View-Based Query Answering and Query Containment over Semistructured Data

Diego Calvanese[1], Giuseppe De Giacomo[1], Maurizio Lenzerini[1], and Moshe Y. Vardi[2]

[1] Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
`lastname@dis.uniroma1.it`
[2] Department of Computer Science, Rice University, P.O. Box 1892
Houston, TX 77251-1892, U.S.A.
`vardi@cs.rice.edu`

Abstract. The basic querying mechanism over semistructured data, namely regular path queries, asks for all pairs of objects that are connected by a path conforming to a regular expression. We consider conjunctive two-way regular path queries (C2RPQ$_c$'s), which extend regular path queries with two features. First, they add the inverse operator, which allows for expressing navigations in the database that traverse the edges both backward and forward. Second, they allow for using conjunctions of atoms, where each atom specifies that a regular path query with inverse holds between two terms, where each term is either a variable or a constant. For such queries we address the problem of view-based query answering, which amounts to computing the result of a query only on the basis of a set of views. More specifically, we present the following results: (1) We exhibit a mutual reduction between query containment and the recognition problem for view-based query answering for C2RPQ$_c$'s, i.e., checking whether a given tuple is in the certain answer to a query. Based on such a result, we can show that the problem of view-based query answering for C2RPQ$_c$'s is EXPSPACE-complete. (2) By exploiting techniques based on alternating two-way automata we show that for the restricted class of tree two-way regular path queries (in which the links between variables form a tree), query containment and view-based query answering are, rather surprisingly, in PSPACE (and hence, PSPACE-complete). (3) We present a technique to obtain view-based query answering algorithms that compute the whole set of tuples in the certain answer, instead of requiring to check each tuple separately. The technique is parametric wrt the query language, and can be applied both to C2RPQ$_c$'s and to tree-queries.

1 Introduction

Semistructured data are usually modeled as labeled graphs, and methods for extracting information from semistructured data incorporate special querying mechanisms that are not common in traditional database systems [1]. One such basic mechanism is the one of regular path queries (RPQs), which retrieve all pairs of nodes in the graph connected by a path conforming to a regular expression [11, 3]. Regular expressions provide a (limited) form of recursion, which is used in regular path queries to flexibly navigate the database graph.

In order to increase the expressiveness of RPQs, in this paper we consider two basic additions. First, we add to RPQs the inverse operator, which allows for expressing navigations in the database that traverse the edges both backward and forward. Second, we extend RPQs with the possibility of using conjunctions of atoms, where each atom specifies that one regular path query with inverse holds between two terms, where each term is either a variable or a constant. Notably, several authors argue that these kinds of extensions are essential for making RPQs useful in real settings (see for example [10, 11, 43]). The resulting queries will be called *conjunctive two-way regular path queries* (C2RPQ$_c$'s), and capture the core of virtually all query languages for semistructured data [11, 3, 32], including the ones for XML [29, 19]. In this paper, we consider also a restricted form of C2RPQ$_c$'s, called *tree two-way regular path queries* (T2RPQ$_c$'s), in which the body of the query has the structure of a tree, if we view each variable as a node and each atom in the query as an edge. T2RPQ$_c$'s are a generalization of several subclasses of queries, including generalized path expressions [11, 3] and branching path queries [42, 44], which have been studied extensively both in semistructured and XML data. Notably, XPath expressions [26] can be captured by T2RPQ$_c$'s in which the transitive closure operator is used in a limited way.

Semistructured data pose challenging problems to the research on databases [56]. Among them, reasoning on queries and views is expected to be particularly difficult, due to the presence of recursion in RPQs. For example, containment of Datalog queries with no limitations on recursion, is undecidable [48]. Here, we concentrate on two reasoning services involving queries that are relevant in database management. The first one is checking *containment*, i.e., verifying whether, for every database, one query yields a subset of the result of another one. The second one is *view-based query answering*, which amounts to computing the answer to a query having information only on the extension of a set of views. Both problems are crucial in several contexts, such as information integration [51], data warehousing [55], query optimization [23], mobile computing [6], and maintaining physical data independence [50]. Indeed, they have been investigated and largely solved for relational databases. Instead, for the case of semistructured data and XML, the above problems are largely unexplored and fundamental results are still missing [54].

Most of the results on query containment concern relational conjunctive queries and their extensions [22, 37, 52, 46, 24, 52]. In particular, [22] shows that containment of conjunctive queries is NP-complete. One interesting special case for which containment is tractable is when the right-hand side query has bounded treewidth [25], and, in particular, when it is acyclic. Other papers consider the case of conjunctive query containment in the presence of various types of constraints [5, 28, 20, 40, 41, 12]. Conjunctive RPQs without inverse have been studied in [33], where an EXPSPACE algorithm for query containment in this class is presented. In [16], it is shown that containment for conjunctive two-way regular path queries without constants is EXPSPACE-complete. The complexity of query containment for conjunctive RPQs has been studied in [30] under various restrictions on the allowed classes of regular expressions.

View-based query answering has been investigated in [39, 45] for the case of conjunctive queries (with or without arithmetic comparisons), in [4] for disjunctive views, in [49, 27, 35] for queries with aggregates, in [31] for recursive queries and nonrecursive views, and in [13, 7] for queries expressed in Description Logics. Comprehensive frameworks for view-based query answering, as well as several interesting results for various query languages, are presented in [34, 2].

Recently view-based query processing has been studied for the case of RPQs. It has been shown that computing the RPQ which is the maximal rewriting of an RPQ wrt a set of RPQ views can be done in 2EXPTIME, and verifying the existence of a nonempty rewriting is EXPSPACE-complete [14]. View-based query answering for RPQs is PSPACE-complete in combined complexity and coNP-complete in data complexity [15]. In [17] the above results on view-based query rewriting and query answering have been extended to 2RPQs, i.e., RPQs extended with the inverse operator. The relationship between view-based query answering and query rewriting has been studied in [18].

None of the above results apply to C2RPQ$_c$'s or T2RPQ$_c$'s. Thus, decidability and complexity of query containment and view-based query answering is still an open problem for such queries.

Our goal is to devise techniques and characterize the computational complexity of the two problems for the two classes of queries. In particular, we present the following main contributions:

1. We exhibit a mutual reduction between query containment and view-based query answering for the case of C2RPQ$_c$'s (see Section 3). The reduction applies also to the case where the query in the right-hand side of the containment, and, respectively, the query to be answered using the views, is a T2RPQ$_c$.
2. Based on the technique presented in [16], we devise in Section 4 an EXPSPACE algorithm for view-based query answering for C2RPQ$_c$'s, thus showing that the problem is EXPSPACE-complete. By virtue of the reduction illustrated in Section 3, the same complexity result holds for query containment. This is the first result showing that view-based query answering is decidable in the case where both the query and the views are expressed in a query language for semistructured data with at least the power of conjunctive queries with constants.
3. By exploiting techniques based on alternating two-way automata we show in Section 5 that, in the case where the views are C2RPQ$_c$'s and the query to be answered is a T2RPQ$_c$, view-based query answering is PSPACE-complete. This represents a provable exponential improvement wrt the case of C2RPQ$_c$'s. The same bound holds for query containment when the right-hand side query is a T2RPQ$_c$. These results are rather surprising in view of the fact that, while tree automata seem the natural formal tool for tree-queries, containment of such automata is EXPTIME-complete [47].
4. View-based query answering has been generally tackled in the form of the so-called *recognition problem*: the input includes a tuple and the goal is to check whether such a tuple is a certain answer to the query, i.e., whether the tuple satisfies the query in every database coherent with the views. However, traditional query answering consists of retrieving *all* tuples satisfying the query. A further question addressed in this paper is whether we can devise a method to characterize the whole set of certain answers to a query, rather than just solving the recognition problem tuple by tuple. We present a solution to this problem, both for the case of C2RPQ$_c$'s (Section 4.4) and for T2RPQ$_c$'s (Section 5.4), by illustrating how to compute a representation of the whole set of certain answers. The method computes such a representation with the same computational complexity as the corresponding methods for checking whether a tuple is a certain answer. To the best of our knowledge, this is the first technique that deals with view-based query answering rather than checking if a tuple is a certain answer.

The result on T2RPQ$_c$'s shows that C2RPQ$_c$'s exhibit a behavior analogous to the case of ordinary conjunctive queries: cycles in the right-hand side query constitute a source of complexity for containment checking [57]. Observe that it is the query structure and not the database structure that determines the complexity, since, containment (and hence query answering) of conjunctions of RPQs is already EXPSPACE-hard over linear databases [16].

In the next section we introduce all concepts and definitions used in the subsequent sections. Section 6 concludes the paper, by pointing out several possible extensions of our work.

2 Databases and Queries

We consider a *semistructured database* (DB) $\mathcal{B}$ as an edge-labeled graph $(\mathcal{D}, \mathcal{E})$, where $\mathcal{D}$ is the set of nodes (representing objects) and $\mathcal{E}$ is the set of edges (representing binary relations) labeled with elements of an alphabet Δ[1]. We denote an edge from x to y labeled by p with $x \stackrel{p}{\to} y$. We use constants as names for nodes, and we impose the *unique name assumption*, i.e., we disallow two constants to denote the same node. In the following, we do not distinguish between a node and the constant naming it.

The basic querying mechanism on a DB is that of *regular path queries* (RPQs). An RPQ R is expressed as a regular expression or a finite automaton, and computes the set of pairs of nodes of the DB connected by a path that conforms to the regular language $L(R)$ defined by R. We consider queries that extend regular path queries with both the inverse operator, and the possibility of using conjunctions, variables, and constants.

Formally, let $\Sigma = \Delta \cup \{p^- \mid p \in \Delta\}$ be the alphabet including a new symbol p^- for each p in Δ. Intuitively, p^- denotes the inverse of the binary relation p. If $r \in \Sigma$, then we use r^- to denote the *inverse* of r, i.e., if r is p, then r^- is p^-, and if r is p^-, then r^- is p. *Two-way regular path queries* (2RPQs) are expressed by means of regular expressions or finite automata over Σ [16, 17]. Thus, in contrast with RPQs, 2RPQs may use also the inverse p^- of p, for each $p \in \Delta$. When evaluated over a DB $\mathcal{B}$, a 2RPQ R computes the set $ans(R, \mathcal{B})$ of pairs of nodes connected by a semipath that conforms to the regular language $L(R)$ defined by R. A *semipath* in $\mathcal{B}$ from x to y is a sequence of the form $(y_1, r_1, y_2, r_2, y_3, \ldots, y_q, r_q, y_{q+1})$, where $q \geq 0$, $y_1 = x$, $y_{q+1} = y$, and for each $i \in \{1, \ldots, q\}$, either $y_i \stackrel{r_i}{\to} y_{i+1}$ or $y_{i+1} \stackrel{r_i^-}{\to} y_i$ is in $\mathcal{B}$. The semipath *conforms to* R if $r_1 \cdots r_q \in L(R)$. The semipath is *simple* if each y_i, for $i \in \{2, \ldots, q\}$, is a node that does not occur elsewhere in the semipath.

Finally, we add to 2RPQs the possibility of using conjunctions of atoms, where each atom specifies that a regular path query with inverse holds between two terms, where a term is either a variable or a constant. More precisely, a *conjunctive two-way regular path query with constants* (C2RPQ$_c$) Q is a formula of the form

$$Q(x_1, \ldots, x_n) \leftarrow y_1\, E_1\, y_2 \wedge \cdots \wedge y_{2m-1}\, E_m\, y_{2m}$$

where $x_1, \ldots, x_n$ are variables, called *distinguished variables*, $y_1, \ldots, y_{2m}$ are either variables or constants, and $E_1, \ldots, E_m$ are 2RPQs.

We consider also a restricted form of C2RPQ$_c$'s, called *tree two-way regular path queries* (T2RPQ$_c$'s). In a T2RPQ$_c$, the body of the query has the structure of a tree, if we view each variable and each constant as a node and each atom $y\, E\, y'$ as an edge from y to y', and consider different occurrences of constants as different nodes of the tree.

In the following we assume that the variables in the head of a query are pairwise distinct. This assumption can be made without loss of generality, since, if a variable x occurs twice in the head, we can replace one of the occurrences by a fresh variable y and introduce the (equality) atom $x\, \varepsilon\, y$ in the body. Also, we assume that each distinguished variable occurs among $y_1, \ldots, y_{2m}$, since we can always add to the body an atom $x_i\, \varepsilon\, x_i$. For the same reason we can avoid constants in the head of a query. Notice that adding such atoms to the body does not destroy the tree structure of a T2RPQ$_c$.

[1] All the techniques presented in this paper can be adapted to the case of *rooted DBs*, i.e., to the case where the DB is an edge-labeled rooted graph (see Section 6), and to the case where nodes are labeled.

The *answer set* $ans(Q,\mathcal{B})$ to a C2RPQ$_c$ Q over a DB $\mathcal{B} = (\mathcal{D},\mathcal{E})$ is the set of tuples $(d_1,\ldots,d_n)$ of nodes of $\mathcal{B}$ such that there is a mapping σ from the variables and the constants of Q to $\mathcal{D}$ with

– $\sigma(x_i) = d_i$ for every distinguished variable x_i,
– $\sigma(c) = c$ for every constant c, and
– $(\sigma(y),\sigma(y')) \in ans(E,\mathcal{B})$ for every conjunct $y\,E\,y'$ in Q.

Given two C2RPQ$_c$'s Q_1 and Q_2, we say that Q_1 is *contained* in Q_2, written $Q_1 \subseteq Q_2$, if for every DB $\mathcal{B}$, $ans(Q_1,\mathcal{B}) \subseteq ans(Q_2,\mathcal{B})$. The problem of *query containment* (QC) is checking whether one query is contained in another one. Obviously, $Q_1 \not\subseteq Q_2$ iff there is a *counterexample* DB to $Q_1 \subseteq Q_2$, i.e., a DB $\mathcal{B}$ with a tuple in $ans(Q_1,\mathcal{B})$ and not in $ans(Q_2,\mathcal{B})$.

In view-based query answering, we consider a DB that is accessible only through a set of C2RPQ$_c$ views $\mathcal{V} = \{V_1,\ldots,V_k\}$. Each view V_i is characterized by its definition $def(V_i)$ in terms of a C2RPQ$_c$, and by its extension $ext(V_i)$ in terms of a set of constant tuples. We denote the set of constants appearing either in $ext(V_1) \cup \cdots \cup ext(V_k)$ or in $def(V_1) \cup \cdots \cup def(V_k)$ by $\mathcal{D}_V$. We say that a DB $\mathcal{B}$ is *consistent with a view* V_i if all tuples in its extension satisfy the definition of the view in $\mathcal{B}$, i.e., if $ext(V_i) \subseteq ans(def(V_i),\mathcal{B})$ [2]. We say that $\mathcal{B}$ is *consistent with* $\mathcal{V}$ if it is consistent with each view $V_i \in \mathcal{V}$.

Given a set $\mathcal{V} = \{V_1,\ldots,V_k\}$ of C2RPQ$_c$ views, a C2RPQ$_c$ Q of arity n whose constants are in $\mathcal{D}_V$, and an n-tuple $\vec{t}$ of constants[3] in $\mathcal{D}_V$, the problem of *view-based query answering* (QA) consists in deciding whether $\vec{t}$ is a *certain answer* to Q wrt $\mathcal{V}$, written $\vec{t} \in cert(Q,\mathcal{V})$, i.e., whether $\vec{t} \in ans(Q,\mathcal{B})$, for every DB $\mathcal{B}$ that is consistent with $\mathcal{V}$. Given a set $\mathcal{V} = \{V_1,\ldots,V_k\}$ of C2RPQ$_c$ views, and a C2RPQ$_c$ Q of arity n, the problem of *computing the set of certain answers* consists in characterizing all n-tuples of constants in $\mathcal{D}_V$ that are certain answers to Q wrt $\mathcal{V}$. As for the computational complexity of view-based query answering, we concentrate on *combined complexity*, i.e., we measure the complexity with respect to the size of the query, the view expressions, and the data in the view extensions.

3 Relationship between QA and QC

There is a strong connection between QA and QC. In particular, [2] discusses mutual reductions between the two problems (Theorem 4.1 in [2]). Next we show that such reductions can be adapted to our case.

For the reduction from QA to QC, consider an instance of QA with C2RPQ$_c$ views $\mathcal{V} = \{V_1,\ldots,V_k\}$, definitions $def(V_i)$ and extensions $ext(V_i)$, for $i \in [1..k]$, where we ask whether the tuple $(c_1,\ldots,c_n)$ is in the certain answer to the C2RPQ$_c$ Q of arity n. From such an instance, we construct a C2RPQ$_c$ $Q_\mathcal{V}$ such that $Q_\mathcal{V} \subseteq Q$ iff $(c_1,\ldots,c_n) \in cert(Q,\mathcal{V})$. We define $Q_\mathcal{V}$ as follows:

$$Q_\mathcal{V}(x_1,\ldots,x_n) \;\leftarrow\; x_1 \varepsilon c_1 \wedge \cdots \wedge x_n \varepsilon c_n \wedge \bigwedge_h \alpha_h$$

where $x_i \varepsilon c_i$ denotes an equality between the distinguished variable x_i and the constant c_i, and we have one α_h for each view V_i (of arity n_i) and each tuple $(d_1,\ldots,d_{n_i})$ in $ext(V_i)$. Such an

[2] As often done, we assume views to be *sound*, but not necessarily *complete* [2, 15]

[3] We use $\vec{t}$ to denote tuples of constants of the appropriate arity, and $\vec{t}[i]$ to denote the i-th component of tuple $\vec{t}$.

α_h is obtained from the body of $def(V_i)$ by replacing the distinguished variables $x_1, \ldots, x_{n_i}$ respectively by $(d_1, \ldots, d_{n_i})$ and by replacing the non-distinguished variables with fresh ones. Note that, if the query Q in QA is a T2RPQ$_c$, then we obtain an instance of QC in which the right-hand side query is a T2RPQ$_c$.

By using the same line of reasoning as in the proof of Theorem 4.1 in [2] we get:

Theorem 1. *Let $Q_\mathcal{V}$ be defined as above. Then $(c_1, \ldots, c_n) \in cert(Q, \mathcal{V})$ iff $Q_\mathcal{V} \subseteq Q$.*

The size of the QC instance obtained as specified above is polynomial with respect to the size of the QA instance. Therefore, we can conclude that QA is polynomially reducible to QC.

For the reduction from QC to QA, consider an instance of QC asking whether $Q_1 \subseteq Q_2$, where $Q_1(\vec{z}) \leftarrow \beta_1(\vec{z}, \vec{y}_1)$ and $Q_2(\vec{z}) \leftarrow \beta_2(\vec{z}, \vec{y}_2)$. From such an instance of QC, we construct an instance of QA as follows. We have only one view V_{Q_1}, whose extension is constituted by a single tuple $(c_1, \ldots, c_n)$ with the constants $c_1, \ldots, c_n$ not occurring in Q_1 and Q_2, and whose definition is:

$$V_{Q_1}(x_1, \ldots, x_n) \;\leftarrow\; x_1 \varepsilon c_1 \wedge \cdots \wedge x_n \varepsilon c_n \;\wedge\; \beta_1(\vec{z}, \vec{y}) \;\wedge\; \bigwedge_{z_i \in \vec{z}} z_i \, p_i^{new} \, c_1$$

where $x_1, \ldots, x_n$ are fresh variables, $\beta_1(\vec{z}, \vec{y})$ is the body of Q_1, and p_i^{new} are new symbols that do not appear in the alphabet of Q_1 and Q_2. The query we want to answer with the view is:

$$Q_{Q_2}(x_1, \ldots, x_n) \;\leftarrow\; x_1 \varepsilon c_1 \wedge \cdots \wedge x_n \varepsilon c_n \;\wedge\; \beta_2(\vec{z}, \vec{y}) \;\wedge\; \bigwedge_{z_i \in \vec{z}} z_i \, p_i^{new} \, c_1$$

where $x_1, \ldots, x_n$ are fresh variables, and $\beta_2(\vec{z}, \vec{y})$ is the body of Q_2. Finally, we ask whether $(c_1, \ldots, c_n) \in cert(Q_{Q_2}, \{V_{Q_1}\})$. Note that, if Q_2 is a T2RPQ$_c$, then we obtain an instance of QA in which the query to answer using the views is a T2RPQ$_c$.

Again, by using the same line of reasoning as in the proof of Theorem 4.1 in [2] we get:

Theorem 2. *Let V_{Q_1} and Q_{Q_2} be defined as above. Then $Q_1 \subseteq Q_2$ iff $(c_1, \ldots, c_n) \in cert(Q_{Q_2}, \{V_{Q_1}\})$.*

It is easy to see that the above construction provides a polynomial reduction from QC to QA.

4 QA and QC for C2RPQ$_c$'s

The technique for QA for C2RPQ$_c$'s we present is based on searching for a "counterexample DB" to the QA problem, i.e., a DB consistent with the views in which the given tuple does not satisfy the query. For our purposes, it is crucial to show that we can limit our search to counterexample DBs of a special form, called canonical DBs. Such DBs can be represented as finite words and we exploit finite word automata. In particular, we make use of standard one-way automata (1NFA) and two-way automata (2NFA) [36].

4.1 Canonical databases

Let $\mathcal{D}_V$ be the set of constants appearing in the extensions of the views. We introduce a set $\mathcal{D}_E$ of new constants, called *skolem constants*, one constant $y_{V,\vec{t}}$ for each view $V \in \mathcal{V}$, each non-distinguished variable y appearing in $def(V)$, and each tuple $\vec{t}$ in $ext(V)$. On such constants we do not enforce the unique name assumption, i.e., we may have two skolem constants denoting the same node. Let $\mathcal{D}_N = \mathcal{D}_V \cup \mathcal{D}_E$. In the following, to distinguish actual constants from skolem constants, we refer to the former as *proper constants*. We use the term *constants* for both proper and skolem constants.

Definition 1. *Given a set $\mathcal{V}$ of C2RPQ$_c$ views, a DB is called $\mathcal{V}$-*canonical *if it is composed of a set of simple semipaths $(\alpha, r_1, x_1, \ldots, x_{n-1}, r_n, \beta)$, one for each view $V \in \mathcal{V}$, each tuple $\vec{t} \in ext(V)$, and each atom of the form $y\,E\,y'$ in $def(V)$, where*

- *α is $\vec{t}[i]$ if y is the i-th distinguished variable in $def(V)$, α is y if y is a proper constant, and α is $y_{\vec{t},V}$ if y is a non-distinguished variable (similarly for β),*
- *$r_1 \cdots r_n \in L(E)$, and*
- *$x_1, \ldots, x_{n-1}$ are not in $\mathcal{D}_N$ and do not occur in any other semipath in the set.*

The following theorem, which can be shown similarly to an analogous theorem in [16], provides an important characterization of QA in terms of $\mathcal{V}$-canonical DBs.

Theorem 3. *Let Q be a C2RPQ$_c$ of arity n, $\mathcal{V}$ a set of views, and $\vec{t}$ a tuple of proper constants. If there exists a DB $\mathcal{B}$ that is consistent with $\mathcal{V}$ such that $\vec{t} \notin ans(Q, \mathcal{B})$, then there exists a $\mathcal{V}$-canonical DB $\mathcal{B}'$ that is consistent with $\mathcal{V}$ and such that $\vec{t} \notin ans(Q, \mathcal{B}')$.*

By Theorem 3, we can restrict the search for a counterexample DB to $\mathcal{V}$-canonical DBs only. The basic idea that allows us to exploit automata is that we can represent such DBs in a linearized form as special words, and use two-way automata to check that candidate counterexample DBs satisfy all required conditions [16, 17].

More precisely, each $\mathcal{V}$-canonical DB $\mathcal{B}$ can be represented as a word $w_{\mathcal{B}}$ over the alphabet $\Sigma \cup \mathcal{D}_N \cup \{\$\}$ of the form

$$\$d_1 w_1 d_2 \$ d_3 w_2 d_4 \$ \cdots \$ d_{2m-1} w_m d_{2m} \$$$

where m is some positive integer, $d_1, \ldots, d_{2m}$ are in $\mathcal{D}_N$, $w_i \in \Sigma^*$, and the \$ acts as a separator. In particular, each symbol d_i represents a node of $\mathcal{B}$, and $w_{\mathcal{B}}$ consists of one subword $d_{2i-1} w_i d_{2i}$, for each simple semipath conforming to w_i in $\mathcal{B}$, from the constant d_{2i-1} to the constant d_{2i}. Observe that we may have that $w_i = \varepsilon$, and in such a case d_{2i-1} and d_{2i} represent the same node, which is denoted by two different constants. Obviously, this can be the case only if at least one of d_{2i-1} and d_{2i} is a skolem constant, otherwise the unique name assumption is violated.

4.2 Automaton accepting canonical DBs

To verify whether $w_{\mathcal{B}}$ represents a DB $\mathcal{B}$ that is consistent with the views $\mathcal{V}$, we construct for each view $V \in \mathcal{V}$, each tuple $\vec{t} \in ext(V)$, and each atom R of the form $y\,E\,y'$ in $def(V)$, an 1NFA $A_{V,\vec{t},R}$ that accepts the language $\$ \cdot \alpha \cdot E \cdot \beta$, where α is $\vec{t}[i]$ if y is the i-th distinguished variable in $def(V)$, α is y if y is a proper constant, and α is $y_{V,\vec{t}}$ if y is a non-distinguished variable (similarly for β).

We now construct a 1NFA $A^-_{\mathcal{V}}$ that accepts the concatenation of the languages accepted by all 1NFAs $A_{V,\vec{t},R}$, in some fixed order (chosen arbitrarily), and of the language $(\$ \cdot \sum_{y \in \mathcal{D}_E, y' \in \mathcal{D}_N} (y \cdot y'))^* \cdot \$$, representing additional equalities between skolem constants in $\mathcal{D}_E$ and (skolem or proper) constants in $\mathcal{D}_N = \mathcal{D}_V \cup \mathcal{D}_E$. The 1NFA $A^-_{\mathcal{V}}$ accepts words representing candidate counterexample DBs, in which however the unique name assumption for proper constants is not enforced.

We then construct a 2NFA $A_{\neg UNA}$ that accepts words of the form above in which the unique name assumption is violated by equating two distinct proper constants (taking into account also transitivity and symmetry of equality). To do so we exploit the ability of 2NFAs to: (i) move on the word both forward and backward; (ii) "jump" from one position in the word with a certain symbol to any other position (either preceding or succeeding) with the same symbol [17].

$A_{\neg UNA}$ is the disjoint union of one 2NFA $A_{c,c'} = (\Sigma, S, s_0, \delta, \{c'\})$ for each pair of distinct proper constants c and c' in $\mathcal{D}_V$, where $S = \mathcal{D}_N \cup \{s_0, s^{\hookrightarrow}, s^{\hookleftarrow}\}$, and δ is defined as follows:

1. Starting from the initial state, the automaton searches for the symbol c, and when it finds it, it switches to the corresponding state. When it is in the final state c', it moves to the end of the word and accepts.

$$\begin{array}{rl} (c,0) \in \delta(s_0,c) & \\ (s_0,1) \in \delta(s_0,\ell) & \text{for each } \ell \in \Sigma \\ (c',1) \in \delta(c',\ell) & \text{for each } \ell \in \Sigma \end{array}$$

2. When in a state corresponding to a constant, the automaton moves either forward or backward. For each $d \in \mathcal{D}_N$, and for each $\ell \in \Sigma$

$$\begin{array}{r} (d,1) \in \delta(d,\ell) \\ (d,-1) \in \delta(d,\ell) \end{array}$$

3. When the automaton reaches a constant d_1 while it is in the state corresponding to d_1, and d_2 appears immediately to the right of d_1, the automaton may switch to the state corresponding to d_2. Similarly, if d_2 appears immediately to the left of d_1. For each $d_1, d_2 \in \mathcal{D}_N$

$$\begin{array}{r} (s^{\hookrightarrow},1) \in \delta(d_1,d_1) \\ (d_2,0) \in \delta(s^{\hookrightarrow},d_2) \\ (s^{\hookleftarrow},-1) \in \delta(d_1,d_1) \\ (d_2,0) \in \delta(s^{\hookleftarrow},d_2) \end{array}$$

Let $A_{\mathcal{V}}$ be the 1NFA obtained as the intersection of $A_{\mathcal{V}}^-$ and of the complement of $A_{\neg UNA}$. $A_{\mathcal{V}}$ accepts words representing $\mathcal{V}$-canonical DBs consistent with the views in which the unique name assumption is satisfied. We call a word w accepted by $A_{\mathcal{V}}$ a *$\mathcal{V}$-word*.

4.3 Automaton for a single tuple

Let $Q(\vec{x})$ be a C2RPQ$_c$ of arity n with distinguished variables $\vec{x}$. Adapting the construction in [16], we construct a 1NFA $A_{Q(\vec{t})}$ that, given a tuple $\vec{t}$ of proper constants, accepts the words representing DBs $\mathcal{B}$ such that $\vec{t} \in ans(Q(\vec{x}), \mathcal{B})$. To define $A_{Q(\vec{t})}$, we have to check for each 2RPQ in $Q(\vec{t})$, i.e., the query obtained by substituting the distinguished variables $\vec{x}$ with $\vec{t}$, whether the corresponding atom is satisfied. To check whether a pair of constants (d_1, d_2) explicitly appearing in a word representing a DB $\mathcal{B}$, is in the answer to a 2RPQ E, one can again exploit the ability of 2NFAs to move back and forth on a word and to jump from one position in the word representing a constant to any other position in the word representing the same constant.

Now, observe that, to check whether a certain tuple $\vec{t}$ of proper constants is in $ans(Q(\vec{x}), \mathcal{B})$, the distinguished variables $\vec{x}$ of $Q(\vec{x})$ are mapped to the constants of $\vec{t}$, which explicitly appear in the word representing the linearized DB, while the non-distinguished variables may be mapped to any node of the DB, i.e., to any symbol in the corresponding word. Hence, for each atom $d\,E\,d'$ of $Q(\vec{t})$ involving only proper constants assigned to the distinguished variables, we can directly construct a 2NFA that checks whether the atom is satisfied. On the other hand, for an atom involving a non-distinguished variable y, we need to explicitly represent where y is mapped to in the word, since we have to guarantee that all occurrences of y in distinct atoms are mapped to the same node. We represent such mappings of non-distinguished variables of $Q(\vec{t})$ as *annotations* of $\mathcal{V}$-words. More precisely, the $\mathcal{V}$-word $\$\ell_1 \cdots \ell_r\$$, with each symbol $\ell_i \neq \$$ annotated with a set γ_i of non-distinguished variables, is represented by the word $\$(\ell_1, \gamma_1) \cdots (\ell_r, \gamma_r)\$$ over the alphabet $((\Sigma \cup \mathcal{D}_N) \times 2^{\mathcal{Y}}) \cup \{\$\}$, where $\mathcal{Y}$ is the set of non-distinguished variables of $Q(\vec{x})$. The intended meaning is that the variables in γ_i are mapped in $\mathcal{B}_w$ to the node ℓ_i, if $\ell_i \in \mathcal{D}_N$, and are mapped to the target node of the edge corresponding to the occurrence of ℓ_i, if $\ell_i \in \Sigma$.

Given a word w' representing an annotated $\mathcal{V}$-word w, a 1NFA $A^{an}_{Q(\vec{t})}$ can check if each atom in $Q(\vec{t})$ is satisfied in the DB represented by w, given the annotation in w'. We first define the following automata:

- A 1NFA A_s that checks that for every non-distinguished variable y,
  - either y appears in the annotation of a single occurrence of a symbol in Σ, and it does not appear in the annotation of any other position in the word;
  - or y appears in the annotation of every occurrence of a symbol $d \in \mathcal{D}_N$, and it does not appear in the annotation of any other symbol different from d.
- A 1NFA $A_\$$ that checks that every occurrence in w of a symbol preceding a \$ is annotated in w' with the same set of variables as the symbol preceding it. This takes into account equalities introduced by subwords of the form $\$d_1 d_2\$$, as well as that the symbol a in a subword of the form $\cdots ad\$$ actually represents the target node of the a-edge, i.e., d itself.
- A 2NFA A_q that checks that each atom in $Q(\vec{t})$ is satisfied in $\mathcal{B}_w$, given the assignment to the non-distinguished variables represented by the annotation in w'. Such an automaton can be constructed by exploiting the ability of 2NFAs to move in both directions and jump on the word, as discussed above.

The 1NFA $A^{an}_{Q(\vec{t})}$ is constructed as the product of A_s, $A_\$$, and of the 1NFA equivalent to A_q. Next we define a 1NFA $A_{Q(\vec{t})}$ that simulates the guess of an annotation of a $\mathcal{V}$-word, and emulates the behaviour of $A^{an}_{Q(\vec{t})}$ on the resulting annotated word. The simulation of the guess and the emulation of $A^{an}_{Q(\vec{t})}$ can be obtained simply by constructing $A^{an}_{Q(\vec{t})}$ and then projecting out the annotation from the transitions. Observe that $A_{Q(\vec{t})}$ has the same number of states as $A^{an}_{Q(\vec{t})}$. Complementing $A_{Q(\vec{t})}$ and intersecting it with $A_{\mathcal{V}}$, we obtain a 1NFA $A_{\neg Q(\vec{t}),\mathcal{V}}$ having the following property.

Theorem 4. *Let $\mathcal{V}$ be a set of C2RPQ$_c$ views, Q a C2RPQ$_c$ of arity n, $\vec{t}$ an n-tuple of proper constants, and $A_{\neg Q(\vec{t}),\mathcal{V}}$ the 1NFA constructed above. Then, $\vec{t} \in cert(Q, \mathcal{V})$ if and only if $A_{\neg Q(\vec{t}),\mathcal{V}}$ is empty.*

Theorem 5. *QA and QC for C2RPQ$_c$'s are EXPSPACE-complete.*

Proof. The lower bound for QC follows from EXPSPACE-hardness of QC for C2RPQ$_c$'s without constants and inverse, shown in [16]. By Theorem 2 we obtain the same lower bound for QA.

For the upper bound for QA, we appeal to the construction above.

The 1NFA $A_{\mathcal{V}}$ is the intersection of a 1NFA $A^{-}_{\mathcal{V}}$, which is polynomial in the size of $\mathcal{V}$ (i.e., the view definitions and view extensions), and of a 1NFA that complements the 2NFA $A_{\neg UNA}$, which is exponential in the number of proper constants in the view extension and view definitions.

As for $A_{Q(\vec{t})}$, we observe that the number of states of the 2NFA A_q, is polynomial in the size of Q and the number of constants in the view extensions, while the number of states of the 1NFAs A_s and $A_\$$ is exponential in the number of variables of Q and the number of constants in the view extensions. $A_{Q(\vec{t})}$ is obtained by projecting out the annotation from the intersection of A_s, $A_\$$, and of the 1NFA equivalent to A_q. Hence $A_{Q(\vec{t})}$ is a 1NFA exponential in the size of Q and the number of proper constants in the view extensions.

$A_{\neg Q(\vec{t}),\mathcal{V}}$ is obtained from the intersection of $A_{\mathcal{V}}$ and of the 1NFA that complements $A_{Q(\vec{t})}$. Hence, $A_{\neg Q(\vec{t}),\mathcal{V}}$ is a 1NFA of size double exponential in the size of Q and the number of proper constants. Considering that nonemptiness is NLOGSPACE and that we can build $A_{\neg Q(\vec{t}),\mathcal{V}}$ "on the fly" while checking for emptiness [16], we get an EXPSPACE upper bound for QA. By Theorem 2 we get also an EXPSPACE upper bound for QC.

4.4 Automaton for whole answer set

We show now how to modify the construction in the previous subsection to obtain *all* tuples that are not in $cert(Q(\vec{x}), \mathcal{V})$.

We represent each answer to $Q(\vec{x})$, with arity n, as a word in the language $\mathcal{L}_t = 1{\cdot}\mathcal{D}_V{\cdot}2{\cdot}\mathcal{D}_V \cdots n{\cdot}\mathcal{D}_V$, and we first construct an automaton accepting the set of words in $\mathcal{L}_t$ representing tuples that are *not* certain answers to $Q(\vec{x})$.

To do so, we construct a 1NFA $A_{Q(\vec{x})}$ that accepts all words of the form $w_{\vec{t}} \diamond w_{\mathcal{B}}$, where

- $w_{\mathcal{B}}$ represents a $\mathcal{V}$-canonical DB $\mathcal{B}$,
- $w_{\vec{t}}$ is a word in $\mathcal{L}_t$ representing a tuple $\vec{t}$ that is *not* in $ans(Q(\vec{x}), \mathcal{B})$, and
- "$\diamond$" acts as a separator.

The 1NFA $A_{Q(\vec{x})}$ is obtained from $A_{Q(\vec{t})}$ by changing the construction of A_q as explained below. Let $\vec{x} = (x_1, \ldots, x_n)$. The automaton uses the prefix $w_{\vec{t}}$, representing a tuple $\vec{t} = (c_1, \ldots, c_n)$, to obtain the mapping from x_i to c_i, for $i \in \{1, \ldots, n\}$, and once it has performed such a mapping, it proceeds essentially as A_q.

Let us first consider an atom in $Q(\vec{x})$ of the form $x_i \, E \, x_j$, where x_i and x_j are respectively the i-th and j-th distinguished variable, and assume that E is represented as a 1NFA $E = (\Sigma, S, \{s_0\}, \delta, \{s_f\})$ over the alphabet Σ. As shown in [16, 17], one can construct from E a 2NFA that checks whether a specified pair of constants (d, d') is in $ans(E, \mathcal{B})$. Such an automaton has the form $A' = (\Sigma', S', \{s'_0\}, \delta', \{s'_f\})$, where $\Sigma' = ((\Sigma \cup \mathcal{D}_N) \times 2^{\mathcal{Y}}) \cup \{\$\}$, $S' = S \cup \{s'_0, s'_f\} \cup \{s^{\leftarrow} \mid s \in S\} \cup S \times \mathcal{D}_N$, and δ_A is defined as in [16]. The states $s^{\leftarrow}$ are used to evaluate E moving backward on the word, while states (s, d) are used to search for occurrences of d while in state s of E. We define the 2NFA $A'' = (\Sigma'', S'', \{s''_0\}, \delta'', \{s''_f\})$, where $\Sigma'' = \Sigma' \cup \{\diamond, 1, \ldots, n\}$, $S'' = S' \cup \{s''_0, s^i_0, s''_f\} \cup \{s^d_0, s^d_f, s^{d\diamond}_f, s^{dd}_f \mid d \in \mathcal{D}_N\}$ and δ''

is obtained by adding to δ' the following transitions:

$$\begin{array}{rcll}
(s_0'', 1) & \in & \delta''(s_0'', \ell) & \text{for each } \ell \neq i \\
(s_0^i, 1) & \in & \delta''(s_0'', i) & \\
(s_0^d, 1) & \in & \delta''(s_0^i, d) & \text{for each } d \in \mathcal{D}_N \\
(s_0^d, 1) & \in & \delta''(s_0^i, \ell) & \text{for each } \ell \neq \diamond \\
((s_0, d), 1) & \in & \delta''(s_0^d, \diamond) & \\
(s_f^d, 1) & \in & \delta''(s_f, d) & \text{for each } d \in \mathcal{D}_N \\
(s_f^d, -1) & \in & \delta''(s_f^d, \ell) & \text{for each } d \in \mathcal{D}_N \text{ and } \ell \neq \diamond \\
(s_f^{d\diamond}, -1) & \in & \delta''(s_f^d, \diamond) & \text{for each } d \in \mathcal{D}_N \\
(s_f^{d\diamond}, -1) & \in & \delta''(s_f^{d\diamond}, \ell) & \text{for each } d \in \mathcal{D}_N \text{ and } \ell \neq d \\
(s_f^{dd}, -1) & \in & \delta''(s_f^{d\diamond}, d) & \text{for each } d \in \mathcal{D}_N \\
(s_f'', -1) & \in & \delta''(s_f^{dd}, j) & \text{for each } d \in \mathcal{D}_N
\end{array}$$

Intuitively, A'' finds the constant d associated to x_i in the prefix, then runs E starting from an occurrence of d in the postfix, and finally, when E finishes, checks that it has finished on the constant d' associated to x_j in the prefix.

We can proceed in a similar way for the atoms in $Q(\vec{x})$ where only one of the involved variables is a distinguished one. In this case we have to add only the transitions at the beginning, when the distinguished variable is the first one, and those at the end, when the distinguished variable is the second one. If both variables are non-distinguished, we use exactly the same 2NFA as in A_q.

Finally, we complement $A_{Q(\vec{x})}$ and intersect it with the 1NFA accepting the concatenation of $\mathcal{L}_t \cdot \diamond$ and of the language accepted by $A_{\mathcal{V}}$. Then we project away from the resulting automaton the part following "$\diamond$" (including the symbol "$\diamond$"), and obtain the 1NFA $A_{\neg Q(\vec{x}),\mathcal{V}}$ that accepts the set of tuples not in $cert(Q(\vec{x}), \mathcal{V})$. Observe that the latter projection can be obtained by simply removing all transitions labeled by "$\diamond$".

Theorem 6. *Let $\mathcal{V}$ be a set of C2RPQ$_c$ views, $\mathcal{D}_V$ the set of constants appearing in the view extensions and view definitions, Q a C2RPQ$_c$ of arity n, and $A_{\neg Q(\vec{x}),\mathcal{V}}$ the 1NFA constructed above. Then, $\mathcal{L}(A_{\neg Q(\vec{x}),\mathcal{V}}) = \{1c_1 \cdots nc_n \mid (c_1, \ldots, c_n) \notin cert(Q, \mathcal{V})\}$.*

Observe that all changes in the construction of $A_{\neg Q(\vec{x}),\mathcal{V}}$ wrt the construction of $A_{\neg Q(\vec{t}),\mathcal{V}}$ are polynomial. We get that, given a tuple $\vec{t}$, $A_{\neg Q(\vec{x}),\mathcal{V}}$ can be used to decide QA for $\vec{t}$ in a computationally optimal way.

Theorem 7. *Checking whether a word $1c_1 \cdots nc_n$, representing a tuple $(c_1, \ldots, c_n)$, is not accepted by $A_{\neg Q(\vec{x}),\mathcal{V}}$ is EXPSPACE-complete.*

Observe that one could compute the whole answer set for QA by constructing for each single tuple $\vec{t}$ of constants in $\mathcal{D}_V$ the automaton $A_{\neg Q(\vec{t}),\mathcal{V}}$, and then checking it for emptiness. The advantage of constructing $A_{\neg Q(\vec{x}),\mathcal{V}}$ instead, lies in the fact that such a construction factors out the common parts of the various $A_{\neg Q(\vec{t}),\mathcal{V}}$ leaving only a small part of the automaton specialized for each tuple.

5 QA and QC for T2RPQ$_c$'s

We study now QA and QC in the case where the query to answer using the views (resp., the right-hand side query for QC) is a T2RPQ$_c$. To do so we exploit two-way alternating word automata (2AFAs).

5.1 Two-way alternating automata

Given a set X, we define the set $B^+(X)$ as the set of all positive Boolean formulas over X, including 'true' and 'false' (i.e., for all $x \in X$, x is a formula, and if φ_1 and φ_2 are formulas, so are $\varphi_1 \wedge \varphi_2$ and $\varphi_1 \vee \varphi_2$). We say that a subset $X' \subseteq X$ *satisfies* a formula $\varphi \in B^+(X)$ (denoted $X' \models \varphi$) if, by assigning 'true' to all members of X' and 'false' to all members of $X \setminus X'$, the formula φ evaluates to 'true'. Clearly 'true' is satisfied by every subset of X (including the empty set) and 'false' cannot be satisfied.

In the following we denote the concatenation of I and J, where $I, J \in \mathbb{N}^*$, with IJ. A *tree* is a finite set $T \subseteq \mathbb{N}^*$ such that, if $Ii \in T$, where $I \in \mathbb{N}^*$ and $i \in \mathbb{N}$, then also $I \in T$. T is *complete* if, whenever $Ii \in T$, then also $Ij \in T$ for every $j \in [1..i]$. The elements of T are called *nodes*. When referring to trees, we make also use of the standard notions of *root*, *successor*, *descendant*, *predecessor*, and *leaf*. Given an alphabet Σ, a Σ*-labeled tree* is a pair (T, V), where T is a tree and $V : T \to \Sigma$ maps each node of T to an element of Σ. A k*-ary tree*, where k is a positive integer, is a tree $T \subseteq [1..k]^*$.

Two-way alternating automata generalize 1NFAs with the ability to move on the input string in both directions, and with the possibility to perform universal or existential moves (actually a combination of both). Formally, a *two-way alternating automaton* (2AFA) [21, 38, 9] $A = (\Gamma, S, S_0, \delta, S_f)$ consists of an alphabet Γ, a finite set of states S, a set $S_0 \subseteq S$ of initial states, a transition function $\delta : S \times \Sigma \to B^+(S \times \{-1, 0, 1\})$, and a set $S_f \subseteq S$ of accepting states. Intuitively, a transition $\delta(s, a)$ spawns several copies of A, each one starting in a certain state and with the head on the symbol to the left of a (-1), to the right of a (1), or on a itself (0), and specifies by means of a positive Boolean formula how to combine acceptance or non-acceptance of the spawned copies. A *run* of A on a finite word $w = a_0 \cdots a_\ell$ is a labeled tree (T, r), where $r : T \to S \times [0..\ell + 1]$. A node in T labeled by (s, i) describes a copy of A that is in state s and reads the letter a_i of w. The labels of adjacent nodes of T have to satisfy the transition function δ. Formally, $r(\varepsilon) = (s_0, 0)$ where $s_0 \in S_0$, and for all nodes I with $r(I) = (s, i)$ and $\delta(s, a_i) = \varphi$, there is a (possibly empty) set $Z = \{(s_1, c_1), \ldots, (s_h, c_h)\} \subseteq S \times \{-1, 0, 1\}$ such that $Z \models \varphi$ and for all $j \in [1..h]$, there is a successor of I in T labeled $(s_j, i + c_j)$. The run is *accepting* if all leaves of the run are labeled by $(s, \ell + 1)$ for some $s \in S_f$. A *accepts* w if it has an accepting run on w. The set of words accepted by A is denoted $L(A)$.

It is known that 2AFAs define regular languages [38], and that, given a 2AFA with n states accepting a regular language L, one can construct a 1NFA with $2^{O(n^2)}$ states, accepting L [8]. In addition, by exploiting the same idea used in the reduction in [53] from 2NFAs to 1NFAs, we can show the following result.

Theorem 8. *Given a 2AFA with n states accepting a language L, one can construct a 1NFA with $2^{O(n)}$ states, accepting the complement of L.*

5.2 T2RPQ$_c$'s

Obviously, QA for the case where the query to be answered is a T2RPQ$_c$ is a special case of QA for C2RPQ$_c$'s. However, the special tree-structure of the query Q to be answered allows us to avoid the construction of the doubly exponential automaton for Q. Instead, to build $A_{Q(\vec{t})}$ we make use of 2AFAs, which can directly simulate the evaluation of a T2RPQ$_c$ on words representing canonical DBs, without the need to introduce annotations for the non-distinguished variables of Q. In particular, we exploit the ability of 2AFAs to: (i) move on the word both forward and backward, which corresponds to traversing edges of the DB in both directions;

(ii) "jump" between positions in the word representing the same node; (iii) check a "conjunction" of conditions.

To denote a T2RPQ$_c$ we use variables indexed with nodes of a tree, each of which is represented as usual by a (possibly empty) sequence of integers, denoting the path leading from the root to the node. Let the set $\mathcal{I}$ of indices of the variables in the query be a finite k-ary tree (for some k) and let Q be the T2RPQ$_c$ of arity n

$$Q(x_1, \ldots, x_n) \leftarrow \bigwedge_{y_I, y_{Ij} \in \mathcal{I}} y_I \, E_{Ij} \, y_{Ij}$$

where I is the sequence of integers denoting a node in the tree, Ij is the sequence denoting the j-th successor of node I, and E_{Ij} denotes the regular expression in the (unique) atom involving variables or proper constants y_I and y_{Ij}. Notice that the body of the query has the structure of a tree, i.e., each variable appears at most once as right variable in an atom, and, except for y_ε, if a variable appears as left variable in an atom, then it must also appear as right variable in some other atom.

5.3 Automaton for a single tuple

We first address the problem of checking whether a tuple $\vec{t}$ is in the certain answer to a T2RPQ$_c$ Q wrt a set of C2RPQ$_c$ views $\mathcal{V}$.

We can build the automaton $A_\mathcal{V}$ accepting $\mathcal{V}$-words representing $\mathcal{V}$-canonical DBs exactly as in Section 4.2. Let w be a $\mathcal{V}$-word over the alphabet $\Sigma \cup \mathcal{D}_N \cup \{\$\}$, representing a $\mathcal{V}$-canonical DB $\mathcal{B}_w$. We construct a 2AFA $A_{Q(\vec{t})}$, that accepts w if and only if $\vec{t} \in ans(Q, \mathcal{B}_w)$.

To construct $A_{Q(\vec{t})}$, we assume that each E_I is represented as a 1NFA $E_I = (\Sigma, S_I, s_I^0, \delta_I, s_I^f)$ over the alphabet Σ, and that the automata for different E_I's have disjoint sets of states. Then $A_{Q(\vec{t})} = (\Sigma^Q, S^Q, \{s_\varepsilon^f\}, \delta^Q, F^Q)$, where $\Sigma^Q = \Sigma \cup \mathcal{D}_N \cup \{\$\}$, and S^Q, F^Q, and δ^Q are defined below.

For simplicity, we use the following notation for transitions of 2AFAs: we write $(s', \ell') \in \delta^Q(s, \ell)$ meaning that $\delta^Q(s, \ell)$ is a disjunction of transitions, and that (s', ℓ') is one of the disjuncts.

We first construct for each atom $y_I \, E_J \, y_J$ (with $J = Ij$ for some j) in the query a part of $A_{Q(\vec{t})}$ that checks that such atom is satisfied in the $\mathcal{V}$-canonical DB represented by the current word. To do so, we introduce in S^Q a set of states $S_J^Q = S_J \cup \{s^\leftarrow \mid s \in S_J\} \cup \{s^\hookleftarrow \mid s \in S_J\} \cup S_J \times \mathcal{D}_N$ and add the following transitions to δ^Q (this construction is a variation of the one in [16]):

1. $(s^\leftarrow, -1) \in \delta^Q(s, \ell)$, for each $s \in S_J$ and $\ell \in \Sigma \cup \mathcal{D}_N$. At any point such transitions make the automaton ready to scan one step backward by placing it in "backward-mode".
2. $(s_2, 1) \in \delta^Q(s_1, r)$ and $(s_2, 0) \in \delta^Q(s_1^\leftarrow, r^-)$, for each transition $s_2 \in \delta_J(s_1, r)$ of E_J. These transitions correspond to the transitions of E_J that are performed forward or backward according to the current "scanning mode".
3. For each $s \in S_J$ and each $d \in \mathcal{D}_N$

$$\begin{array}{rll}
((s,d),0) \in \delta^Q(s,d) & & \\
((s,d),0) \in \delta^Q(s^\leftarrow,d) & & \\
((s,d),1) \in \delta^Q((s,d),\ell), & \text{for each } \ell \in \Sigma^Q & \\
((s,d),-1) \in \delta^Q((s,d),\ell), & \text{for each } \ell \in \Sigma^Q & \\
(s,0) \in \delta^Q((s,d),d) & & \\
(s,1) \in \delta^Q(s,d) & &
\end{array}$$

On a symbol representing a node d while in a state s, the automaton may enter into "search-for-d-mode", reflected in a state (s, d) (first and second clause) and move to any other occurrence of d in the word. When it finds such an occurrence, the automaton exits search-mode (second last clause) and continues its computation either forward (last clause) or backward (see item 2).

4. For each $s \in S_J$ and each $d_1, d_2 \in \mathcal{D}_N$

$$(s^{\hookrightarrow}, 1) \ \in \ \delta^Q((s, d_1), d_1) \qquad\qquad ((s, d_2), 0) \ \in \ \delta^Q(s^{\hookrightarrow}, d_2)$$

Whenever the automaton reaches a symbol representing a node d_1 while it is in search-for-d_1-mode, and d_2 appears immediately to the right of d_1, the automaton may switch to search-for-d_2-mode. This takes into account that two adjacent symbols $d_1 d_2$ actually represent the same node of the DB. Notice that switching from search-for-d_2-mode to search-for-d_1-mode is already taken into account by exiting search-for-d_1-mode (transition 5 in 3), switching to backward-mode (point 1), and switching to search-for-d_2-mode while exiting from backward-mode (second transition in item 3).

Observe that the separator symbol \$ does not allow transitions except in search-mode. Its role is to force the automaton to move in the correct direction when exiting search-mode.

We then "connect" the different parts of $A_{Q(\vec{t})}$ corresponding to the different atoms in the query by taking into account the tree structure of the query.

1. $(s^f_\varepsilon, 1) \in \delta^Q(s^f_\varepsilon, \ell)$ for each $\ell \in \Sigma^Q$. These transitions place the head of the automaton in some randomly chosen position of the input string.
2. Consider now a variable or proper constant y_I, and all atoms in which y_I appears on the left of the regular expression. Let such atoms be $y_I \, E_{Ij} \, y_{Ij}$, for $j \in [1..m_I]$:
   - if y_I is a distinguished variable x_i, then we add the transition $(s^0_{I1}, 0) \wedge \cdots \wedge (s^0_{Im_I}, 0) \in \delta^Q(s^f_I, \vec{t}[i])$, where $\vec{t}[i]$ is the i-th component of $\vec{t}$, and s^0_{Ij} is the starting state of the part of the automaton for $y_I \, E_{Ij} \, y_{Ij}$;
   - If y_I is a proper constant c, then we add the transition $(s^0_{I1}, 0) \wedge \cdots \wedge (s^0_{Im_I}, 0) \in \delta^Q(s^f_I, c)$;
   - if y_I is a non-distinguished variable of Q, then we add the transitions $(s^0_{I1}, 0) \wedge \cdots \wedge (s^0_{Im_I}, 0) \in \delta^Q(s^f_I, \ell)$, for each $\ell \in \Sigma \cup \mathcal{D}_N$.

   These transitions connect in the appropriate way s^f_I (which is either the final state of A_I or the initial state s^f_ε of A^{Q_2}) with the starting states s^0_{Ij} of the parts of the automaton corresponding to the atoms $y_I \, E_{Ij} \, y_{Ij}$, without moving on the input string. They also check that distinguished variables or constants are mapped to the appropriate nodes in the word. Note that the one above are the only "and"-transitions used in the automaton.
3. $(s, 1) \in \delta^Q(s, \ell)$, for each $s \in F^Q$ and each $\ell \in \Sigma^Q$. These transitions move the head of the automaton to the end of the input string, when the automaton enters a final state.

We still have to specify the set F^Q of final states of $A_{Q(\vec{t})}$. For each variable or constant y_J which is a leaf of the tree representing Q (and which appears only in an atom $y_I \, E_J \, y_J$):

- If y_J is a distinguished variable x_i, then we introduce in S^Q a new state s^F_J, add the transition $s^F_J \ \in \ \delta^Q(s^f_J, \vec{t}[i])$, and add s^F_J to the set F^Q of final states of $A_{Q(\vec{t})}$. This corresponds to checking that the distinguished variable x_i is actually mapped to $\vec{t}[i]$.
- If y_J is a proper constant c, then we introduce in S^Q a new state s^F_J, add the transition $s^F_J \ \in \ \delta^Q(s^f_J, c)$, and add s^F_J to the set F^Q of final states of $A_{Q(\vec{t})}$. This corresponds to checking that the constant c is actually mapped to itself.

– If y_J is a non-distinguished variable of Q, then we simply add s_J^f to the set F^Q of final states of $A_{Q(\vec{t})}$.

Observe that the mapping of non-distinguished variables of Q to nodes of a $\mathcal{V}$-canonical DB represented by a $\mathcal{V}$-word w accepted by $A_{Q(\vec{t})}$ is not explicited as an annotation of w. However, the existence of an accepting run of $A_{Q(\vec{t})}$ over w guarantees the existence of such a mapping, since each branching in the query is handled by an "and"-transition in the automaton. In this way the different occurrences of a non-distinguished variable in the body of Q are all mapped to the same node, which is represented by the symbol in w at which the "and"-transition in the run occurs.

Finally, we define $A_{\neg Q(\vec{t}),\mathcal{V}}$ as the intersection of the 1NFA $A_{\mathcal{V}}$ and of the 1NFA corresponding to the complement of $A_{Q(\vec{t})}$ (see Theorem 8).

Theorem 9. *Let $\mathcal{V}$ be a set of C2RPQ$_c$ views, Q a T2RPQ$_c$ of arity n, $\vec{t}$ an n-tuple of proper constants, and $A_{\neg Q(\vec{t}),\mathcal{V}}$ the 1NFA constructed above. Then, $\vec{t} \in cert(Q,\mathcal{V})$ if and only if $A_{\neg Q(\vec{t}),\mathcal{V}}$ is empty.*

Considering the construction above we get the following complexity characterization.

Theorem 10. *QA for C2RPQ$_c$ views and a T2RPQ$_c$ query, and QC between a C2RPQ$_c$ and a T2RPQ$_c$ are PSPACE-complete.*

Proof. QC is clearly PSPACE-hard, since already containment of regular expressions (and hence RPQs) is PSPACE-hard [36]. By Theorem 2 we obtain the same lower bound for QA.

For the upper bound for QA, by Theorem 9, it suffices to check the emptiness of $A_{\neg Q(\vec{t}),\mathcal{V}}$, which is the intersection of the 1NFA $A_{\mathcal{V}}$ and of the 1NFA corresponding to the complement of $A_{Q(\vec{t})}$. The 1NFA $A_{\mathcal{V}}$ is the intersection of a 1NFA $A_{\mathcal{V}}^-$ which is polynomial in the size of $\mathcal{V}$ (i.e., the view definitions and view extensions) and of a 1NFA, called A_{UNA} in the following, that complements the 2NFA $A_{\neg UNA}$, which is exponential in the number of proper constants in the view extension and view definitions.

The 2AFA $A_{Q(\vec{t})}$ is polynomial in Q. By Theorem 8, the 1NFA corresponding to the complement of $A_{Q(\vec{t})}$, called $A_{\neg Q(\vec{t})}$ in the following, is exponential in Q. However, we can check "on-the-fly" whether $A_{\mathcal{V}}^- \cap A_{UNA} \cap A_{\neg Q(\vec{t})}$ is empty, and we do not need to construct A_{UNA} and $A_{\neg Q(\vec{t})}$ explicitly: Whenever the emptiness algorithm wants to move from a state s_1 of the intersection of $A_{\mathcal{V}}^-$, A_{UNA}, and $A_{\neg Q(\vec{t})}$ to a state s_2, it guesses s_2 and checks that it is directly connected to s_1. Such a check can be done in time polynomial in the sizes of $A_{\mathcal{V}}^-$, A_{UNA}, and $A_{\neg Q(\vec{t})}$ [53]. Once this has been verified, the algorithm can discard s_1. Thus, at each step the algorithm needs to keep in memory at most two states and there is no need to generate all of A_{UNA} and $A_{\neg Q(\vec{t})}$ at any single step of the algorithm.

Considering Theorem 2, we also get that QC for the case where the query on the left-hand side is a C2RPQ$_c$ and the one on the right-hand side is a T2RPQ$_c$ can be decided in PSPACE.

5.4 Automaton for whole answer set

The technique developed in Section 4.4 to compute the whole set of certain answers can be adapted also to the case where the query to be answered is a T2RPQ$_c$. Observe that we cannot deal with each RPQ separately, as done in Section 4.4. Instead, at those points where the automaton for a single tuple checks the presence of a proper constant assigned to a distinguished variable, the automaton for the whole answer set has to check that the proper constant encountered in the word is mapped in the prefix to the index of the distinguished variable that the automaton expects. This can be done by adding a suitable conjunct in the transition of the automaton that switches to a state from which the check is done.

As for the case of C2RPQ$_c$'s, this construction maintains the same computational complexity as the one for a single tuple. Hence, using the automaton for the whole answer set, one can decide in PSPACE whether a certain tuple is in the certain answer to a T2RPQ$_c$ wrt a set of C2RPQ$_c$ views.

6 Conclusions

We have studied query containment and view-based query answering for the class of C2RPQ$_c$'s. We have presented a mutual reduction between the two problems, and we have shown that both are EXPSPACE-complete in the general case, and PSPACE-complete in the special case of T2RPQ$_c$'s. Observe that PSPACE and EXPSPACE algorithms currently can be implemented in time that is respectively exponential and doubly exponential in the size of the input. Hence, the results for T2RPQ$_c$'s imply that query containment for not too large queries is indeed feasible, since it can be done in time that is exponential in a small number. This has to be contrasted with the general case where even containment for moderately sized queries appears to be infeasible, since it requires time that is doubly exponential, which is a large amount of time even for small queries.

For the sake of simplicity, we did not consider union in this paper. However, all the results presented here can be directly extended to unions of C2RPQ$_c$'s (respectively, unions of T2RPQ$_c$'s). Also, as already mentioned, they can be easily extended to the case of rooted DBs, i.e., to the case where the DB is an edge-labeled rooted graph. In particular, the notion of root can be simulated both in query containment and in query answering. For query containment, it is sufficient to add one variable to the left-hand side query and suitable atoms to enforce that it is connected to all other variables (and hence to all nodes of the counterexample database). For view-based query answering, the root can be modeled by means of a distinguished constant that is forced to be connected to all other constants appearing in the view extensions by means of an additional view with definition $V(x, y) \leftarrow x\,\Sigma^*\,y$.

In the future, we aim at extending our work in order to take into account the following aspects. (i) While we have assumed in this paper that views are sound (they return a subset of the answers to the associated query), in data integration views can also be used to model data sources that are either complete (they return a superset of the answers to the associated query) or exact (they are both sound and complete). We conjecture that our techniques for view-based query answering can be adapted in order to take into account both complete and exact views. (ii) It would be interesting to study the complexity of view-based query answering with respect to the size of the view extensions only (data complexity), as done in [15, 17] for RPQs and 2RPQs.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, Los Altos, 1999.
2. S. Abiteboul and O. Duschka. Complexity of answering queries using materialized views. In *Proc. of PODS'98*, pages 254–265, 1998.
3. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.

4. F. N. Afrati, M. Gergatsoulis, and T. Kavalieros. Answering queries using materialized views with disjunction. In *Proc. of ICDT'99*, volume 1540 of *LNCS*, pages 435–452. Springer-Verlag, 1999.
5. A. V. Aho, Y. Sagiv, and J. D. Ullman. Equivalence among relational expressions. *SIAM J. on Computing*, 8:218–246, 1979.
6. D. Barbará and T. Imieliński. Sleepers and workaholics: Caching strategies in mobile environments. In *Proc. of ACM SIGMOD*, pages 1–12, 1994.
7. C. Beeri, A. Y. Levy, and M.-C. Rousset. Rewriting queries using views in description logics. In *Proc. of PODS'97*, pages 99–108, 1997.
8. J.-C. Birget. State-complexity of finite-state devices, state compressibility and incompressibility. *Mathematical Systems Theory*, 26(3):237–269, 1993.
9. J. A. Brzozowski and E. Leiss. Finite automata and sequential networks. *Theor. Comp. Sci.*, 10:19–35, 1980.
10. P. Buneman. Semistructured data. In *Proc. of PODS'97*, pages 117–121, 1997.
11. P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization technique for unstructured data. In *Proc. of ACM SIGMOD*, pages 505–516, 1996.
12. D. Calvanese, G. De Giacomo, and M. Lenzerini. On the decidability of query containment under constraints. In *Proc. of PODS'98*, pages 149–158, 1998.
13. D. Calvanese, G. De Giacomo, and M. Lenzerini. Answering queries using views over description logics knowledge bases. In *Proc. of AAAI 2000*, pages 386–391, 2000.
14. D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Rewriting of regular expressions and regular path queries. In *Proc. of PODS'99*, pages 194–204, 1999.
15. D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Answering regular path queries using views. In *Proc. of ICDE 2000*, pages 389–398, 2000.
16. D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *Proc. of KR 2000*, pages 176–185, 2000.
17. D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Query processing using views for regular path queries with inverse. In *Proc. of PODS 2000*, pages 58–66, 2000.
18. D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. View-based query processing and constraint satisfaction. In *Proc. of LICS 2000*, pages 361–371, 2000.
19. D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery: A query language for XML. W3C Working Draft, Feb. 2001. Available at http://www.w3.org/TR/xquery.
20. E. P. F. Chan. Containment and minimization of positive conjunctive queries in oodb's. In *Proc. of PODS'92*, pages 202–211, 1992.
21. A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *J. of the ACM*, 28(1):114–133, 1981.
22. A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. of STOC'77*, pages 77–90, 1977.
23. S. Chaudhuri, S. Krishnamurthy, S. Potarnianos, and K. Shim. Optimizing queries with materialized views. In *Proc. of ICDE'95*, Taipei (Taiwan), 1995.
24. S. Chaudhuri and M. Y. Vardi. On the equivalence of recursive and nonrecursive Datalog programs. In *Proc. of PODS'92*, pages 55–66, 1992.
25. C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In *Proc. of ICDT'97*, pages 56–70, 1997.
26. J. Clark and S. DeRose. XML Path Language (XPath) version 1.0 – W3C recommendation 16 november 1999. Technical report, World Wide Web Consortium, 1999. Available at http://www.w3.org/TR/1999/REC-xpath-19991116.
27. S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *Proc. of PODS'99*, pages 155–166, 1999.
28. A. C. K. David S. Johnson. Testing containment of conjunctive queries under functional and inclusion dependencies. *J. of Computer and System Sciences*, 28(1):167–189, 1984.
29. A. Deutsch, M. F. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. Submission to the World Wide Web Consortium, Aug. 1998. Available at http://www.w3.org/TR/NOTE-xml-ql.

30. A. Deutsch and V. Tannen. Optimization properties for classes of conjunctive regular path queries. In *Proc. of DBPL 2001*, 2001.
31. O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. In *Proc. of PODS'97*, pages 109–116, 1997.
32. M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Catching the boat with strudel: Experiences with a web-site management system. In *Proc. of ACM SIGMOD*, pages 414–425, 1998.
33. D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *Proc. of PODS'98*, pages 139–148, 1998.
34. G. Grahne and A. O. Mendelzon. Tableau techniques for querying information sources through global schemas. In *Proc. of ICDT'99*, volume 1540 of *LNCS*, pages 332–347. Springer-Verlag, 1999.
35. S. Grumbach, M. Rafanelli, and L. Tininini. Querying aggregate data. In *Proc. of PODS'99*, pages 174–184, 1999.
36. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley Publ. Co., Reading, Massachussetts, 1979.
37. A. C. Klug. On conjunctive queries containing inequalities. *J. of the ACM*, 35(1):146–160, 1988.
38. R. E. Ladner, R. J. Lipton, and L. J. Stockmeyer. Alternating pushdown and stack automata. *SIAM J. on Computing*, 13(1):135–155, 1984.
39. A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. of PODS'95*, pages 95–104, 1995.
40. A. Y. Levy and M.-C. Rousset. Combining horn rules and description logics in CARIN. *Artificial Intelligence*, 104(1–2):165–209, 1998.
41. A. Y. Levy and D. Suciu. Deciding containment for queries with complex objects. In *Proc. of PODS'97*, pages 20–31, 1997.
42. J. McHugh and J. Widom. Optimizing branching path expressions. Technical report, Stanford University, 1999. Available at http://www-db.stanford.edu/pub/papers/mp.ps.
43. T. Milo and D. Suciu. Index structures for path expressions. In *Proc. of ICDT'99*, volume 1540 of *LNCS*, pages 277–295. Springer-Verlag, 1999.
44. Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proc. of PODS 2000*, pages 35–46, 2000.
45. A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *Proc. of PODS'95*, 1995.
46. Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. of the ACM*, 27(4):633–655, 1980.
47. H. Seidl. Deciding equivalence of finite tree automata. *SIAM J. on Computing*, 19(3):424–437, 1990.
48. O. Shmueli. Equivalence of Datalog queries is undecidable. *J. of Logic Programming*, 15(3):231–241, 1993.
49. D. Srivastava, S. Dar, H. V. Jagadish, and A. Levy. Answering queries with aggregation using views. In *Proc. of VLDB'96*, pages 318–329, 1996.
50. O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis. The GMAP: A versatile tool for phyisical data independence. *VLDB Journal*, 5(2):101–118, 1996.
51. J. D. Ullman. Information integration using logical views. In *Proc. of ICDT'97*, volume 1186 of *LNCS*, pages 19–40. Springer-Verlag, 1997.
52. R. van der Meyden. *The Complexity of Querying Indefinite Information*. PhD thesis, Rutgers University, 1992.
53. M. Y. Vardi. A note on the reduction of two-way automata to one-way automata. *Information Processing Letters*, 30(5):261–264, 1989.
54. V. Vianu. A web odyssey: From Codd to XML. In *Proc. of PODS 2001*, 2001. Invited talk.
55. J. Widom, editor. *IEEE Bull. on Data Engineering – Special Issue on Materialized Views and Data Warehousing*, volume 18. IEEE Computer Society Press, 1995.
56. J. Widom, editor. *IEEE Bull. on Data Engineering – Special Issue on Materialized Views and Data Warehousing*, volume 22. IEEE Computer Society Press, 1999.
57. M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. of VLDB'81*, pages 82–94, 1981.