Containment of Data Graph Queries

Egor V. Kostylev
University of Oxford and
University of Edinburgh
egor.kostylev@cs.ox.ac.uk

Juan L. Reutter
PUC Chile
jreutter@ing.puc.cl

Domagoj Vrgoč
University of Edinburgh
domagoj.vrgoc@ed.ac.uk

ABSTRACT

The graph database model is currently one of the most popular paradigms for storing data, used in applications such as social networks, biological databases and the Semantic Web. Despite the popularity of this model, the development of graph database management systems is still in its infancy, and there are several fundamental issues regarding graph databases that are not fully understood. Indeed, while graph query languages that concentrate on topological properties are now well developed, not much is known about languages that can query both the topology of graphs and their underlying data.

Our goal is to conduct a detailed study of static analysis problems for such languages. In this paper we consider the containment problem for several recently proposed classes of queries that manipulate both topology and data: regular queries with memory, regular queries with data tests, and graph XPath. Our results show that the problem is in general undecidable for all of these classes. However, by allowing only positive data comparisons we find natural fragments that enjoy much better static analysis properties: the containment problem is decidable, and its computational complexity ranges from PSPACE-complete to EXPSPACE-complete. We also propose extensions of regular queries with an inverse operator, and study query evaluation and query containment for them.

Categories and Subject Descriptors

F.4.1 [**Mathematical logic and formal languages**]: Mathematical logic; H.2.3 [**Database management**]: Languages—*Query Languages*

General Terms

Theory, Languages, Algorithms

Keywords

Graph Databases, data values, containment

1. INTRODUCTION

Managing graph-structured data is one of the most active topics in the database community these days. Although first introduced in the eighties [16, 17], the model has recently gained popularity due to a high demand from services that find the relational model too restrictive, such as social networks, Semantic Web, crime detection networks, biological databases and many others. There are several vendors offering graph database systems [19, 23, 36] and a growing body of literature on the subject (for a survey see e.g. [2, 7, 44]).

In such applications the data is usually modelled as a graph, with each node describing one entity in the database, for example a user in a social network, and the edges of the graph representing various connections between nodes, such as friends in a social network, supervisor connections in a database modelling the structure of a company, etc. Nodes can have various types of connections, so usually each edge in the graph is labelled. Finally, nodes by themselves contain the actual data, modelled as traditional relational data with values coming from an infinite domain [2].

To query graph-structured data, one can, of course, use traditional relational languages and treat the model as a relational database. What makes graph databases attractive in modern applications is the ability to query intricate navigational patterns between objects, thus obtaining more information about the *topology* of the stored data and how it relates to the actual data. Earliest graph query languages, such as regular path queries (RPQs) [17] and conjunctive regular path queries (CRPQs) [13, 16], concentrate on retrieving the topology of the graph and ignore the actual data stored. These languages have been well studied in last decades, and many extensions were defined for them, such as 2-way RPQs [13], that allow backward navigation; nested regular expressions [5], that allow existential tests; or extended CRPQs [3], that allow checks of nontrivial relations amongst paths. Industry is also taking account of navigational languages. For example, RPQs have been added to SPARQL, a query language for Semantic Web graph databases [27], as a primitive for querying navigational properties of graphs.

But purely navigational languages such as RPQs or CRPQs cannot reason on the data stored in the nodes. Thus such data was usually queried using relational languages, without a way of specifying the interplay between the data stored and various navigational patterns connecting the data.

10.5441/002/icdt.2014.16

This interplay is indeed a requirement in many applications using graph-structured data. For example, in a database modelling the inner workings of a company one might be interested in finding chains of people living in the same city that are connected by professional links, or in a social network one could look for a sequence of friends, all of which like the same type of music. Recently, several languages that can handle such queries have been proposed [30–32] and they were all built on the idea of extending RPQs, or some variation thereof, with the ability to reason about data values that appear along the navigated path.

Our goal is to study static analysis aspects of this new generation of graph query languages, understanding them as basic building blocks for more complex navigational languages. We concentrate on the query containment problem, which is the problem of deciding, given two queries in some graph language, whether the answer set of the first query is contained in the answer set of the second one. Deciding query containment is a fundamental problem in database theory, and is relevant to several complex database tasks such as data integration [29], query optimisation [1], view definition and maintenance [24], and query answering using views [15].

The importance of this problem motivated sustained research for relational query languages (see e.g. [1]), XML query languages (see e.g. [41]) and even extensions of RPQs and other graph query languages [3, 4, 13, 20]. The overall conclusion is that containment is generally undecidable for first order logic and other similar formalisms (see e.g. [1]), but becomes decidable if we restrict to queries with little or no negation. For example, containment of conjunctive queries is NP-complete, while containment of RPQs, 2-way RPQs and nested regular expressions is PSpace-complete. For CRPQs it jumps to ExpSpace-complete.

While much is known about the containment of above mentioned classes of queries, no detailed study has been conducted for query languages that deal both with navigational and data aspects of graph databases. In this work we concentrate on three such languages. Namely, we consider *regular queries with memory* (or *RQMs* for short), *regular queries with data tests* (or *RQDs*), both introduced in [32], as well as a recent adaptation of the widely used XML query language XPath to the graph setting, which is called *graph* XPath (or GXPath) [31]. We primarily concentrate on containment, but the techniques presented here can easily be adapted to deal with other similar problems, such as satisfiability or equivalence of queries.

The intuition behind RQMs is that one can navigate through a graph in the same way as with RPQs, but along the path it is also possible to store a data value into a register and later on compare it with another value encountered further on the path. This idea is very similar to the one of register automata [28, 37] and in fact one can show that these two formalisms are equivalent [33]. RQDs operate in a similar fashion, but storing and comparing values adheres to a more strict stack-like discipline, so they enjoy much better evaluation properties. Lastly, the language of GXPath allows one to define patterns that are not necessarily just paths, as it is for RQMs and RQDs, and also provides the ability to test whether some data values in these patterns are equal.

Contributions By using equivalence of RQMs with register automata, we obtain our first result: the problem of checking whether one RQM is contained in another RQM is undecidable. This, of course, opens up the question of fragments of the language that do have decidable containment problem. The class of *positive RQMs* is one of such fragments, in which we allow testing only if two data values are equal, but not different. We show that the problem of positive RQM query containment is decidable, and, in fact, ExpSpace-complete—the same complexity as for CRPQs [13].

Next we move onto the class of RQDs, which was shown to be strictly contained in the class of RQMs [32]. The imposed restrictions to RQMs are quite heavy, and computational complexity of query evaluation drops by almost one exponent when we consider RQDs instead of RQMs. For this reason one may expect the containment problem to be decidable for RQDs. On the contrary, as we show, it remains undecidable even in this restricted scenario. However, this changes once again when we consider *positive RQDs*, for which a PSpace algorithm for testing containment is obtained. This is the best possible bound for any extension of RPQs, since their containment is already PSpace-hard [14].

A common assumption when considering graph languages is that edges can be traversed in both directions. Indeed, the authors in [12,13] argue that any practical query mechanism for graphs should incorporate this functionality, as there are many scenarios when backward navigation is required. It is therefore natural to study what happens when RQMs and RQDs are extended with the inverse operator. This gives rise to two new classes of languages, called 2RQMs and 2RQDs, respectively. Remarkably, we show that adding this operator carries no extra computational cost with respect to query evaluation. However, it does make a big difference for containment, as even the subclass of 2RQMs that allows only positive data comparisons has undecidable query containment problem.

Finally, we consider GXPath and its various dialects. This language has recently attracted attention because it provides considerable expressive power while maintaining good query evaluation properties (in particular, the combined complexity is in polynomial time). However, with respect to containment the story is different: even the navigational fragment that does not allow data value comparisons has undecidable containment problem. Although this bound follows from some folklore results on satisfiability of the three variable fragment of first order logic, we could not find a formal proof of this fact and, hence, provide a self-contained one by a reduction from an unusual variation of the tiling problem.

The reason for the undecidability of GXPath is the presence of a powerful negation operator that allows complementation of binary relations. We show, that if one excludes such negation from the language, then containment becomes decidable (ExpTime-complete). Such a language is in fact close to propositional dynamic logic (PDL), whose containment is also known to be ExpTime-complete [26].

The classes above do not consider data values tests in GXPath queries. Following [31], we consider an extension

of these classes with an operator to test whether data values at the beginning and at the end of a path are same or different. This language can simulate all RQDs [31], and thus our previous results imply that inequalities in tests immediately lead to undecidability of the containment problem.

Hence the natural way to obtain decidability is to consider GXPath queries that allow only equalities between data values. Whether or not the containment problem is decidable for this class promises to be a challenging task, worthy of a research line of its own. Indeed, even seemingly simpler problems in the XML case (that is, trees with data) are still unanswered [8, 9], and the ones that have been solved usually require very intricate techniques that cannot be applied in the graph scenario (see e.g. [18, 35]).

Overall, we see that when containment is considered, the situation is quite different for languages handling both topology and data than it is for traditional graph languages allowing only navigational queries. While for the latter containment is generally decidable, we show that for the languages considered here the problem resembles behaviour of relational algebra, where containment is undecidable for the full language, but various restrictions on the use of negation lead to decidable fragments. Hence, the existence of real-world relational systems which deal with similar problems, demonstrates that undecidability or high complexity should not be viewed as an insurmountable obstacle for practical use of the languages studied here, but as a foundation for further research.

Organization In Section 2 we formally define the data model and the problem studied. In Sections 3 and 4 we introduce RQMs and RQDs, respectively, and study their containment problems. In Section 5 we show how these classes can be extended with inverses, and turn our attention to GXPath in Section 6. We conclude with some remarks about future work in Section 7. Due to space limitations, most of the proofs are only sketched, and complete proofs can be found in the appendix.

2. PRELIMINARIES

Data graphs Let Σ be a finite alphabet of *labels* and $\mathcal{D}$ an infinite set of *data values*. A *data graph* over labels Σ and data values $\mathcal{D}$ is a triple $\langle V, E, \rho\rangle$, where:

- V is a finite set of nodes,
- $E \subseteq V \times \Sigma \times V$ is a set of labelled edges, and
- $\rho : V \to \mathcal{D}$ is a function that assigns a data value to each node in the graph.

An example of a data graph is shown in Figure 1. If data values are not important, we disregard ρ and only talk about *graphs* $\langle V, E\rangle$ over Σ.

Regarding data values, this paper follows [31, 32] and the standard convention for data trees (as a model for XML), and assumes that data values are attached to nodes. There are of course other possibilities, but they are all essentially equivalent. We also assume that each node is assigned with

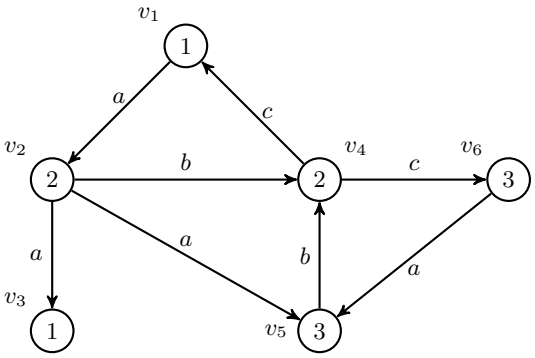

Figure 1: A data graph over labels $\{a, b, c\}$ and natural numbers as data values, in which nodes are v_i, $1 \leq i \leq 6$.

a single data value. This is not a real restriction, since tuples of attributes can be modelled by a set of edges, each labelled with the attribute name and connecting the current node to a new node with a data value for the corresponding attribute.

Paths A *path* π between nodes v_1 and v_n in a graph $\langle V, E\rangle$ is a sequence

$$v_1 a_1 v_2 a_2 v_3 \dots v_{n-1} a_{n-1} v_n,$$

such that each (v_i, a_i, v_{i+1}), for $1 \leq i < n$, is an edge in E. The *label* of the path π is the word $a_1 \dots a_{n-1}$ obtained by reading the edge labels appearing along this path.

Queries The default core of each query language for graphs is *regular path queries* (or *RPQs*) which are just regular languages over labels Σ, usually defined by regular expressions. The *evaluation* $[\![e]\!]^G$ of an RPQ e over a graph G, is the set of all pairs (v_1, v_2) of nodes in G for which there exists a path from v_1 to v_2 with the label from the language of e.

There are a number of extensions of RPQs proposed in the literature. In this paper we concentrate on those that are capable of dealing with data values. Also, all of the queries we study are binary queries, i.e. such that their evaluations (i.e. answers) are sets of pairs of nodes. We denote by $[\![e]\!]^G$ the evaluation of a query e over a data graph G.

Containment A query e_1 is *contained* in a query e_2 (written $e_1 \subseteq e_2$) if for each data graph G over Σ and $\mathcal{D}$ we have that

$$[\![e_1]\!]^G \subseteq [\![e_2]\!]^G.$$

The queries e_1 and e_2 are *equivalent* (written $e_1 \equiv e_2$) iff $[\![e_1]\!]^G = [\![e_2]\!]^G$ for every G.

The containment and equivalence are at the core of many static analysis tasks, such as query optimisation. All the classes of queries considered in this paper are closed under union, so these two problems are easily interreducible: $e_1 \equiv e_2$ iff e_1 and e_2 contain each other, and $e_1 \subseteq e_2$ iff $e_1 \cup e_2 \equiv e_2$. That is why in this paper we concentrate just on the first and consider the following decision problem parametrized by a class of queries $\mathcal{Q}$.

Containment ($\mathcal{Q}$)	
Input:	Queries e_1 and e_2 from $\mathcal{Q}$.
Question:	Is e_1 contained in e_2?

The semantics of RPQs is defined for graphs, but it is straightforward to see that for any two RPQs e_1 and e_2, we have that $e_1 \subseteq e_2$ if and only if the language accepted by the regular expression e_1 is contained in the language accepted by e_2 [14]. From this fact we obtain that containment of RPQs is PSpace-complete, following the classic result that containment of regular expressions is PSpace-complete. Since all of the classes of queries studied in this paper are extensions of RPQs, this establishes a lower bound for containment of any of these classes.

3. REGULAR QUERIES WITH MEMORY

Regular queries with memory, or *RQMs* for short, were introduced in [32] (where they were called *regular expressions with memory*) as a formalism for querying data graphs that allows data comparisons while navigating through the structure of the graph. They are based on *register automata*, an extension of finite-state automata for words over infinite alphabets (see, e.g. [32, 37] for a detailed description).

The idea of RQMs is the following. They can store data values in a number of named *registers*, while parsing the input graph according to a specified regular navigation pattern. Also, they can compare the current data value with values that had previously been stored. An example of an RQM is the expression $\downarrow x.a^+[x^=]$, which returns all pairs (v, v') of nodes in a graph that have the same data value and are connected by a path labelled only with a's. Intuitively the expression works as follows: it first stores the data value of the node v into register x, and after navigating an a-labelled path, it checks that the node v' at the end of this path has the same data value as the first node. This check is done via test $x^=$ which makes sure that the data value of v' is the same as the one stored in register x.

The proposal of RQMs as a formalism for querying data graphs was motivated not only by their ability to handle data values, but also by the low computational complexity of their evaluation: it is PSpace-complete in general, and NLogSpace-complete if the query is fixed (i.e. in *data complexity*) [32]. Hence, their complexity is essentially the same as for first-order or relational algebra queries.

3.1 Syntax and Semantics of RQMs

Let X be a countable set of *registers*. We denote them by letters x, y, z, etc. A *condition* over X is a positive boolean combination of atoms of the form $x^=$ or $x^{\neq}$, for $x \in X$.

Definition 3.1. *A* regular query with memory *(or* RQM*) over an alphabet of labels Σ and set of registers X is an expression satisfying the grammar*

$$e \;:=\; \varepsilon \mid a \mid e \cup e \mid e \cdot e \mid e^+ \mid e[c] \mid \downarrow x.e \quad (1)$$

where ε is the empty word, a ranges over labels, x over registers, and c over conditions.

Before formally defining the semantics, let us give some examples of RQMs and explain their intuitive meaning.

$$\begin{aligned}
\mathcal{H}^G(\varepsilon) &= \{(s,s) \mid s \text{ is a state}\},\\
\mathcal{H}^G(a) &= \{((v,\lambda),(v',\lambda)) \mid (v,a,v') \in E\},\\
\mathcal{H}^G(e_1 \cup e_2) &= \mathcal{H}^G(e_1) \cup \mathcal{H}^G(e_2),\\
\mathcal{H}^G(e_1 \cdot e_2) &= \mathcal{H}^G(e_1) \circ \mathcal{H}^G(e_2),\\
\mathcal{H}^G(e^+) &= \mathcal{H}^G(e) \cup \mathcal{H}^G(e \cdot e) \cup \ldots,\\
\mathcal{H}^G(e[c]) &= \{((v,\lambda),(v',\lambda')) \mid\\
&\qquad ((v,\lambda),(v',\lambda')) \in \mathcal{H}^G(e) \text{ and } (\rho(v'),\lambda') \models c\},\\
\mathcal{H}^G(\downarrow x.e) &= \{((v,\lambda),(v',\lambda')) \mid\\
&\qquad ((v,\lambda),(v',\lambda')) \in \mathcal{H}^G(e) \text{ and } \lambda(x) = \rho(v)\}.
\end{aligned}$$

Table 1: Definition of the function $\mathcal{H}^G$ with respect to a data graph G.

Example 3.2.

1. The RQM $\downarrow x.(a[x^=])^+$ returns all pairs of nodes connected by a path, along which all edges are labelled a and all data values are equal. The evaluation starts with $\downarrow x$, which stores the first data value into register x. The subexpression $(a[x^=])^+$ then checks that each subsequent label along the path is a, and that the data value of each node on this path is equal to the one of the first node (this is done by comparison with the value stored in register x). The fact that this subexpression is in the scope of $^+$ indicates that the length of the sequence of checks is of arbitrary length.
2. The RQM $\downarrow x.(a[x^{\neq}])^+$ returns all pairs of nodes connected by a path where all edges are labelled with a and the first data value is different from all other data values. It works analogously as the expression above, except that it checks for inequality.
3. The RQM $\downarrow x.(abc)^+[x^{\neq}]$ returns all pairs of nodes connected by a path, whose label is of the form $abc \ldots abc$, and the first data value is different from the last. Note that the order of $^+$ and condition is different from the previous examples: the condition is checked only once, after verifying that the label is in $(abc)^+$, i.e. at the end of the path. □

To define what it means for a data value to satisfy a condition we need the following notion. An *assignment* of registers X is a partial function λ, from X to the set of data values $\mathcal{D}$. Intuitively, an assignment models the current state of the registers at some point of computation, with some registers containing stored data values, and some still being empty. Formally, a data value d and an assignment λ *satisfy* a condition $x^=$ (or $x^{\neq}$) iff $\lambda(x)$ is defined and $d = \lambda(x)$ (or $d \neq \lambda(x)$, correspondingly). This satisfaction relation is denoted $\models$ and extended to general conditions in a straightforward way.

Given a data graph G and a set of registers X, a *state* is a pair consisting of a node of G and an assignment of X.

The semantics of RQMs over a data graph $G = \langle V, E, \rho\rangle$ is defined in terms of function $\mathcal{H}^G$, which binds each RQM with a set of pairs of states. The intuition of the set $\mathcal{H}^G(e)$, for some RQM e, is as follows. Given states $s = (v, \lambda)$ and $s' = (v', \lambda')$, the pair (s, s') is in $\mathcal{H}^G(e)$ if there exists a path w from v to v', such that the expression e can parse

w assuming that the registers are initialized according to λ, modified and compared as dictated by e, and the resulting assignment after traversing the path is λ'.

Formally, given a data graph $G = \langle V, E, \rho \rangle$, the function $\mathcal{H}^G$ is constructed by the inductive definition in Table 1.

The symbol $\circ$ in the table refers to the usual composition of binary relations:

$$\mathcal{H}^G(e_1) \circ \mathcal{H}^G(e_2) = \{(s_1, s_3) \mid \exists s_2 \text{ s.t. } (s_1, s_2) \in \mathcal{H}^G(e_1) \text{ and } (s_2, s_3) \in \mathcal{H}^G(e_2)\}.$$

Finally, the *evaluation* $[\![e]\!]^G$ of an RQM e over a data graph G is the following set of pairs of nodes in G:

$$\{(v, v') \mid \exists \lambda' \text{ s.t. } ((v, \bot), (v', \lambda')) \in \mathcal{H}^G(e)\},$$

where $\bot$ is the empty assignment.

Example 3.3. Consider the evaluations of expressions from Example 3.2 over the data graph from Figure 1:

1. the evaluation of $\downarrow x.(a[x^=])^+$ is $\{(v_6, v_5)\}$;
2. the evaluation of $\downarrow x.(a[x^\neq])^+$ is $\{(v_1, v_2), (v_1, v_5), (v_2, v_5), (v_2, v_3)\}$;
3. the evaluation of $\downarrow x.(abc)^+[x^\neq]$ contains $(v_1, v_6), (v_6, v_1), (v_2, v_1)$ and (v_2, v_6) (but not (v_6, v_6)). □

3.2 From Graphs to Words

As we mentioned in the preliminaries, standard algorithms for containment of RPQs rely on the fact that two RPQs are contained if and only if the regular languages they define are contained [14]. In this section we exhibit a similar behaviour for RQMs.

Data words are a widely studied extension of words over finite alphabets [40], in which every position carries not only a label from the finite alphabet Σ, but also a data value from the infinite domain $\mathcal{D}$. However, just for uniformity of presentation, we follow [32] and opt to the following essentially equivalent definition, by which data values are attached not to positions in a word, but "between" them.[1]

Definition 3.4. *A* data word *over a finite alphabet of labels Σ and infinite set of data values $\mathcal{D}$ is a sequence $d_1 a_1 d_2 a_2 \dots a_{n-1} d_n$, where $n > 0$, $a_i \in \Sigma$, for each $1 \leq i < n$, and $d_i \in \mathcal{D}$, for each $1 \leq i \leq n$.*

Every data word $w = d_1 a_1 d_2 \dots a_{n-1} d_n$ can be easily transformed to a data graph G_w, consisting of n different nodes with data values $d_1, \dots, d_n$, respectively, consequently connected by edges labelled with $a_1, \dots, a_{n-1}$, as illustrated in Figure 2.

[1] In [32] to distinguish this notion from the original, the term "data path" was used.

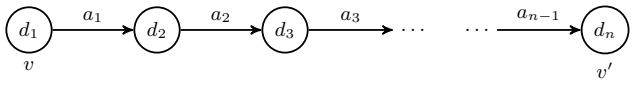

Figure 2: The data graph G_w corresponding to the data word $w = d_1 a_1 d_2 \dots a_{n-1} d_n$ (some node identifiers are omitted).

The semantics of RQMs over data words is defined in the straightforward way: a data word w is *accepted* by an RQM e iff $(v, v') \in [\![e]\!]^{G_w}$, where v and v' are the first and the last node of G_w. The set of all data words, accepted by an RQM e is denoted $\mathcal{L}(e)$.

Coming back to graphs, each path

$$v_1 a_1 v_2 a_2 v_3 \dots v_{n-1} a_{n-1} v_n,$$

in a data graph $\langle V, E, \rho \rangle$ has the *corresponding* data word

$$\rho(v_1) a_1 \rho(v_2) a_2 \rho(v_3) \dots \rho(v_{n-1}) a_{n-1} \rho(v_n).$$

As noted in [32], for each RQM e, data graph G, and nodes v, v' of G, it holds that $(v, v') \in [\![e]\!]^G$ iff there exists a path between v and v' such that its corresponding data word is accepted by e. Exploiting usual techniques in query containment we arrive at the following proposition, similar to the property of RPQs, mentioned in the preliminaries.

Proposition 3.5. *Given two RQMs e_1 and e_2, it holds that $e_1 \subseteq e_2$ iff $\mathcal{L}(e_1) \subseteq \mathcal{L}(e_2)$.*

In the proposition above $e_1 \subseteq e_2$ is defined on data graphs, but $\mathcal{L}(e_1)$ and $\mathcal{L}(e_2)$ are sets of data words.

3.3 Containment of RQMs

We now turn to the containment problem for RQMs. Unfortunately, as the following theorem shows, the power that RQMs gain through its data manipulation mechanism comes with a high price for static analysis tasks.

Theorem 3.6. *The problem* Containment *(*RQMs*) is undecidable.*

This fact follows from Proposition 3.5 and the undecidability of the containment problem for register automata ([37]), which are known to be equivalent to RQMs evaluated on data words ([32, 33]).

The theorem above naturally leads to question of finding decidable subclasses. It is known that testing containment of an expression using at most one register in an expression using at most two registers is decidable [37]. This approach appears to be too restrictive, and thus we concentrate instead on *positive RQMs*, i.e. RQMs that use only atoms of the form $x^=$ in the conditions. In [42] it was shown that the containment of positive RQMs is decidable, but no complexity bounds were given. The next theorem fills the gap.

THEOREM 3.7. *The problem* CONTAINMENT *(*positive RQMs*) is* EXPSPACE*-complete.*

PROOF SKETCH. For the EXPSPACE upper bound, let e_1 and e_2 be two RQMs over Σ, and assume that $e_1 \not\subseteq e_2$. Then by Proposition 3.5 there is a data word w, such that $(v, v') \in [\![e_1]\!]^{G_w}$, v and v' being the first and last nodes of the graph G_w. We associate to each node of G_w an assignment λ for the registers in e_1, representing the way how e_1 is parsed over G_w in accordance to the relation $\mathcal{H}^{G_w}(e_1)$. We show that one can always have such a graph G_w satisfying, in addition, the following property: two nodes u and u' of G_w have the same data value d if and only if the assignment for the registers in all nodes in the path from u to u' assigns d to at least one of the registers. In other words, once a data value is dropped by all registers, one does not encounter the data value again further along the path. Let now A_{e_1} and A_{e_2} be the register automata equivalent to e_1 and e_2, respectively. The above fact allows us to check whether w belongs to $\mathcal{L}(A_{e_1})$ but does not belong to $\mathcal{L}(A_{e_2})$, using only exponential space. The idea is to build a transition system that simulates a succesful run of A_{e_1} and all possible runs of A_{e_2}. Of course this is not feasible because there can be infinitely many assignments for the registers (because there are infinitely many data values), but by the fact explained above we show that we only need to focus on which registers are assigned the same values, and which ones are not (instead of their precise assignments). In other words, to check word w we can build a transition system whose states store roughly the following information: (1) a state of A_{e_1} and A_{e_2}, and (2) the equivalence relation formed by the data values stored in the registers of e_1 and e_2.

Hardness is by reduction from the acceptance problem of a Turing machine that works in EXPSPACE. The reduction is similar to the one used in [6, Theorem 6], except that the gadgets in this proof are constructed by taking advantage of registers, instead of the variable assignments used there. □

The previous proof relies on the fact that the set of registers X is unbounded. Carefully checking the proof reveals the following corollary. Here *n-bounded positive RQMs* refers to the class of positive RQMs which can use at most n registers.

COROLLARY 3.8. *Let n be a natural number. The problem* CONTAINMENT *(n-bounded* positive RQMs*) is* PSPACE*-complete.*

Hence, positive RQMs are a natural subclass of RQMs with decidable query containment. However, when comparing the complexity with the one for RPQs, we see that allowing positive data test comparisons results in an exponential jump. In the following section we consider another class of queries extending RPQs, which also allows data value comparisons, but in a more restricted way than RQMs. As we will see, the positive subclass of this class has the same complexity of query containment as RPQs.

4. REGULAR QUERIES WITH DATA TESTS

Looking for classes of queries handling data values, but having better query answering properties than RQMs, the authors of [32] introduced *regular queries with data tests*, or *RQDs* for short (these were called *regular expressions with equality* in the original paper). An example of such a query is the expression $a(b^+)_{=}c$, whose intention is to return all pairs of nodes connected by a path labelled $ab \dots bc$ and where the data values before and after the sequence of b's are the same.

All RQDs are RQMs, but the usage of registers is restricted: each stored data value can be retrieved and compared only once, and the order of these storing and retrieving operations is not arbitrary, but on the "last in, first out" basis. The data complexity of RQDs' evaluation is the same as for RQMs—in NLOGSPACE, but the combined complexity is much better, in fact tractable, in PTIME [32].

4.1 Syntax and Semantics of RQDs

The syntax for RQDs can be defined in a direct, much simpler way than for RQMs, without even mentioning registers and conditions.

DEFINITION 4.1. *A* regular query with data tests *(or* RQD*) over an alphabet of labels Σ is an expression satisfying the grammar*

$$e \;:=\; \varepsilon \mid a \mid e \cup e \mid e \cdot e \mid e^+ \mid e_{=} \mid e_{\neq} \qquad (2)$$

where a ranges over labels.

Again, before the formal definition of semantics we give some examples of RQDs and their connection to RQMs.

EXAMPLE 4.2. Recall RQMs from Example 3.3 (we consider them here in different order for better understanding of the relation between RQMs and RQDs).

1. The RQM $\downarrow x.(abc)^+[x^{\neq}]$ can be written as the RQD $((abc)^+)_{\neq}$: the first data value is stored, then the sequence of abc's is read, and then the value is retrieved and compared for inequality with the current one. Note that the stored value is used just once.
2. The RQM $\downarrow x.(a[x^{=}])^+$ can be written as the RQD $(a_{=})^+$: the first data value is stored; then a is read; then the stored data value is retrieved and compared with the current one for equality; if successful, this current value (equal to the original!) is stored again, another a is read, and so on. If the parsing continues, then the current data value is always equal to the original one, even if we use each stored value just once.
3. Contrary to the previous case, it can be shown that the RQM $\downarrow x.(a[x^{\neq}])^+$ cannot be expressed as an RQD: indeed, after the first comparison the original data value is lost, and storing the current data value (different from the original) cannot help with correct comparison on the next step.
4. The RQM $\downarrow x.a \downarrow y.b[y^{=}]c[x^{=}]$ can be written as the RQD $(ab_{=}c)_{=}$. However, the very similar RQM $\downarrow x.a \downarrow y.b[x^{=}]c[y^{=}]$ is not expressible as an RQD, since the sequence in which data values have to be retrieved does not respect the "first-in-last-out" discipline required by RQD syntax. □

$$
\begin{array}{lcl}
[\![\varepsilon]\!]^G & = & \{(v,v) \mid v \in V\},\\
[\![a]\!]^G & = & \{(v,v') \mid (v,a,v') \in E\},\\
[\![e_1 \cdot e_2]\!]^G & = & [\![e_1]\!]^G \circ [\![e_2]\!]^G,\\
[\![e_1 \cup e_2]\!]^G & = & [\![e_1]\!]^G \cup [\![e_2]\!]^G,\\
[\![e^+]\!]^G & \text{is the transitive closure of } & [\![e]\!]^G,\\
[\![e_=]\!]^G & = & \{(v,v') \mid (v,v') \in [\![e]\!]^G, \rho(v) = \rho(v')\},\\
[\![e_\neq]\!]^G & = & \{(v,v') \mid (v,v') \in [\![e]\!]^G, \rho(v) \neq \rho(v')\}.
\end{array}
$$

Table 2: Semantics of RQDs with respect to a data graph G. The composition of binary relations is again denoted $\circ$.

The semantics of RQDs is also defined in a much simpler way than for RQMs. The *evaluation* $[\![e]\!]^G$ of an RQD e over a data graph $G = \langle V, E, \rho\rangle$ is the set of all pairs (v_1, v_2) of nodes in V defined recursively in Table 2.

As Example 4.2 suggests, and as it is formally shown in [32], the class of RQDs is strictly contained in the class of RQMs. Indeed, to transform an RQD to RQM we just need to recursively replace each subexpression of the form $e_\sim$, $\sim \in \{=, \neq\}$, with the subexpression $\downarrow x.e[x^\sim]$, where x is a previously unused register. However, there are RQMs which cannot be transformed to RQDs, which is also justified by the lower complexity of query evaluation.

Similarly to RQMs, each RQD defines a language of data words. A data word w is *accepted* by an RQD e iff $(v, v') \in [\![e]\!]^{G_w}$, with G_w as in the Figure 2. The set of all data words accepted by an RQD e is denoted $\mathcal{L}(e)$. It is easy to see that for each RQD e, data graph G and nodes v, v' in G, it holds that $(v, v') \in [\![e]\!]^G$ iff there exists a path between v and v' such that its corresponding data word is accepted by e. This allows us to show an analogue of Proposition 3.5, thus reducing query containment to language containment.

Proposition 4.3. *Given two RQDs e_1 and e_2, it holds that $e_1 \subseteq e_2$ iff $\mathcal{L}(e_1) \subseteq \mathcal{L}(e_2)$.*

4.2 Containment of RQDs

RQDs were originally introduced as a restriction of RQMs that enjoys much better query evaluation properties. In light of this result, one might also hope for good behaviour when query containment is considered. Surprisingly, the following theorem shows that this is not the case.

Theorem 4.4. *The problem* Containment *(*RQDs*) is undecidable.*

Proof Sketch. The proof exploits the idea of coding the Post correspondence problem by data words from [37]. However, the expressions used there are RQMs and they rely on the fact that one can store a data value and then compare it with a value encountered later with no restrictions. This is not immediately possible when dealing with RQDs, since testing for (in)equality must adhere to the first-in-last-out discipline. The trick used to circumvent this issue is based on the observation that part of the coding from [37] can be reversed, thus allowing us to nest data value tests as dictated by the syntax of RQDs. □

This naturally opens the search for subclasses of RQDs with decidable containment problem. Similarly to positive RQMs, we now consider the class of *positive RQDs*, i.e. RQDs where subexpressions of the form $e_\neq$ are not allowed. We can obtain a positive RQM from a positive RQD by the described above procedure that transforms an RQD into an RQM. Hence, we again have a strict containment of the corresponding classes, and from Theorem 3.7 we conclude that containment of positive RQDs is decidable and in ExpSpace. However, the following theorem says that we can perform even better, in fact, the best possible in light of the PSpace lower bound for plain RPQs.

Theorem 4.5. *The problem* Containment *(*positive RQDs*) is* PSpace*-complete.*

Proof Sketch. The hardness follows from the bounds for RPQs, so next we give an idea of an PSpace algorithm which decides whether $\mathcal{L}(e') \subseteq \mathcal{L}(e)$ holds for positive RQDs e' and e. The main difficulty here is to identify and carefully exploit those nontrivial properties of RQDs that allow to extend a standard algorithm for containment of RPQs.

Let's start with a simple PSpace algorithm for containment of RPQs: (1) transform the RPQs to nondeterministic finite state automata (NFAs) A' and A without ε-transitions; (2) put a pebble to each of the initial states; (3) nondeterministically repeat moving the single pebble in A' along transitions, moving at the same time all the pebbles in A in parallel, along the transitions labelled the same as the current transition in A': if we have several options, the pebble multiplies, if a pebble cannot move, it is removed, if several pebbles meet, just one is left; (4) stop and fail if the pebble in A' is in a final state, but none of the pebbles in A are; stop and succeed if the search space is exhausted. Essentially, the set of pebbles in A is the state in the typical power set construction, done "on the fly".

A naive adaptation of this algorithm to deal with data values can be as follows.

(a) Before transforming to NFAs, *normalise* e and e' such that none of the equality checks $()_=$ can be opened together and none of them can be closed together on any run. This can be done, essentially, by applying the rules

$$((e_1)_= e_2)_= \rightsquigarrow (e_1)_= (e_2)_=, \quad (e_1 (e_2)_=)_= \rightsquigarrow (e_1)_= (e_2)_=,$$

and some others. After this, RQDs can be transformed to NFAs whose transitions have extra labels from the set $R = \{\emptyset, \uparrow, \downarrow, \downarrow\uparrow\}$, where $\uparrow$ means that an equality is opening, and $\downarrow$ that an equality is closing.

(b) Attach a stack of *reactions* to all the pebbles in A, where each reaction is a symbol from R. Then, during a run of the algorithm, if the pebble in A' moves along a transition with $\uparrow$, then every moved pebble in A pushes into its stack the extra label of the transition, but only if it is either $\emptyset$ or $\uparrow$; otherwise pebble does not pass (of course, the usual label matching is also checked). In turn, if the pebble in A' moves along a transition with $\downarrow$, then only those pebble pass, which popped extra label *pairs* with the label of the current transition: $\downarrow$ pairs with $\uparrow$, and $\emptyset$ pairs with itself.

The extra label $\downarrow\uparrow$ can be handled similarly.

By this, e.g. $(ab_{=}c)_{=}$ is contained in $(abc)_{=}$ because the only pebble in the second NFA when reading b has stack $(\uparrow, \emptyset)$ and the current label is pairing $\emptyset$. The same $(ab_{=}c)_{=}$ is contained in $ab_{=}c$, because, after b the stack is $(\emptyset, \uparrow)$ and the label is pairing $\downarrow$, but it is not contained in $(ab)_{=}c$, because they are $(\uparrow, \emptyset)$ and not pairing $\downarrow$.

Such an adaptation would work, but it has space issues.

First, the normalisation step can cause an exponential blow-up if nested simultaneously opened or closed equalities are combined with $\cup$ operation. So, a PSPACE algorithm should not apply the rules above, but deal with such situations on the fly: e.g. we may allow a pebble in A to pass through an equality opening, but only with a condition that this equality will be closed together with the previous one.

Second, and more serious problem is that even if the depth of each stack is bounded by the depth of the equality tests nesting in A', the number of different stacks is exponential. In fact, there are examples where exponentially big set of pebbles with different stacks are on the same state in A at some point of a run. However, such a set is *never arbitrary*, and lots of information in the stacks can be shared: if a stack can be seen as a unary tree, then every set of such trees which appears on a run can be represented as a *dag*, whose size is polynomial.

Careful exploiting of the ideas above leads us to the desired PSPACE algorithm for checking containment of RQDs. □

5. LANGUAGES WITH INVERSE

RQMs and RQDs are recent, but established extensions of RPQs which manage data values. However, as noted in [13], RPQs by themselves lack a very natural construction for navigation through the structure of graphs—namely, the *inverse* operator. Indeed, consider for example a genealogy graph over a single *parent* label, such as the one presented in Figure 3. We assume that nodes represent people and data values are their names. A natural query over this graph, which does not deal with data values, would be to ask for all pairs of siblings. This, however, is clearly not expressible as an RPQ. On the other hand, it can be written as $parent^{-}parent$, where '$^{-}$' is the inverse operator, which traverses edges *backwards*. This query will retrieve e.g. (v_2, v_4) from the graph in Figure 3, since these nodes have a common parent v_1.

The class of queries enriching RPQs with inverse, called *2-way RPQs*, or *2RPQs* for short, was introduced in [13], where it was shown that even with this extension query evaluation remains the same as for RPQs (namely NLOGSPACE-complete). Moreover, in [14] the authors also show that query containment is as efficient as for plain RPQs (namely PSPACE-complete).

In this section we consider the extensions of RQMs and RQDs with the inverse operator, called *2RQMs* and *2RQDs* respectively. As far as we are aware, these languages have never been formally investigated, but we believe that they are natural and intuitive formalisms for querying data

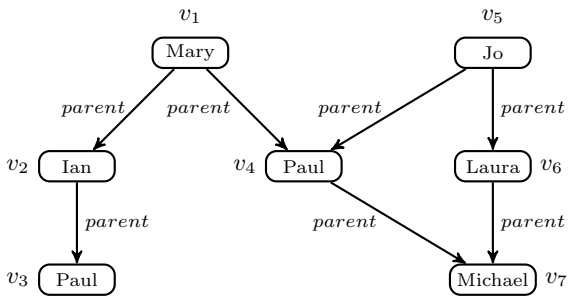

Figure 3: A genealogy database over the *parent* label.

graphs. For example, one query of interest in our genealogy database might be to retrieve all pairs of (blood) relatives with the same name. This can be easily done by the means of 2RQD $((parent^{-})^{+}parent^{+})_{=}$, which checks that two people have a common ancestor and ensures that they also have the same name. For example the pair (v_3, v_4) is an answer to this query in our sample graph.

The main focus of this paper is query containment. But since we introduce the languages of 2RQMs and 2RQDs here, after the formal definitions we first explore the complexity of query evaluation, and afterwards proceed to containment.

5.1 Definition and Evaluation of 2RQMs and 2RQDs

The syntax of 2RQMs results from adding the inverse operator to RQMs. The similar holds for 2RQDs.

DEFINITION 5.1. *A* 2-way regular query with memory*, or* 2RQM*, over alphabet of labels* Σ *and registers* X*, is an expression satisfying the grammar (1) in Definition 3.1 extended with the alternative* a^{-}*, where* a *ranges over* Σ*. A* 2-way regular query with data tests*, or* 2RQD*, over labels* Σ *is an expression satisfying (2) in Definition 4.1 extended with* a^{-}*.*

By this definition, 2RQDs restrict 2RQMs in the same way as RQDs restrict RQMs. The semantics of these languages extends their one-way analogs in the intuitive way. For 2RQMs, given a data graph $G = \langle V, E, \rho \rangle$, the function $\mathcal{H}^G$ extends the definition from Table 1 to the inverse construction as follows:

$$\mathcal{H}^G(a^{-}) \quad = \quad \{((v', \lambda), (v, \lambda)) \mid (v, a, v') \in E\}.$$

Then, the *evaluation* $[\![e]\!]^G$ of an 2RQM e over a data graph G stays the same as for RQMs.

Similarly, the *evaluation* $[\![e]\!]^G$ of a 2RQD e over a data graph $G = \langle V, E, \rho \rangle$ is obtained by adding the following rule to Table 2:

$$[\![a^{-}]\!]^G \quad = \quad \{(v', v) \mid (v, a, v') \in E\}.$$

As noted above, the complexity of 2RPQ evaluation is the same as for plain RPQs. Next we show that the same also holds for RQMs and RQDs with their two-way variants.

PROPOSITION 5.2. *The problem of deciding whether a pair of nodes belongs to* $[\![e]\!]^G$ *for a 2RQM* e *and a data graph* G *is* PSPACE*-complete. The same problem is in* PTIME *if* e *is a 2RQD. If we assume that* e *is fixed the problem becomes* NLOGSPACE*-comlete for both 2RQMs and 2RQDs.*

The proof of this proposition follows from the evaluation algorithms for RQMs and RQDs described in [32], and the observation that such two-way query can be viewed as an ordinary one-way query over the extended alphabet $\Sigma' = \Sigma \cup \{a^- \mid a \in \Sigma\}$. Then a pair (v, v') is an answer of e over a graph G if and only if it is an answer of e when viewed as a one-way query over the extended graph G' (over the alphabet Σ') which contains the edge (v', a^-, v) edge for each edge (v, a, v') in G.

5.2 Containment of 2RQMs and 2RQDs

The classic result by Calvanese et al. [14] states that one can add the inverse operator to RPQs and maintain not only the same complexity of query evaluation, but also the same complexity of query containment. The proposition above gives a hope that the inverse functionality will not affect the complexity of containment of 2RQMs and 2RQDs as well. Of course, by the results of the previous sections, containment is undecidable when full languages are considered. Unfortunately, as we show next, decidability for positive RQMs does not propagate to their two-way variant.

The class of *positive 2RQMs* is defined as the subclass of 2RQMs that use only conditions built from atoms of the form $x^=$ (but not $x^{\neq}$). Note that for 2RQMs we can no longer use language containment to check for query containment [14]. Indeed, it might be tempting to do the same as we did for Proposition 5.2, and reduce containment checking of two-way queries to containment of the same queries, but viewed as one-way queries over the extended alphabet containing symbols a^- for each $a \in \Sigma$. However, this does not imply that queries are contained, because labels of the form a^- can also symbolise going backwards (for example, the 2RPQ a is contained in the 2RPQ aa^-a, but the containment does not hold when the queries viewed as regular expressions over the extended alphabet). This leads to the following announced result.

THEOREM 5.3. *The problem* CONTAINMENT *(*positive 2RQMs*) is undecidable.*

PROOF SKETCH. The proof is by reduction from the emptiness problem of stateless multihead automata, known to be undecidable [45]. Two-way register automata are known to be able to simulate stateless multihead automata [37], and the same can be shown for 2RQMs. However, such simulation requires both equalities and inequalities, so the proof does not follow directly from this fact.

The idea of our novel reduction is to simulate only the accepting runs of a particular stateless multihead automaton $\mathcal{A}$. In other words, we define positive 2RQMs e_1 and e_2 such that $\mathcal{A}$ accepts no words if and only if $e_1 \subseteq e_2$. In our coding, a witness G_w for $e_1 \not\subseteq e_2$ represents a word belonging to $\mathcal{A}$. □

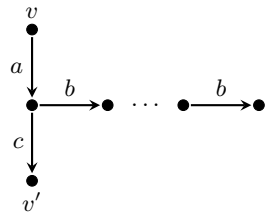

Figure 4: A pattern for GXPath query $a[\langle b^+ \rangle]c$.

This negative result comes as a surprise, and it poses a question on whether the containment problem is at least decidable for positive 2RQDs. We leave this question for future work.

6. GRAPH XPATH

As we saw in the previous section, 2RQMs and 2RQDs extend RPQs with the constructs for data values comparisons and also with an additional navigational feature. The language of *graph XPath*, or GXPath for short, which was introduced in [31] as an adaptation of the widely used XML query language XPath to the graph setting, goes further in this direction, extending the classes considered above with even more elaborate navigational tools, including a branching operator that allows one to check conditions along more than one path in the graph. For example, the GXPath query $a[\langle b^+ \rangle]c$ retrieves all pairs (v, v') of nodes connected by a path labelled ac, such that the intermediate node on this path has an outgoing sequence of b-labelled edges. The end point of that sequence can be arbitrary, we are only interested in its existence. The pattern described by this query is illustrated in Figure 4.

One consequence of this gain in navigational expressiveness is that we cannot always go from graphs to words as before: for instance, there are GXPath queries which are satisfiable on graphs, but not on words (like the one above). It means that we cannot hope for anything like Propositions 3.5 and 4.3, because query containment no longer corresponds to containment of languages.

Contrary to 2RQMs and 2RQDs, static analysis aspects of GXPath were not previously studied even for purely navigational fragment $\mathsf{GXPath}_{\mathrm{reg}}$ that uses no data value comparisons. That is why we start by exploring the containment problem for this fragment, and only after it proceed to various extensions with data tests.

Before proceeding to the formal details, it is worth to note, that the aforementioned class $\mathsf{GXPath}_{\mathrm{reg}}$ essentially corresponds to the well studied formalism of *propositional dynamic logic*, or *PDL* [26], with negation on paths.

6.1 Syntax and Semantics of $\mathsf{GXPath}_{\mathrm{reg}}$

As in XPath, formulas of $\mathsf{GXPath}_{\mathrm{reg}}$ are divided into *path formulas*, returning pairs of nodes, and *node formulas*, returning single nodes. Since we are interested in extensions of RPQs (which are binary), we concentrate on path formulas, and node ones will play just an auxiliary role. The formulas are defined by mutual recursion as follows.

$$
\begin{array}{rcl}
[\![\top]\!]^G & = & \{v \mid v \in V\},\\
[\![\neg\varphi]\!]^G & = & V - [\![\varphi]\!]^G,\\
[\![\varphi \wedge \psi]\!]^G & = & [\![\varphi]\!]^G \cap [\![\psi]\!]^G,\\
[\![\varphi \vee \psi]\!]^G & = & [\![\varphi]\!]^G \cup [\![\psi]\!]^G,\\
[\![\langle\alpha\rangle]\!]^G & = & \{v \mid \exists v'\ (v,v') \in [\![\alpha]\!]^G\};
\end{array}
$$

$$
\begin{array}{rcl}
[\![\varepsilon]\!]^G & = & \{(v,v) \mid v \in V\},\\
[\![a]\!]^G & = & \{(v,v') \mid (v,a,v') \in E\},\\
[\![a^-]\!]^G & = & \{(v',v) \mid (v,a,v') \in E\},\\
[\![[\varphi]]\!]^G & = & \{(v,v) \in G \mid v \in [\![\varphi]\!]^G\},\\
[\![\alpha \cup \beta]\!]^G & = & [\![\alpha]\!]^G \cup [\![\beta]\!]^G,\\
[\![\alpha \cdot \beta]\!]^G & = & [\![\alpha]\!]^G \circ [\![\beta]\!]^G,\\
[\![\overline{\alpha}]\!]^G & = & V \times V - [\![\alpha]\!]^G,\\
[\![\alpha^+]\!]^G & \text{is the transitive closure of } & [\![\alpha]\!]^G.
\end{array}
$$

Table 3: The semantics of $\mathsf{GXPath}_{\mathrm{reg}}$. The symbol '$-$' stands for set-theoretic difference.

DEFINITION 6.1. *Node formulas of φ, ψ of $\mathsf{GXPath}_{\mathrm{reg}}$ and* path formulas *α, β are expressions satisfying the grammar*

$$
\begin{array}{rcl}
\varphi, \psi & := & \top \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \langle\alpha\rangle,\\
\alpha, \beta & := & \varepsilon \mid a \mid a^- \mid [\varphi] \mid \alpha \cup \beta \mid \alpha \cdot \beta \mid \overline{\alpha} \mid \alpha^+.
\end{array} \tag{3}
$$

Just by glancing the definition one immediately notices that $\mathsf{GXPath}_{\mathrm{reg}}$ is a formalism much richer in navigational properties than RPQs: it allows inverse traversal of edges (the a^- operator), non-existence of paths (the $\overline{\alpha}$ operator), and testing for existence of (boolean combinations of) paths starting from the current node (the $[\varphi]$ operator). The formal semantics with respect to a graph $G = \langle V, E\rangle$ is given in Table 3: a node formula φ defines the set $[\![\varphi]\!]^G$ of nodes, and a path formula α defines the set $[\![\alpha]\!]^G$ of pairs of nodes.

Since we are dealing with navigational queries the tree analogue of our language would be the regular fragment of XPath. Note that there negation over path formulas is usually not included in the syntax, as one can show that this class is closed under complementation of path expressions [34]. This, however, is not the case for GXPath as shown in [31], so complementation is added to preserve close connection between GXPath and first-order logic.

6.2 Containment of GXPath$_{\mathrm{reg}}$ Queries

Analysing the expressive power of $\mathsf{GXPath}_{\mathrm{reg}}$ reveals that this class of queries is equivalent to the extension of first order logic with three variables (FO^3) with the transitive closure operator [31]. It is well known that satisfiability of FO^3 formulas is undecidable over arbitrary (possibly infinite) graphs, and it is folklore to assume that this bound is maintained for finite graphs, which we study in this paper. Since containment is a more general problem, than satisfiability, we have the following theorem.

THEOREM 6.2. *The* CONTAINMENT *($\mathsf{GXPath}_{\mathrm{reg}}$) problem is undecidable.*

PROOF SKETCH. Since we could not find a formal proof of the aforementioned result about finite satisfiability of FO^3, we include a self contained proof in the appendix, as for all other theorems of this paper. The proof shows that even satisfiability problem for $\mathsf{GXPath}_{\mathrm{reg}}$ formulas is undecidable. To obtain this result we give a reduction from a variation of tiling problem from [25]. In particular we use the fact that the set $\mathcal{S}_{\mathrm{notiling}}$, of all finite sets of tiles that can not tile the positive plane, and the set $\mathcal{S}_{\mathrm{period}}$, of all finite sets of tiles that can tile the plane periodically, are recursively inseparable.

Following the ideas from [21], we then show how to construct, for each finite set of tiles $\mathcal{T}$, a $\mathsf{GXPath}_{\mathrm{reg}}$ node formula $\gamma_{\mathcal{T}}$ such that satisfiability of $\gamma_{\mathcal{T}}$ implies that $\mathcal{T}$ can tile the positive plane, while the fact that $\mathcal{T}$ can tile the plane periodically implies that $\gamma_{\mathcal{T}}$ is satisfiable. Note that this shows that the set $S = \{\varphi \mid \exists G \text{ s.t. } [\![\varphi]\!]^G \neq \emptyset\}$ contains the set $\{\gamma_{\mathcal{T}} \mid \mathcal{T} \in \mathcal{S}_{\mathrm{period}}\}$ and is disjoint from $\{\gamma_{\mathcal{T}} \mid \mathcal{T} \in \mathcal{S}_{\mathrm{notiling}}\}$. The fact that $\mathcal{S}_{\mathrm{notiling}}$ and $\mathcal{S}_{\mathrm{period}}$ are recursively inseparable then implies that S can not be recursive, so satisfiability, and thus containment, of $\mathsf{GXPath}_{\mathrm{reg}}$ queries is undecidable.

To define the formula $\gamma_{\mathcal{T}}$ we rely heavily on the fact that $\mathsf{GXPath}_{\mathrm{reg}}$ can force loops in a graph, thus allowing us to check that tiles are placed correctly and that the tiling can proceed from any point in the plane. □

By analysing the proof one can also observe that the usage of the transitive closure operator $^+$ is restricted to edge labels only. Thus, we actually show that the satisfiability problem is already undecidable for the fragment of $\mathsf{GXPath}_{\mathrm{reg}}$, called $\mathsf{GXPath}_{\mathrm{core}}$ by analogy with the core fragment of XPath, which allows only a^+ and $(a^-)^+$ instead of α^+ in the grammar for path queries in (3). Note, that $\mathsf{GXPath}_{\mathrm{core}}$ does not contain RPQs any more, and in fact these two classes are incomparable [31].

Due to the aforementioned connection to PDL, we have a result on satisfiability of PDL with negation over finite models.

COROLLARY 6.3. *The satisfiability problem for PDL with negation on paths is undecidable over finite models, even in the absence of propositional variables.*

In fact, by carefully examining the proof, one can check that the use of negation is quite limited and that we only use intersection and the fact that $\mathsf{GXPath}_{\mathrm{reg}}$ can define the set of all pairs of mutually different nodes via the expression $\overline{\varepsilon}$. We are hoping that further adaptations of the proof could lead to solving the well known open problem of finite satisfiablity for PDL formulas with intersection [22].

As in the previous sections, we have the following question: what are the restrictions on $\mathsf{GXPath}_{\mathrm{reg}}$ that make containment decidable? The most natural candidates are of course the ones that forbid negation. Since we have two forms of negation, one on node formulas and another on path formulas, we consider two positive subclasses of $\mathsf{GXPath}_{\mathrm{reg}}$.

DEFINITION 6.4. *The* positive $\mathsf{GXPath}_{\mathrm{reg}}$, *denoted* $\mathsf{GXPath}_{\mathrm{reg}}^{\mathrm{pos}}$, *does not allow node formulas of the form* $\neg\varphi$

Data comparisions	RQD	RQM	2RQD	2RQM	$\mathsf{GXPath}_{\mathrm{reg}}^{\mathrm{pos}}$	$\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$	$\mathsf{GXPath}_{\mathrm{reg}}$
none	PSPACE-c*		PSPACE-c*		PSPACE-c*	EXPTIME-c	und.
positive	PSPACE-c	EXPSPACE-c	?	und.	?	?	und.
full	und.	und.	und.	und.	und.	und.	und.

Table 4: Complexity of containment of data graph queries. Some classes have synonyms, not given for clarity: i.e. RQDs and RQMs with no data comparisons are RPQs. Results, known before, are marked with '*', "-c" stands for "complete".

nor path formulas of the form $\overline{\alpha}$ in the grammar (3) of $\mathsf{GXPath}_{\mathrm{reg}}$.

The path-positive $\mathsf{GXPath}_{\mathrm{reg}}$*, denoted* $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$*, does not allow $\overline{\alpha}$, but keeps $\neg\varphi$ in the grammar.*

Note that, as opposed to the classes from previous sections, the word "positive" refers here to restrictions of navigational properties, and not of data manipulation abilities.

A PSPACE upper bound for complexity of containment problem for $\mathsf{GXPath}_{\mathrm{reg}}^{\mathrm{pos}}$ queries was shown in [39]. Hence, this complexity is the same as for RPQs. Exploiting connections with PDL, we obtain the following result for the second, bigger class defined above.

THEOREM 6.5. *The decision problem* CONTAINMENT *(*$\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$*) is* EXPTIME*-complete.*

Note that this result gives us an upper bound of containment for *path-positive* $\mathsf{GXPath}_{\mathrm{core}}$, i.e. the intersection of $\mathsf{GXPath}_{\mathrm{core}}$ and $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$. We leave the precise bounds for *core* fragments for future work, as our focus in this paper is on queries extending RPQs.

6.3 Adding Data Values

There are two approaches to add data value comparisons to XPath. We consider the one which is in line with RQDs. The syntax of this new class $\mathsf{GXPath}_{\mathrm{reg}}(\sim)$ extends the grammar (3) of $\mathsf{GXPath}_{\mathrm{reg}}$ with path formulas of the form $\alpha_{=}$ and $\alpha_{\neq}$. The semantics over a data graph $G = \langle V, E, \rho\rangle$ enriches Table 3 in a way similar to semantics of RQDs:

$$\begin{aligned}[\![\alpha_{=}]\!]^G &= \{(v,v') \in [\![\alpha]\!]^G \mid \rho(v) = \rho(v')\},\\ [\![\alpha_{\neq}]\!]^G &= \{(v,v') \in [\![\alpha]\!]^G \mid \rho(v) \neq \rho(v')\}.\end{aligned}$$

Similarly to previous sections, we also consider subclasses $\mathsf{GXPath}_{\mathrm{reg}}^{\mathrm{pos}}(\sim)$ and $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}(\sim)$ of $\mathsf{GXPath}_{\mathrm{reg}}(\sim)$, the first of which does not allow node negations $\neg\varphi$ and path negations $\overline{\alpha}$, and the second one does not allow just path negations.

Another way to add data value tests would be to follow usual XPath and add node formulas $\langle\alpha = \beta\rangle$ to the syntax. The evaluation of such a formula contains all the nodes in the graph from which one can reach two nodes v' and v'' by following paths satisfying α and β respectively, such that $\rho(v') = \rho(v'')$. In [31] it was shown that such an extension of $\mathsf{GXPath}_{\mathrm{reg}}$ is strictly contained in the defined above $\mathsf{GXPath}_{\mathrm{reg}}(\sim)$.

Next we come to query containment for $\mathsf{GXPath}_{\mathrm{reg}}(\sim)$ and its fragments. However, it is shown in [31], that even $\mathsf{GXPath}_{\mathrm{reg}}^{\mathrm{pos}}(\sim)$, i.e. the smallest subclass defined above, contains the class of RQDs. That is why we have the following corollary of Theorem 4.4.

COROLLARY 6.6. *The problems*

- CONTAINMENT *(*$\mathsf{GXPath}_{\mathrm{reg}}^{\mathrm{pos}}(\sim)$*),*
- CONTAINMENT *(*$\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}(\sim)$*) and*
- CONTAINMENT *(*$\mathsf{GXPath}_{\mathrm{reg}}(\sim)$*)*

are undecidable.

The next step in the search for decidable fragments of GXPath would be to restrict data tests to equality tests of the form $\alpha_{=}$ only (i.e. forbid the form $\alpha_{\neq}$). We did such a restriction for RQDs and RQMs before. From Theorem 6.2 we already know that containment for $\mathsf{GXPath}_{\mathrm{reg}}(\sim)$ with such restriction is undecidable. However, results for similar fragments of RQDs give some hope that containment for $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}(\sim)$ and $\mathsf{GXPath}_{\mathrm{reg}}^{\mathrm{pos}}(\sim)$ with such restrictions might be decidable. In future work we would like to extend our research in this direction, as well as study what happens in core fragments, where one might even be allowed to use inequality tests and still retain decidability of basic static analysis tasks.

7. CONCLUSIONS AND FUTURE WORK

After conducting a detailed study of query containment for main classes of queries for graphs with data, we conclude that the picture here is quite different from the one for traditional navigational languages. In particular, there is a sharp contrast between RPQs or CRPQs, where containment is decidable, and any of the known extension of RPQs that handle data values. Undecidability for the class of RQMs comes as not a surprise, due to high complexity of query evaluation and powerful data manipulation mechanism, but we have seen that even classes with good query evaluation properties can have undecidable containment.

The presence of inequality tests seems to be one of the major detractors here, although the ability to define complex navigational patterns can lead to undecidability as well. Thus, it seems that to obtain decidable fragments one has to limit attention to purely positive subclasses. The situation further complicates in the presence of inverse operator. We summarise all of the results in Table 4.

All of this shows that, although most of graph query languages are already well established, there is still some fine tuning needed to define languages with desirable static analysis properties. In particular, we would like to fully understand the containment problem for all fragments of GXPath. Some results in previous sections give us hope that decidability could be obtained for positive fragments using only equality tests and for core fragments, which we did not consider here.

In particular, the decidability of containment is open for positive 2RQDs; and the equalities-only versions of $\mathsf{GXPath}_{\text{reg}}^{\text{pos}}$ and $\mathsf{GXPath}_{\text{reg}}^{\text{path-pos}}$. The expressive power of these classes is tightly related: positive $\mathsf{GXPath}_{\text{reg}}^{\text{pos}}$ is obtained by adding the test operator $[\varphi]$ to positive 2RQDs, and of course positive $\mathsf{GXPath}_{\text{reg}}^{\text{path-pos}}$ contains positive $\mathsf{GXPath}_{\text{reg}}^{\text{pos}}$. An undecidability result for positive 2RQDs would settle the question for all three classes, but we conjecture that the the decidability frontier is somewhere between positive $\mathsf{GXPath}_{\text{reg}}^{\text{pos}}$ and positive $\mathsf{GXPath}_{\text{reg}}^{\text{path-pos}}$. Another approach is to consider only standard XPath-like tests of the form $\langle \alpha = \beta \rangle$, which were shown to be weaker than the equality tests used here [31]. Finally, it could be interesting to look at graph queries over various description logics, where some results are known, but only about 2RPQs and C2RPQs [10].

8. REFERENCES

[1] S. Abiteboul, R. Hull, V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] R. Angles, C. Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1), 2008.

[3] P. Barceló, L. Libkin, A.W. Lin, P. Wood. Expressive languages for path queries over graph-structured data. *ACM TODS* 38(4) (2012).

[4] P. Barceló, L. Libkin, J. Reutter. Querying graph patterns. In *PODS'11*, pages 199–210.

[5] P. Barceló, J. Pérez, J. L. Reutter. Relative expressiveness of nested regular expressions. In *AMW'12*, pages 180–195.

[6] P. Barceló, J. Reutter, L. Libkin. Parameterized regular expressions and their languages.. *TCS* 474: 21–45 (2013).

[7] P. Barceló. Querying Graph Databases. In *PODS'13*.

[8] M. Benedikt, C. Koch. XPath leashed. *ACM Computing Surveys (CSUR)*, 41(1) (2008).

[9] M. Benedikt, W. Fan, F. Geerts. XPath satisfiability in the presence of DTDs. In *J. ACM*, 55(2) (2008).

[10] M. Bienvenu, M. Ortiz, M. Šimkus. Conjunctive Regular Path Queries in Lightweight Description Logics. In *IJCAI*, 2013.

[11] E. Börger, E. Grädel, Y. Gurevich *The Classical Decision Problem.* Perspectives in Mathematical Logic, Springer, 2001.

[12] P. Buneman, S. B. Davidson, G. G. Hillebrand, D. Suciu A Query Language and Optimization Techniques for Unstructured Data. In *SIGMOD Conference 1996*, pages 505–516

[13] D. Calvanese, G. De Giacomo, M. Lenzerini, M.Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR'2000*, pages 176–185.

[14] D. Calvanese, G. De Giacomo, M. Lenzerini, M.Y. Vardi. Reasoning on regular path queries. *ACM SIGMOD Record*, 32(4):83–92, 2003.

[15] D. Calvanese, G. De Giacomo, M. Lenzerini, M.Y. Vardi. View-Based query answering and query containment over semistructured data. In *DBPL* 2001, pages 176–185.

[16] M. Consens, A. Mendelzon. GraphLog: a visual formalism for real life recursion. In *PODS'90*, pages 404–416.

[17] I. Cruz, A.O. Mendelzon, P. Wood. A graphical query language supporting recursion. In *SIGMOD'87*, pages 323–330.

[18] C. David, A. Gheerbrant, L. Libkin, W. Martens. Containment of pattern-based queries over data trees. *ICDT 2013*, pages 201–212.

[19] DEX query language. `http://www.sparsity-technologies.com/dex.php`.

[20] D. Florescu, A. Y. Levy, D. Suciu. Query Containment for Conjunctive Queries with Regular Expressions. *PODS'98*, pages 139–148.

[21] R. Goldblatt, M. Jackson. Well structured program equivalence is highly undecidable. *ACM Trans. Comput. Log.*, 13(3):26, 2012.

[22] S. Göller, M. Lohrey, C. Lutz. PDL with intersection and converse: satisfiability and infinite-state model checking. In *J. Symb. Log.*, 74(1): 279-314 (2009).

[23] The Gremlin graph traversal language. `http://gremlin.tinkerpop.com`.

[24] A. Gupta, I.S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. IEEE Data Eng. Bull., 18(2): 3–18, 1995.

[25] Y. Gurevich, I. Koryakov. Remarks on Berger's paper on the domino problem. In *Siberian Math. Journal*, 1972.

[26] D. Harel, D. Kozen, J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[27] S. Harris et al. *SPARQL 1.1 Query Language*. `http://www.w3.org/TR/sparql11-query`.

[28] M. Kaminski, N. Francez. Finite memory automata. *TCS*, 134(2):329–363, 1994.

[29] M. Lenzerini. Data integration: a theoretical perspective. In *PODS*, 2002.

[30] L. Libkin, J. L. Reutter, D. Vrgoč. TriAL for RDF: Adapting Graph Query Languages for RDF Data. In *PODS*, 2013.

[31] L. Libkin, W. Martens, D. Vrgoč. Querying graph databases with XPath. In *ICDT*, 2013.

[32] L. Libkin, D. Vrgoč. Regular path queries on graphs with data. In *ICDT'12*, pages 74–85.

[33] L. Libkin, D. Vrgoč. Regular expressions for data words. *LPAR'12*, pages 274–288.

[34] M. Marx. Conditional XPath. *ACM Trans. Database Syst.* 30(4): 929–959 (2005).

[35] G. Miklau, D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1): 2-45 (2004).

[36] The Neo4j Manual. `http://docs.neo4j.org`.

[37] F. Neven, T. Schwentick, V. Vianu. Finite state machines for strings over infinite alphabets. *ACM TOCL* 5(3): 403–435 (2004).

[38] J. Pérez, M. Arenas, C. Gutierrez. nSPARQL: A navigational language for RDF. *J. Web Sem.*, 8(4):255–270, 2010.

[39] J. L. Reutter. Containment of Nested Regular Expressions. CoRR abs/1304.2637 , (2013).

[40] L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL'06*, pages 41-57.

[41] T. Schwentick. XPath query containment. *ACM SIGMOD Record*, 33(1):101–109, 2004.

[42] A. Tal. *Decidability of Inclusion for Unification Based Automata*. M.Sc. thesis (in Hebrew), Technion, 1999.

[43] B. ten Cate, C. Lutz. The complexity of query containment in expressive fragments of XPath 2.0. In *J. ACM*, 56(6), 2009.

[44] P. Wood. Query languages for graph databases. *Sigmod Record*, 41(1):50–60, 2012.

[45] L. Yang, Z. Dang, O. H. Ibarra. On stateless automata and P systems. In *International Journal of Foundations of Computer Science*, 19(05), 1259–1276, 2008.