2": true, "doc1._id": true,
      "doc1.lastName1": {$cond: {if: {$eq: ["$actDoc", 1]},
          then: "$doc1.doc1.name.last", else: "$dummy"}},
      "doc1.lastName2": {$cond: {if: {$eq: ["$actDoc", 1]},
          then: "$doc1.doc2.name.last", else: "$dummy"}},
      "doc1.awardName": {$cond: {if: {$eq: ["$actDoc", 1]},
          then: "$doc1._id.awardName", else: "$dummy"}},
      "doc1.awardYear": {$cond: {if: {$eq: ["$actDoc", 1]},
          then: "$doc1._id.awardYear", else: "$dummy"}},
      "doc1.toJoin": {$cond: {if: {$eq: ["$actDoc", 1]},
          then: {$ne: ["$doc1.doc1._id", "$doc1.doc2._id"]},
          else: "$dummy"}}}},
  {$match: {$or: [{"actDoc": {$ne: 1}}, {"doc1.toJoin": true}]}},
  {$project : {
      "actDoc": true, "doc2": true, "doc1.lastName1": true,
      "doc1.lastName2": true, "doc1.awardName": true, "doc1.awardYear": true}},
  // Sub-query 2
  {$match: {$or: [{"actDoc": {$ne: 2}}, {"doc2.birth": {$lt: ISODate("1950-01-01")}}]}},
  {$project: {
      "actDoc": true, "doc1": true,
      "doc2.firstName": {$cond: {if: {$eq: ["$actDoc", 2]}, then: "$doc2.name.first", else: "$dummy"}},
      "doc2.lastName": {$cond: {if: {$eq: ["$actDoc", 2]}, then: "$doc2.name.last", else: "$dummy"}}}},
  // Normalization
  {$group: {_id: null, "rel1": {$push: "$doc1"}, "rel2": {$push: "$doc2"}}}
])
```

B.3.2 Low level Minus

Let $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ be queries that respectively retrieves all the persons and the persons born before 1950 ($\boldsymbol{q}_2$ was described in B.3.1). Then $\boldsymbol{q}_1 \setminus \boldsymbol{q}_2$ is the following query:

```
db.bios.aggregate([
  // Duplication
  {$project: {
      "_id": false,
      "origDoc": "$$ROOT",
      "actDoc": [1, 2]
    }},
  {$unwind: "$actDoc"},
  // Specialization
  {$project: {
      "actDoc": true,
      "doc1": {$cond: {if: {$eq: ["$actDoc", 1]}, then: "$origDoc", else: "$dummy"}},
      "doc2": {$cond: {if: {$eq: ["$actDoc", 2]}, then: "$origDoc", else: "$dummy"}}
    }},
  // Subquery 1
  {$project: {
      "actDoc": true, "doc2": true,
      "doc1.firstName": "$doc1.name.first",
      "doc1.lastName":  "$doc1.name.last"
```

```
    }},
  // Subquery 2
  {$match: {$or: [{"actDoc": {$ne: 2}}, {"doc2.birth": {$lt: ISODate("1950-01-01")}}]}},
  {$project: {
      "actDoc": true, "doc1": true,
      "doc2.firstName": "$doc2.name.first",
      "doc2.lastName":  "$doc2.name.last"
    }},
  // Normalization
  {$group: {
      _id: null,
      "rel1": {$push: "$doc1"},
      "rel2": {$push: "$doc2"}
    }},
  // Minus
  {$unwind: "$rel1"},
  {$unwind: "$rel2"},
  {$project: {
      "rel1": true,
      "inIntersection": {$and: [
        {$eq: ["$rel1.firstName", "$rel2.firstName"]},
        {$eq: ["$rel1.lastName", "$rel2.lastName"]}
        ]}
    }},
  {$group: {
      _id: {"rel1": "$rel1"},
      "inIntersection": {$push: "$inIntersection"}
    }},
  {$match: {"inIntersection": {$ne: true}}},
  {$project: {
      _id: false,
      "rel1": "$_id.rel1"
    }}
])
```

B.3.3 Union

Let $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ be queries that retrieve respectively the persons born before 1950 and after 1949. Then $\boldsymbol{q}_1 \cup \boldsymbol{q}_2$ is the following query:

```
db.bios.aggregate([
  // Duplication
  {$project: {
      "_id": false,
      "origDoc": "$$ROOT",
      "actDoc": [1, 2]
    }},
  {$unwind: "$actDoc"},
  // Specialization
  {$project: {
      "actDoc": true,
      "doc1": {$cond: {if: {$eq: ["$actDoc", 1]}, then: "$origDoc", else: "$dummy"}},
      "doc2": {$cond: {if: {$eq: ["$actDoc", 2]}, then: "$origDoc", else: "$dummy"}}
    }},
  // Sub-query 1
  {$match: {$or: [{"actDoc": {$ne: 1}}, {"doc1.birth": {$lt: ISODate("1950-01-01")}}]}},
  {$project: {
      "actDoc": true,
      "doc2": true,
      "doc1.firstName": "$doc1.name.first",
      "doc1.lastName":  "$doc1.name.last"
    }},
  // Sub-query 2
  {$match: {$or: [{"actDoc": {$ne: 2}}, {"doc2.birth": {$gte: ISODate("1950-01-01")}}]}},
  {$project: {
      "actDoc": true,
      "doc1": true,
      "doc2.firstName": "$doc2.name.first",
      "doc2.lastName":  "$doc2.name.last"
    }},
  // Normalization
  {$group: {
      _id: null,
      "rel1": {$push: "$doc1"},
      "rel2": {$push: "$doc2"}
    }},
  // Union
  {$project: {
      "rel": ["$rel1", "$rel2"]
    }},
  {$unwind: "$rel"},
  {$unwind: "$rel"},
  {$group: {
```

```
      _id: null,
      "rel": {$push: "$rel"}
    }}
])
```

B.3.4 Subqueries without conditionals

```
db.bios.aggregate([
  {"$project":{
      "origDoc": "$$ROOT",
      "actDoc": [1,2],
    }},
  {$unwind: "$actDoc"},
  {$project: {
      "origDoc.actDoc": "$actDoc",
      "origDoc.awards": "$origDoc.awards",
      "origDoc.name": "$origDoc.name",
      "actDoc": 1}
  },
  {$project: {
      "doc1": "$origDoc",
      "doc2": "$origDoc",
    "actDoc": 1 }},
  {$unwind: {path: "$doc1.awards", preserveNullAndEmptyArrays: true}},
  {$match: {$or: [{"actDoc": 2}, {"doc1.awards.year": 2001}]}},
  {$unwind: {path: "$doc2.awards", preserveNullAndEmptyArrays: true}},
  {$match: {$or: [{"actDoc": 1}, {"doc2.awards.year": 1900}]}},
  //Cleaning
  {$project: {actDoc:1, "doc1": [null, "$doc1"], doc2: [null, "$doc2"]}},
  {$unwind: {path: "$doc1", preserveNullAndEmptyArrays: true}},
  {$unwind: {path: "$doc2", preserveNullAndEmptyArrays: true}},
  {$group: {_id: null, "doc1": {$push: "$doc1"}, "doc2": {$push: "$doc2"}}},
  {$unwind: {path: "$doc2", preserveNullAndEmptyArrays: true}},
  {$match: {$or: [{"doc2.actDoc" : 2}, {"doc2": null}]}},
  {$unwind: "$doc1"},
  {$match: {$or: [{"doc1.actDoc": 1}, {"doc1": null}]}},
  //
  {$group: {_id: null, "doc1": {$addToSet: "$doc1"}, "doc2": {$addToSet: "$doc2"}}},
  // Filter null
  {$project: {"doc2": 1, "doc1": 1,
      "doc1empty": {$eq: ["$doc1", [null]]},
      "doc2empty": {$eq: ["$doc2", [null]]}
      }},
  {$unwind: {path: "$doc1", preserveNullAndEmptyArrays: true}},
  {$match: {"doc1": {$ne: null}}},
  {$group: {
      _id: {"doc2": "$doc2", "doc1empty": "$doc1empty",
          "doc2empty": "$doc2empty"},
      "doc1": {$push: "$doc1"}
      }},
  {$project: {"doc2": "$_id.doc2", "doc1": 1, "doc1empty": "$_id.doc1empty",
      "doc2empty": "$_id.doc2empty"}},
  {$unwind: {path: "$doc2", preserveNullAndEmptyArrays: true}},
  {$match: {$or: [
      {"doc2": {$ne: null}},
      {"doc2empty": {$eq: true}}
      ]}},
  {$group: {_id: {"doc1": "$doc1"}, "doc2": {$push: "$doc2"}}},
  {$project: {"doc1": "$_id.doc1", "doc2": 1}}
])
```

New query (reaches limits):

```
db.bios.aggregate([
  // Duplication
  {$project: {
      "_id": false,
      "origDoc": "$$ROOT",
      "actDoc": [1, 2]
    }},
  {$unwind: "$actDoc"},
  // Specialization'
  {$project: {
      "actDoc": true,
      "doc1": "$origDoc",
      "doc2": "$origDoc"
    }},
  // Subquery 1
  {$match: {$or: [{"actDoc": {$ne: 1}}, {"doc1.awards": {$exists: true, $ne: null, $ne: []}}]}},
  {$unwind: {
      path: "$doc1.awards",
      preserveNullAndEmptyArrays: true
```

```
  }},
  {$project : {
      "actDoc": true,
      "doc2": true,
      "doc1._id": true,
      "doc1.awards": true,
      "doc1.doc._id": "$doc1._id",
      "doc1.doc.name": "$doc1.name"
  }},
  {$group: {
      _id: {
        "actDoc": "$actDoc",
        "doc1_awardYear": "$doc1.awards.year",
        "doc1_awardName": "$doc1.awards.award"
      },
      "doc1_docs": {$addToSet: "$doc1.doc"},
      "doc2": {$addToSet: "$doc2"}
  }},
  {$project: {
      _id: false,
      "doc1.docs": "$doc1_docs",
      "doc2": "$doc2",
      "actDoc": "$_id.actDoc",
      "doc1.awardYear": "$_id.doc1_awardYear",
      "doc1.awardName": "$_id.doc1_awardName"
  }},
  {$unwind: {
      path: "$doc2",
      preserveNullAndEmptyArrays: true
  }},
  {$project : {
      "actDoc": true,
      "doc2": true,
      "doc1._id": true,
      "doc1.doc1": "$doc1.docs",
      "doc1.doc2": "$doc1.docs",
  }},
  {$match: {$or: [{"actDoc": {$ne: 1}}, {"doc1.doc1": {$exists: true, $ne: null, $ne: []}}]}},
  {$unwind: {
      path: "$doc1.doc1",
      preserveNullAndEmptyArrays: true
  }},
  {$match: {$or: [{"actDoc": {$ne: 1}}, {"doc1.doc2": {$exists: true, $ne: null, $ne: []}}]}},
  {$unwind: {
      path: "$doc1.doc2",
      preserveNullAndEmptyArrays: true
  }},
  {$project: {
      "actDoc": true,
      "doc2": true,
      "doc1._id": true,
      "doc1.lastName1": "$doc1.doc1.name.last",
      "doc1.lastName2": "$doc1.doc2.name.last",
      "doc1.awardName": "$doc1._id.awardName",
      "doc1.awardYear": "$doc1._id.awardYear",
      "doc1.toJoin": {$ne: ["$doc1.doc1._id", "$doc1.doc2._id"]}
  }},
  {$match: {$or: [{"actDoc": {$ne: 1}}, {"doc1.toJoin": true}]}},
  {$project : {
      "actDoc": true,
      "doc2": true,
      "doc1.lastName1": true,
      "doc1.lastName2": true,
      "doc1.awardName": true,
      "doc1.awardYear": true
  }},
  //Subquery 2
  {$match: {$or: [{"actDoc": {$ne: 2}}, {"doc2.birth": {$lt: ISODate("1950-01-01")}}]}},
  {$project: {
      "actDoc": true,
      "doc1": true,
      "doc2.firstName": "$doc2.name.first",
      "doc2.lastName": "$doc2.name.last"
  }},
  // Cleaning
  {$project: {
      //TODO: integrate it
      "doc1": [null, {
        "actDoc": "$actDoc",
        "lastName1": "$doc1.lastName1",
        "lastName2": "$doc1.lastName2",
        "awardName": "$doc1.awardName",
        "awardYear": "$doc1.awardYear"
      }],
      "doc2": [null, {
        "actDoc": "$actDoc",
        "firstName": "$doc2.firstName",
```

```
        "lastName": "$doc2.lastName"
      }],
    }},
  {$unwind: "$doc1"},
  {$unwind: "$doc2"},
  {$group: {
      _id: null,
      "doc1": {$addToSet: "$doc1"},
      "doc2": {$addToSet: "$doc2"}
    }},
  {$unwind: "$doc1"},
  {$unwind: "$doc2"},
  {$match: {$or: [
      {"doc1.actDoc": 1},
      {"doc1": null}]}},
  {$match: {$or: [
      {"doc2.actDoc": 2},
      {"doc2": null}]}},
  // Normalization
  {$group: {
      _id: null,
      // TODO: discuss
      "doc1": {$addToSet: "$doc1"},
      "doc2": {$addToSet: "$doc2"},
    }},
  // Non-null
  {$project: {
      "doc1": true,
      "doc2": true,
      "doc1empty": {$eq: ["$doc1", [null]]},
      "doc2empty": {$eq: ["$doc2", [null]]},
    }},
  {$unwind: "$doc1"},
  {$unwind: "$doc2"},
  {$match: {$or: [{"doc1": {$ne: null}}, {"doc1empty": true}]}},
  {$group: {
      _id: {
        "doc2": "$doc2",
        "doc1empty": "$doc1empty",
        "doc2empty": "$doc2empty"},
      "doc1": {$addToSet: "$doc1"}
    }},
  {$project: {
      "doc1": true,
      "doc2": "$_id.doc2",
      "doc1empty": "$_id.doc1empty",
      "doc2empty": "$_id.doc2empty"
    }},
  {$match: {$or: [{"doc2": {$ne: null}}, {"doc2empty": true}]}},
  {$group: {
      _id: {
        "doc1": "$doc1",
        "doc1empty": "$doc1empty",
        "doc2empty": "$doc2empty"},
      "doc2": {$addToSet: "$doc2"}
    }},
  {$project: {
      "doc2": true,
      "doc1": "$_id.doc1",
      "doc1empty": "$_id.doc1empty",
      "doc2empty": "$_id.doc2empty"
    }},
  {$group: {
      _id: null,
      "doc1": {$addToSet: "$doc1"},
      "doc2": {$addToSet:"$doc2"}
    }}
])
```

B.3.5 Lookup for cross-product

```
db.bios.aggregate([
    {$group: {_id: null, "doc": {$addToSet: "$$ROOT"}}},
    {$lookup: {from: "bios", localField: "a", foreignField: "b", as: "others" }}
])
```

B.4 Examples for the Encodings in the Lower Bound Reductions

B.4.1 From QBF to MUPG

The collection `boolean` contains one document `{"values": [0,1]}`, and we encode the following satisfiable QBF:

$$\varphi = \exists x_1.\forall x_2.\exists x_3.\forall x_4.(x_1 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee x_3).$$

```
db.boolean.aggregate([
{$project: {"x1": "$values", "x2": "$values", "x3": "$values", "x4": "$values"}},
{$unwind: "$x1"},
{$unwind: "$x2"},
{$unwind: "$x3"},
{$unwind: "$x4"},
{$project: {"x1": true, "x2": true, "x3": true, "x4": true,
    "phi": {$and: [
        {$or: [
            {$eq: ["$x1", 1]},
            {$eq: ["$x2", 1]},
            {$eq: ["$x4", 1]}]},
        {$or: [
            {$eq: ["$x1", 0]},
            {$eq: ["$x2", 1]},
            {$eq: ["$x3", 1]}]},
            ]}}},
//
{$group: {
    _id: { "x1": "$x1", "x2": "$x2", "x3": "$x3"},
    "values": {$push: "$phi"} }},
{$project: { "phi": {$eq: ["$values", [true, true]] } }},
//
{$group: {
    _id: { "x1": "$_id.x1", "x2": "$_id.x2"},
    values: {$push: "$phi"}}},
{$project: { "phi": {$ne: ["$values", [false, false]] } }},
//
{$group: {
    _id: { "x1": "$_id.x1"},
    "values": {$push: "$phi"}}},
{$project: { "phi": {$eq: ["$values", [true, true]] } }},
//
{$group: {
    _id: null,
    "values": {$push: "$phi"}}},
{$project: { "phi": {$ne: ["$values", [false, false]] } }},
//
{$match: {"phi": true}}
])
```

B.4.2 From QBF to MUG

This query is a modification of the MUPG query above. The commented out match stage encodes the satisfiable formula from the previous reduction. The other match encodes the unsatisfiable formula

$$\varphi = \exists x_1.\forall x_2.\exists x_3.\forall x_4.(x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge (x_1 \vee x_3 \vee x_4).$$

The first project stage is present only because we use the same collection `boolean` as above. Obviously, we can create a collection with n arrays `x`i of the form `[0,1]` directly in the database, and then we can omit the first stage.

```
db.boolean.aggregate([
{$project: {"x1": "$values", "x2": "$values", "x3": "$values", "x4": "$values"}},
{$unwind: "$x1"},
{$unwind: "$x2"},
{$unwind: "$x3"},
{$unwind: "$x4"},
/*{$match: {$and: [
        {$or: [
            {"x1": 1},
            {"x2": 1},
            {"x4": 1}]},
```

```
        {$or: [
            {"x1": 0},
            {"x2": 1},
            {"x3": 1}]},
            ]}},*/
{$match: {$and: [
        {$or: [
            {"x1": 1},
            {"x2": 0},
            {"x4": 1}]},
        {$or: [
            {"x1": 0},
            {"x3": 1},
            {"x4": 0}]},
        {$or: [
            {"x2": 1},
            {"x3": 0}]},
        {$or: [
            {"x1": 1},
            {"x3": 1},
            {"x4": 1}]},
]}},
//
{$group: {
    _id: { "x1": "$x1", "x2": "$x2", "x3": "$x3"},
    "values": {$push: "$x4"} }},
{$match: { $or: [{"values": [0, 1]}, {"values": [1, 0]}] } },
//
{$group: {
    _id: { "x1": "$_id.x1", "x2": "$_id.x2"},
    "values": {$push: "$_id.x3"} }},
{$match: { "values": {$ne: []} }},
//
{$group: {
    _id: { "x1": "$_id.x1"},
    "values": {$push: "$_id.x2"} }},
{$match: { $or: [{"values": [0, 1]}, {"values": [1, 0]}] } },
//
{$group: {
    _id: null,
    "values": {$push: "$_id.x1"} }},
//
{$match: {"values": {$ne: []}}}
])
```

B.4.3 From SAT to MP with $map and $filter

$\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3).$

```
db.boolean.aggregate([
  {$project: {"a": [{"x1": {$literal: 0}}, {"x1": {$literal: 1}}]}},
  //
  {$project: {
      "a0": {$map: {
          input: "$a", as: "ass",
          in: {"x1": "$$ass.x1", "x2": {$literal: 0}}
        }},
      "a1": {$map: {
          input: "$a", as: "ass",
          in: {"x1": "$$ass.x1", "x2": {$literal: 1}}
        }},
    }},
  {$project: {"a": {"$setUnion" : ["$a0", "$a1"]}}},
  //
  {$project: {
      "a0": {$map: {
          input: "$a", as: "ass",
          in: {"x1": "$$ass.x1", "x2": "$$ass.x2", "x3": {$literal: 0}}
        }},
      "a1": {$map: {
          input: "$a", as: "ass",
          in: {"x1": "$$ass.x1", "x2": "$$ass.x2", "x3": {$literal: 1}}
        }},
    }},
  {$project: {"a": {"$setUnion" : ["$a0", "$a1"]}}},
  //
  {$project: {"x": {$filter: {
          input: "$a", as: "item",
          cond: {$and: [
            {$or: [
              {$eq: ["$$item.x1", 1]},
              {$eq: ["$$item.x2", 0]},
              {$eq: ["$$item.x3", 1]}
```

```
              ]},
            {$or: [
              {$eq: ["$$item.x2", 1]},
              {$eq: ["$$item.x3", 1]}
              ]},
            {$or: [
              {$eq: ["$$item.x1", 0]},
              {$eq: ["$$item.x3", 0]}
              ]}
            ]}}}
    }},
  {$match: {"x": {$ne: []}}}
])
```