Querying XML Data Sources That Export Very Large Sets of Views

BOGDAN CAUTIS, Télécom ParisTech, CNRS LTCI
ALIN DEUTSCH, University of California, San Diego
NICOLA ONOSE, University of California, Irvine
VASILIS VASSALOS, Athens University of Economics and Business

We study the problem of querying XML data sources that accept only a limited set of queries, such as sources accessible by Web services which can implement very large (potentially infinite) families of XPath queries. To compactly specify such families of queries we adopt the Query Set Specifications, a formalism close to context-free grammars.

We say that query Q is *expressible* by the specification $\mathcal{P}$ if it is equivalent to some expansion of $\mathcal{P}$. Q is *supported* by $\mathcal{P}$ if it has an equivalent rewriting using some finite set of $\mathcal{P}$'s expansions. We study the complexity of expressibility and support and identify large classes of XPath queries for which there are efficient (PTIME) algorithms. Our study considers both the case in which the XML nodes in the results of the queries lose their original identity and the one in which the source exposes persistent node ids.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing, Distributed Databases*; H.2.3 [**Database Management**]: Languages—*Query languages*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Query rewriting, XML, semi-structured data, limited capabilities, views

ACM Reference Format:
Cautis, B., Deutsch, A., Onose, N., and Vassalos, V. 2011. Querying XML data sources that export very large sets of views. ACM Trans. Datab. Syst. 36, 1, Article 5 (March 2011), 42 pages.
DOI = 10.1145/1929934.1929939 http://doi.acm.org/10.1145/1929934.1929939

1. INTRODUCTION

Current Web data sources usually do not allow clients to ask arbitrary queries, but instead publish as Web Services a set of queries they are willing to answer, which we will refer to as *views.* Main reasons for that are performance requirements, business model considerations, and access restrictions deriving from security policies. Querying such sources involves finding one or several legal views that can be used to answer the client query.

Of particular interest is the case when the set of views is very large (possibly exponential in the size of the schema or even infinite), precluding explicit enumeration by the source owner as well as full comprehension by the client query developer. In such scenarios, recent proposals advocate the owner's specifying the set of legal views

A. Deutsch was supported by the NSF under grants IIS 0910820 and IIS 0705589.
Authors' addresses: B. Cautis, Telecom ParisTech, Computer Science Department, 46 rue Barrault, 75013 Paris, France; email: cautis@telecom-paristech.fr; A. Deutsch, Department of Computer Science and Engineering, University of California, San Diego, CA 92093-0404; N. Onose, Department of Computer Science, 3019 Donald Bren Hall, Irvine, CA 92697-3435; email: onose@ics.uci.edu; V. Vassalos, Informatics Department, Athens University of Economics and Business (AUEB), 76 Patission St., Athens GR 104 34, Greece; email: vassalos@aueb.gr.

DOI 10.1145/1929934.1929939 http://doi.acm.org/10.1145/1929934.1929939

implicitly, using a compact representation (in the same spirit in which a potentially infinite language is finitely specified by a grammar). Clients are unaware of the legal views, and simply pose their query against a logical schema exported by the source (the same schema against which the views are defined). While this approach provides a simpler interface to source owner and client, it raises a technical challenge, as now the system has to automatically identify and extract from the compact encoding a finite set of legal views that can be used to answer the client query.

This problem has been the object of several recent studies in a relational setting [Levy et al. 1999; Vassalos and Papakonstantinou 2000; Cautis et al. 2009], but has not been addressed for sources that publish XML data (as is the case for most current Web Services). Since our focus is on practical algorithms, we consider sources that make XML data available through sets of views belonging to an XPath fragment for which the basic building blocks of rewriting algorithms, namely containment and equivalence, are tractable [Miklau and Suciu 2004]. As a formalism for compactly representing large sets of such views, we adopt a variation of the Query Set Specification Language (QSSL) [Petropoulos et al. 2003], a grammarlike formalism for specifying XPath view families [Newman and Özsoyoglu 2004].

Expressibility and support. As in the literature on sources exporting sets of legal relational queries [Vassalos and Papakonstantinou 2000; Cautis et al. 2009], we consider two settings for query answering. The first one is when the client query has to be fully answered by asking one legal query over the source, with no postprocessing of its result. The corresponding decision problem is called *expressibility* [Cautis et al. 2009]: a query q is said to be *expressed* by the source if it is equivalent to a view published by it. The second setting is when the capabilities of the source are extended by a *source wrapper* [Papakonstantinou et al. 1995] that intercepts the client query, finds an equivalent rewriting for it in terms of the views, post-processes the results locally and returns the query result to the client. The associated problem is called support [Cautis et al. 2009]: given a rewriting query language $\mathcal{L}_R$, q is supported by $\mathcal{P}$ if it has an equivalent rewriting in $\mathcal{L}_R$ using some finite set of legal queries supported by the source.

Expressibility and support generalize the problems of equivalence and existence of a rewriting using views from the classical case in which the set of views is explicitly listed to the case in which this set is very large, potentially infinite, being specified implicitly by a compact representation.

XPath rewriting. Earlier research [Xu and Özsoyoglu 2005; Mandhani and Suciu 2005] on XPath rewriting studied the problem of equivalently rewriting an XPath query by navigating inside a *single* materialized XPath view. This is the only kind of rewritings supported when the query cache can only store or can only obtain *copies* of the XML elements in the query answer, and so the original node identities are lost.

We have recently witnessed an industrial trend towards enhancing XPath queries with the ability to expose node identifiers and exploit them using intersection of node sets (via identity-based equality). This trend is supported by systems such as Balmin et al. [2004]. This development enables for the first time multiple-view rewritings obtained by intersecting several materialized view results. The single-view rewritings considered in early XPath research have only limited benefit, as many queries with no single-view rewriting can be rewritten using multiple views. In this article, we consider both the case in which the XML nodes in the results of the queries lose their original identity (hence a rewriting can only use one view) and the one in which the source exposes persistent node ids (and rewritings using multiple views are possible).

Example 1.1. Throughout this article we consider the example of a tourism agency that enables users to find organized trips matching their criteria. The set of allowed queries is specified by a compact QSS encoding (to be described shortly). On the schema

of views published by the source, the client formulates a query q_1, asking for museums during a tour in whose schedule there is also a slot for taking a walk and which is part of a guided secondary trip:

$$q_1 : \ doc(T)//vacation//trip/trip[guide]//tour[schedule//walk]/museum.$$

The system analyzes the query and the specification and finds two views that may be relevant for answering q_1. These are v_1, which returns museums in secondary trips for which there is a guide:

$$v_1 : \ doc(T)//vacation//trip/trip[guide]//museum.$$

and v_2, which returns museums on a tour in which there has been also scheduled a walk:

$$v_2 : \ doc(T)//vacation//trip//tour[schedule//walk]/museum.$$

q_1 cannot be answered just by navigating into the result of v_1 or into the result of v_2. The reason is that q_1 needs both to enforce that the *trip* has a *guide* and that the *tour* has a *walk* in the *schedule*. v_1 or v_2 taken individually can enforce one of the two conditions, but not both, and navigation down into the view does not help either, since the output node *museum* is below the *trip* and *tour* nodes. Since no other views published by the source can contribute to rewriting q_1, in the absence of ids, the system will reject q_1, as it is neither expressed, nor supported by the source.

However, if the views expose persistent node ids, we will show that q_1 can be rewritten as an intersection of v_1 and v_2.

Contributions. We study the complexity of expressibility and support and identify large classes of XPath queries for which there are efficient (PTIME) algorithms. For expressibility, we give a PTIME decision procedure that works for any QSS and for any XPath query from a large fragment allowing child and descendant navigation and predicates. We show that support in the absence of ids remains in PTIME, for the same XPath fragment for which we studied expressibility. However, for this fragment, support in the presence of ids becomes coNP-hard. This is a consequence of previous results [Cautis et al. 2008], showing that rewriting XPath using an intersection of XPath views (a problem subsumed by support) is already coNP-hard. This is a major difference with respect to the relational case, in which support and expressibility were proven inter-reducible [Cautis et al. 2009]. Since our focus is on practical algorithms, we propose a PTIME algorithm for id-based support that is sound for any XPath query, and becomes complete under fairly permissive restrictions on the query, without further restricting the language of the views. Our results are in stark contrast with previous results in the relational setting [Levy et al. 1999; Vassalos and Papakonstantinou 2000], where already the simple language of conjunctive queries leads to EXPTIME completeness of expressivity and support [Cautis et al. 2009], but on the other hand is closed under intersection, which poses no additional problem.

We also consider several extensions for support and expressibility, asking whether our tractable algorithms can be adapted to richer specification formalisms and rewrite languages. We show that this is not the case and that both problems become NP-hard under extensions, even when assuming strong restrictions in the language of queries and views.

Outline of the article. The article is structured as follows. Section 2 presents the language of client queries (tree patterns) and the language of query rewriting plans (tree patterns and intersections thereof). Section 3 describes the query set specifications (QSS). The problem of expressibility is analyzed in Section 4. The problem of support is studied starting from Section 5, first in the absence of persistent ids and then in their

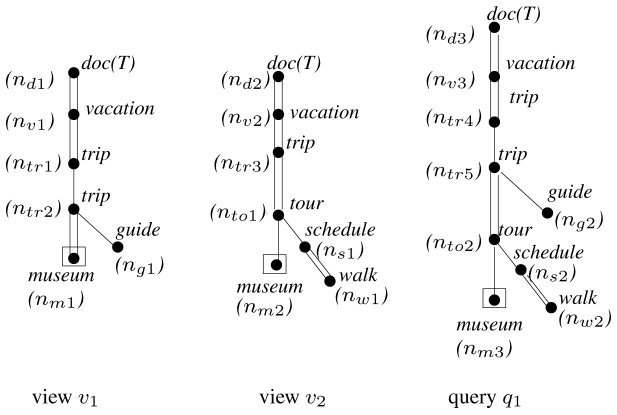

Fig. 1. The tree patterns of queries v_1, v_2 and q_1.

presence (Sections 6, 7, 8). QSS and rewriting language extensions are presented in Sections 9, 10, and 11. Alternatives for the language of queries and views are discussed in Section 12. Section 13 discusses related work and Section 14 concludes.

2. XPATH AND TREE PATTERNS

We consider an XML document as an unranked, unordered rooted tree t modeled by a set of edges EDGES(t), a set of nodes NODES(t), a distinguished root node ROOT(t), and a labeling function λ_t, assigning to each node a label from an infinite alphabet Σ.

We consider XPath queries with child / and descendant // navigation, side branches, without wildcards. We call the resulting language *XP*, and define its grammar as:

$$\begin{aligned} apath &::= doc(\text{“}name\text{”})/rpath \mid doc(\text{“}name\text{”})//rpath \\ rpath &::= step \mid rpath/rpath \mid rpath//rpath \\ step &::= label\ pred \\ pred &::= \epsilon \mid [rpath] \mid [.//rpath] \mid pred\ pred. \end{aligned}$$

The subexpressions inside brackets are called *predicates*.

All definitions and results of this paper extend naturally when allowing equality with constants in predicates. For presentation simplicity, this feature will be ignored in the core of the article, and it is briefly discussed in Section 9.

In the following, we will prefer an alternative representation for XML queries widely used in literature, the one of *tree patterns* [Miklau and Suciu 2004].

Definition 2.1. A *tree pattern* p is a nonempty rooted tree, with a set of nodes NODES(p) labeled with symbols from Σ, a distinguished node called the *output node* OUT(p), and two types of edges: *child edges*, labeled by / and *descendant edges*, labeled by //. The root of p is denoted ROOT(p).

Any *XP* expression can be translated into a tree pattern query and vice versa (see, for instance, Miklau and Suciu [2004]). For a given tree pattern query p, $xpath(p)$ is the associated *XP* expression.

Example 2.2. Figure 1 shows the tree patterns corresponding to v_1, v_2 and q_1 from Example 1.1. Each node has a label and a unique node symbol, written inside parenthesis. Output nodes are distinguished in the graphical representation by a square.

The semantics of a tree pattern can be given using embeddings.

Definition 2.3. An *embedding* of a tree pattern p into a tree t over Σ is a function e from NODES(p) to NODES(t) that has the following properties: (1) e(ROOT(p)) = ROOT(t); (2) for any $n \in$ NODES(p), LABEL($e(n)$) = LABEL(n); (3) for any /-edge (n_1, n_2) in p, $(e(n_1), e(n_2))$ is an edge in t; (4) for any //-edge (n_1, n_2) in p, there is a path from $e(n_1)$ to $e(n_2)$ in t.

The result of applying a tree pattern p to an XML tree t is the set of subtrees of t rooted at nodes in:

$$\{e(\text{OUT}(p))|\ e \textit{ is an embedding of } p \textit{ into } t\}.$$

Definition 2.4. A tree pattern p_1 is *contained* in a tree pattern p_2 iff for any input tree t, $p_1(t) \subseteq p_2(t)$. We write this shortly as $p_1 \sqsubseteq p_2$. We say that p_1 is *equivalent* to p_2, and write $p_1 \equiv p_2$, iff $p_1(t) = p_2(t)$ for any input tree t.

The same notions are also used on *XP* expressions. A pattern p is said *minimal* if no pattern $p' \equiv p$ can have fewer nodes than p. Pattern minimization can be done in polynomial (linear) time for the *XP* fragment of XPath [Amer-Yahia et al. 2002].

For a tree pattern p, we refer to the path starting with ROOT(d) and ending with OUT(p) as the *main branch* of p. We refer to the set of nodes on this path as MBN(p). We say that a pattern is *linear* if it has no side branches. By MB(p) we denote the linear pattern that is isomorphic with the main branch of p. We call a *predicate subtree* of a pattern p any subtree rooted at a nonmain branch node.

Definition 2.5. A *mapping* between two tree patterns p_1, p_2 is a function h : NODES(p_1) $\rightarrow$ NODES(p_2) satisfying properties (2),(4) of an embedding (allowing the target to be a pattern) plus three others: (5) for any $n \in$ MBN(p_1), $h(n) \in$ MBN(p_2); (6) for any /-edge (n_1, n_2) in p_1, $(e(n_1), e(n_2))$ is a /-edge in p_2.

A *root-mapping* is a mapping that satisfies (1). An *output-mapping* is a mapping h such that h(OUT(p_1)) = OUT(p_2). A *containment mapping* denotes a mapping that is simultaneously a root-mapping and an output-mapping.

Previous studies [Amer-Yahia et al. 2002; Miklau and Suciu 2004] show that for two tree patterns p_1 and p_2, $p_2 \sqsubseteq p_1$ iff there is a containment mapping from p_1 into p_2.

Intersection. We consider in this paper the extension $XP^{\cap}$ of *XP* with respect to intersection, whose grammar is obtained from that of *XP* by adding the following rule:

$$\textit{ipath} ::= \textit{apath} \mid \textit{apath} \cap \textit{ipath}.$$

The result of applying an $XP^{\cap}$ pattern of the form $p_1 \cap \cdots \cap p_k$, where $p_1, \ldots, p_k$ are *XP* patterns, to an XML tree t is the set of subtrees of t rooted at nodes in:

$$\bigcap_i \{e(\text{OUT}(p))|\ e \textit{ is an embedding of } p_i \textit{ into } t\ \}_i.$$

By $XP^{\cap}$ expressions over a set of documents D we denote those that use only *apath* expressions that navigate inside D's documents.

As in Benedikt et al. [2005], a *code* is a string of symbols from Σ, alternating with either / or //.

Definition 2.6. An *interleaving* of a finite set of tree patterns $\mathcal{S}$ is any tree pattern p_i produced as follows.

(1) Let $M = \cup_{p \in S}$MBN(p).
(2) Choose a code i and a total onto function f_i that maps M into Σ-positions of i such that:

(a) for any $n \in M$, LABEL($f_i(n)$) = LABEL(n)
(b) for any $p \in \mathcal{S}$, f_i(ROOT(p)) is the first symbol of i,
(c) for any $p \in \mathcal{S}$, f_i(OUT(p)) is the last symbol of i,
(d) for any /-edge (n_1, n_2) of any $p \in \mathcal{S}$, i is of the form $\ldots f_i(n_1)/f_i(n_2)\ldots$,
(e) for any //-edge (n_1, n_2) of any $p \in \mathcal{S}$, i is of the form $\ldots f_i(n_1)\ldots f_i(n_2)\ldots$ (i.e., $f_i(n_1)$ appears before $f_i(n_2)$).

(3) Build the smallest pattern p_i such that:
(a) i is a code for the main branch of p_i,
(b) for any $n \in M$ and its image n' in p_i (via f_i), if a predicate subtree st appears below n then a copy of st appears below n', connected by the same kind of edge.

Two nodes n_1, n_2 from M are said to be *collapsed* if $f_i(n_1) = f_i(n_2)$, with f_i as given. The tree patterns p_i thus obtained are called *interleavings* of $\mathcal{S}$ and we denote their set by *interleave($\mathcal{S}$)*.

Example 2.7. One of the interleavings of v_1 and v_2 from Figure 1 is q_1, as v_1 has a //-edge between nodes n_{tr2} and n_{m1}, which allows the *tour* from v_2 to appear as a direct parent of *museum*.

Considering also unions of tree patterns, having straightforward semantics, one can prove the following intersection-union duality.

LEMMA 2.8. *For any set of XP queries $\mathcal{S} = \{q_1, \ldots, q_n\}$, the $XP^{\cap}$ expression $\cap_i q_i$ is equivalent to the union $\cup$interleave($\mathcal{S}$).*

The following also holds:

LEMMA 2.9. *A tree pattern is contained in a union of tree patterns iff it is contained in a member of the union. A tree pattern contains a union of patterns iff it contains each member of the union.*

The set of interleavings of a set of patterns $\mathcal{S}$ may be exponentially larger than $\mathcal{S}$. Indeed, it was shown that the $XP^{\cap}$ fragment is not included in *XP* (i.e, the union of its interleavings cannot always be reduced to one *XP* query by eliminating the redundant interleavings contained in others) and that an intersection may only be translatable into a union of exponentially many tree patterns [Benedikt et al. 2005].

View-based rewriting. Given a view v defined by an *XP* query over a document D, by $doc(v)$ we denote a document in which the topmost element is labeled with the view name and has as children the trees resulting from the evaluation of v over D. Note that such a document can be queried by *XP* expressions of the form $doc(v)/v\ldots$. For a set of views $\mathcal{V}$, defined by *XP* queries over a document D, by $D_{\mathcal{V}}$ we denote the set of view documents $\{doc(v)|v \in \mathcal{V}\}$. Given a query r, expressed in a rewrite language $\mathcal{L}_R$ (e.g., *XP* or $XP^{\cap}$), over the view documents $D_{\mathcal{V}}$, we define *unfold*(r) as the $\mathcal{L}_R$ query obtained by replacing in r each $doc(v)/v$ with the definition of v.

Given an *XP* query q and a finite set of *XP* views $\mathcal{V}$ over D, we look for an alternative plan r in $\mathcal{L}_R$, called a *rewriting*, that can be used to answer q. We define rewritings as follows.

Definition 2.10. For a given document D, an *XP* query q and *XP* views $\mathcal{V}$ over D, a *rewrite plan* of q using $\mathcal{V}$ is a query $r \in \mathcal{L}_R$ over $D_{\mathcal{V}}$. If *unfold*(r) $\equiv q$, then we also say r is a *rewriting* for q.

According to the definition above, a rewrite plan r in *XP* is of the form $doc(v_j)/v_j$, $doc(v_j)/v_j/p$ or $doc(v_j)/v_j//p$.

Similarly, according to the definition of $XP^{\cap}$, a rewrite plan r in $XP^{\cap}$ is of the form $r = (\bigcap_{i,j} u_{ij})$, for each u_{ij} being of the form $doc(v_j)/v_j$, $doc(v_j)/v_j/p_i$ or $doc(v_j)/v_j//p_i$. Note that such a query r is a rewriting (i.e., equivalent to q) iff

(1) each query $unfold(u_{ij})$ contains q, and
(2) by Lemmas 2.8 and 2.9, q contains all the tree patterns (interleavings) in $interleave(\{unfold(u_{ij})\})$.

Example 2.11. Revisiting Example 1.1, if the views expose persistent node ids, by using Lemmas 2.8 and 2.9 one can check that a rewriting of q using v_1 and v_2 is the following $XP^{\cap}$ expression over $\mathcal{D}_{\{v_1,v_2\}}$:

$$r = doc(v_1)/v_1 \cap doc(v_2)/v_2,$$

whose unfolding is

$$\begin{aligned} unfold(r) \;=\; & doc(T)//vacation//trip/trip[guide]//museum \cap \\ & doc(T)//vacation//trip//tour[schedule//walk]/museum. \end{aligned}$$

Intuitively, this holds because q_1 is one of the interleavings of v_1 and v_2 and all other interleavings are contained in q_1.

Further notation. We introduce now some additional notation, which will be first used in Section 7 and can be skipped until then.

A */-pattern* is a tree pattern having only /-edges in the main branch. A */-predicate* (respectively *//-predicate*) is a predicate subtree that is connected by a /-edge (resp. //-edge) to the main branch.

We will refer to main branch nodes of a pattern p by their *rank* in the main branch, i.e. a value in the range 1 to $|\text{MB}(p)|$, for 1 corresponding to $\text{ROOT}(p)$ and $|\text{MB}(p)|$ corresponding to $\text{OUT}(p)$. For a rank k, by $p(k)$ we denote any pattern isomorphic with the subtree of p rooted at the main branch node of rank k. By $node_p(k)$ we denote the node of rank k in the main branch of p.

A *prefix* p' of a tree pattern p is any tree pattern that can be built from p by setting $\text{ROOT}(p)$ as $\text{ROOT}(p')$, setting some node $n \in \text{MBN}(p)$ as $\text{OUT}(p')$, and removing all the main branch nodes that are descendants of n along with their predicates. For any rank k in p, by $cut(p,k)$ we denote the prefix of p having k main branch nodes. A *suffix* p' of a tree pattern p is any subtree of p rooted at a node in $\text{MBN}(p)$. A *sub-query* of p denotes a suffix of a prefix of p.

We associate a name to each predicate in a pattern p (in lexicographic order). For a given predicate P, by n_P we denote the main branch node that is parent of P in q. By r_P we denote P's position on the main branch, that is, the rank of the node n_P. By q_P we denote the pattern formed by the node n_P, as $\text{ROOT}(q_P)$, the pattern of P, and the edge connecting them. By $root_P$ we denote the node of p representing the root of P's pattern.

We also refer to the *tokens* of tree pattern p: more specifically, the main branch of p can be partitioned by its subsequences separated by //-edges, and each subpattern corresponding to such a subsequence is called a *token* of p. We can thus see a pattern p as a sequence of tokens (i.e., /-patterns) $p = t_1//t_2//\ldots//t_k$. We call t_1, the token starting with $\text{ROOT}(p)$, the *first token* of p. The token t_k, which ends by $\text{OUT}(p)$, is called the *last token* of p.

3. QUERY SET SPECIFICATIONS

We consider sets of XPath queries encoded using a grammar-like formalism, Query Set Specifications (QSS), similar to Petropoulos et al. [2003].

Definition 3.1. A *Query Set Specification* (QSS) is a tuple (F, Σ, P, S) where

(1) F is the set of tree fragment names,
(2) Σ, with $\Sigma \cap F = \emptyset$ is the set of element names,
(3) $S \in F$ is the start tree fragment name,
(4) P is a collection of expansion rules of the form
$f() \rightarrow tf$ or $f(X) \rightarrow tf$.
where f is a tree fragment name, tf is a tree fragment and X denotes the output mark. Empty rules, of the form $f \rightarrow$ (no tree fragment) are also allowed.
f is called the left-hand side (abbreviated as LHS) and tf is called the right-hand side (RHS) of the rule.

A tree fragment is a labeled tree that may consist of the following:

(1) element nodes, labeled with symbols from Σ,
(2) tree fragment nodes n labeled with symbols from F,
(3) edges either of child type, denoted by simple lines, or of descendant type, denoted by double lines,
(4) the output mark X associated to one node (of either kind).

In any rule, in the RHS one unique node may have the output mark (X) if and only if that rule has the output mark on the LHS.

As a notation convention, we serialize QSS tree fragments as *XP* expressions with an output mark (X), if present.

QSS expansions. A finite expansion (in short *expansion*) of a QSS $\mathcal{P}$ is any tree pattern p having a body obtained as follows.

(1) Starting from a rule $S(X) \rightarrow tf$.
(2) apply on tf the following expansion step a finite number of times until no more tree fragment names are left: for some node n labeled by a tree fragment name f, pick a rule defining f (i.e., f is the LHS) and replace n by the RHS of that rule; if n has the output mark, use only rules with LHS $f(X)$.
(3) set the node having the output mark as $\text{OUT}(p)$.

We say that p is *generated* by $\mathcal{P}$. Note that the set of expansions can be infinite if the QSS is recursive.

Definition 3.2 (*Expressibility and Support*). For an *XP* query q, a QSS $\mathcal{P}$, and a rewriting language $\mathcal{L}_R$ we say that

(1) q is *expressed* by $\mathcal{P}$ iff q is equivalent to an expansion of $\mathcal{P}$.
(2) q is *supported* by $\mathcal{P}$ in $\mathcal{L}_R$ iff there is a finite set $\mathcal{V}$ of *XP* queries generated by $\mathcal{P}$, with corresponding view documents $D_\mathcal{V}$, such that there is a rewriting of q formulated in $\mathcal{L}_R$ that navigates only in documents from $D_\mathcal{V}$.

The definition of support given above depends on the language $\mathcal{L}_R$ in which the rewritings can be expressed. If rewritings are expressed in *XP*, then all one can do is navigate inside one view. However, if the source exposes persistent node ids, it becomes possible to intersect view results. In this case, one can choose $\mathcal{L}_R$ to be $XP^\cap$ and use several views in more complex rewritings.

Example 3.3. The QSS $\mathcal{P}$ defined in the following generates queries returning information about museums that will be visited on a guided trip or as part of a tour in whose schedule there is also allotted time for taking a walk. Trips that appear nested

are secondary trips.

$$
\begin{aligned}
(\mathcal{P}) \qquad & f_0(X) \to doc(T)//vacation//f_1(X)\\
& f_1(X) \to trip/f_1(X)\\
& f_1(X) \to trip[guide]//museum(X)\\
& f_1(X) \to trip//tour[schedule//walk]/museum(X)
\end{aligned}
$$

It can be checked that v_1 and v_2 introduced before are among the expansions of $\mathcal{P}$. When considering v_1 and v_2 as user queries, we can also say they are expressed by $\mathcal{P}$.

Consider the following client query q_2, asking for museums that have temporary exhibitions and are visited in secondary trips:

$$q_2 : doc(T)//vacation//trip/trip[guide]//museum[temp].$$

q_2 is obviously not expressed by $\mathcal{P}$(there is no *temp* element node in $\mathcal{P}$). However, it is enough to filter the result of v_1 by the predicate *[temp]* to obtain the same result as q_2, hence q_2 is supported by $\mathcal{P}$:

$$q_2 \equiv doc(v_1)/v_1/museum[temp].$$

Considering the query q_1 of Example 1.1, one can check that q_1 cannot be answered by navigating into a single view. In the presence of persistent node ids, as discussed in Example 2.11, support of q_1 is witnessed by $\mathcal{P}$'s expansions v_1 and v_2.

Normalization. For ease of presentation, we introduce first some normalization steps on the QSS syntax. First, the set of tree fragment names that have the output mark (denoted *unary*) is assumed disjoint from those that do not have it (denoted *boolean*). Second, we equivalently transform all rules such that, in any RHS, tree fragments have depth at most 1, and the nodes of depth 1 can only be labeled by tree fragment names (i.e., a RHS is a tree fragment formed by a root and possibly some tree fragment children, connected by either /-edges or //-edges to the root). For that, we may introduce additional tree fragment names. After normalization, for l being a label in Σ, $c_1, \ldots, c_n$, $d_1, \ldots, d_m$ being two (possibly empty) lists of tree fragment names and g being a tree fragment name as well, any non-empty rule falls into one of the following cases:

$$
\begin{aligned}
f() &\to l[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()]\\
f(X) &\to l(X)[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()]\\
f(X) &\to l[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()]/g(X)\\
f(X) &\to l[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()]//g(X).
\end{aligned}
$$

For any fragment name f and rule $f(X) \to l[c_1() \ldots c_n, .//d_1(), \ldots, .//d_m()]$ *edge* $g(X)$, by v_f we denote any possible expansion of f via that rule. By v'_f we denote any pattern that can be obtained from the rule by (i) expanding g into the empty pattern, and (ii) expanding the c_is and the g_js in some (any) possible way. Note that v'_f has only one main branch node (the root).

Example 3.4. The result of normalizing the QSS $\mathcal{P}$ from Example 3.3 is the following:

$$
\begin{aligned}
& f_0(X) \to doc(T)//f_1(X), && f_1(X) \to vacation//f_2(X)\\
& f_2(X) \to trip/f_2(X), && f_2(X) \to trip[\,f_7()]//f_5(X)\\
& f_2(X) \to trip//f_3(X), && f_3(X) \to tour[\,f_4()]/f_5(X)\\
& f_4() \to schedule//f_6(), && f_5(X) \to museum(X)\\
& f_6() \to walk, && f_7() \to guide.
\end{aligned}
$$

4. EXPRESSIBILITY

We consider in this section the problem of expressibility: given a query q and a QSS $\mathcal{P}$ encoding views, decide if there exists a view v generated by $\mathcal{P}$ that is equivalent to q.

Conceptually, in order to test expressibility, one has to enumerate the set of views and, for each view, check its equivalence to q. This is obviously infeasible in our setting, since the set of views is potentially infinite. But the following observation delivers a naïve algorithm: only views that contain q have to be considered, and there are only finitely many distinct (with respect to isomorphism) candidates since containment mapping into q limits both the maximum length of a path (by the maximal path length in q) and the set of node labels (by the ones of q). Therefore, one can decide expressibility by enumerating all the candidate views and checking for each candidate if (a) it is equivalent to q, and (b) it is indeed an expansion of $\mathcal{P}$. However, this solution has limited practical interest beyond the fact that it shows decidability for our problem, since it is non-elementary in time complexity.

Our main contribution here is to provide a PTIME decision procedure for expressibility. The intuition behind our algorithm is the following. We do not enumerate expansions, and instead we group views and view fragments (which are assembled by the QSS to form a view) into *equivalence classes* with respect to their behavior in the algorithm for checking equivalence with q. There are fewer (only polynomially many) possible behaviors, and we manipulate such equivalence classes instead of explicit views or fragments thereof.

As a compact representation for equivalence classes, we use *descriptors*. Informally, we use two kinds of descriptors for views or view fragments:

(1) *mapping descriptors*, which record if some expansion of a tree fragment name maps into a subtree of q,
(2) *equivalence descriptors*, which record if some expansion of a tree fragment name is equivalent to a subtree of q.

The rest of this section is organized as follows.

We first observe that equivalence for tree patterns is reducible to equivalence for a different flavor of patterns, *boolean tree patterns* [Miklau and Suciu 2004]. These are tree patterns of arity 0 (no output node) that test if evaluating a pattern over an XML document yields an empty result or not. Following this observation, for presentation simplicity, we solve expressibility for Boolean tree patterns (Section 4.1).

Then, in Section 4.3, we show how expressibility for tree patterns (arity 1) can be reduced to expressibility for Boolean tree patterns.

4.1. Expressibility for Boolean Tree Patterns

We study in this section expressibility for Boolean tree patterns. Their semantics, based on the same notion of embedding, can be easily adapted from the case of arity 1: the result of applying a Boolean tree pattern p to an XML tree t is either the empty set $\emptyset$ or the set $\{\text{ROOT}(t)\}$. In the first case, we say that the result is *false*, in the latter, we say it is *true*. Containment and equivalence for Boolean tree patterns are also based on mappings, with the only difference that there is no output node.

In the remainder of this section all patterns (queries and views) are boolean tree patterns. A QSS will have either rules of the form $f() \rightarrow l[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()]$ or empty rules.

In order to clarify the role of descriptors and the equivalence classes they might stand for, let us first consider how one can test equivalence between a query q and view v. The classic approach for checking this is dynamic programming, bottom-up, using

Boolean matrices M that bookkeep mappings in both directions. $M(n_1, n_2)$ is *true* if the subtree rooted at n_1 contains the one rooted at n_2.

We prefer instead a variation on this approach, which will enable our PTIME solution. Since wildcard is not used, equivalence between a query q and a view v translates into q and v being *isomorphic* modulo minimization. Assuming that q is already minimized, this means that v has to be q plus some redundant branches, that is,

(1) q is isomorphic to (part of) v, that is, there is a containment mapping ψ from q into v, and the inverse ψ^{-1} is a partial mapping from v into q,
(2) the partial mapping ψ^{-1} can be completed to a containment mapping from v into q

In the above, no two nodes of q can have the same image under ψ. In other words, some nodes of v have an "equivalence role," and there must be one such node corresponding to each node of q, while the remaining nodes are redundant and it suffices to have only a "mapping role." This suggests that it is enough to build bottom-up only one matrix M, for containment from subtrees of v into subtrees of q, if in parallel we bookkeep in another matrix details about *equivalence* between subtrees. A field in the equivalence matrix, $E(n_1, n_2)$, for $n_1 \in \text{NODES}(v)$, $n_2 \in \text{NODES}(q)$, indicates that the subtree $v(n_1)$ is equivalent with the subtree $q(n_2)$.

With these two matrices, checking $v \equiv q$ by a bottom-up pass is straightforward, by applying the following steps until fix-point:

Assuming that $M(n_1, n_2)$ and $E(n_1, n_2)$ are *true* for any leaf nodes $n_1 \in \text{NODES}(v)$, $n_2 \in \text{NODES}(q)$ having the same label,

A) For each pair (n_1, n_2), $n_1 \in \text{NODES}(v)$, $n_2 \in \text{NODES}(q)$ having the same label, set $M(n_1, n_2)$ to *true* if:

(1) for each /-child n of n_1 there exists a /-child n' of n_2 s.t. $M(n, n') = true$,
(2) for each //-child n of n_1 there exists a descendant n' of n_2 s.t. $M(n, n') = true$.

B) For each pair (n_1, n_2), $n_1 \in \text{NODES}(v)$, $n_2 \in \text{NODES}(q)$ having the same label, set $E(n_1, n_2)$ and $M(n_1, n_2)$ to *true* if:

(1) for each /-child n of n_2 there exists a /-child n' of n_1 s.t. $E(n, n') = true$,
(2) for each //-child n of n_2 there exists a //-child n' of n_1 s.t. $E(n, n') = true$,
(3) for each /-child n of n_1 that was not referred to at step (1), there exists a /-child n' of n_2 s.t. $M(n, n') = true$,
(4) for each //-child n of n_1 that was not referred to at step (2), there exists a descendant n' of n_2 s.t. $M(n, n') = true$.

We are now ready to present our PTIME algorithm for expressibility. We will adapt the above approach for testing equivalence, which builds incrementally (bottom-up, one level at a time) the mapping and equivalence details, to the setting when views are generated by a QSS by expanding fragment names. We will use *mapping* and *equivalence descriptors* to record for each tree fragment name if some of its expansions witnesses equivalence with or existence of mapping into a part of the query.

Definition 4.1. For a fragment name f of a QSS $\mathcal{P}$, a *mapping descriptor* is a tuple $map(f, n)$, where $n \in \text{NODES}(q)$, indicating that f has an expansion v_f in $\mathcal{P}$ that contains the subtree of q rooted at node n.

An *equivalence descriptor* is a tuple $equiv(f, n)$, where $n \in \text{NODES}(q)$, indicating that f has an expansion v_f in $\mathcal{P}$ that is equivalent with the subtree of q rooted at node n.

Note that a descriptor $equiv(f, n)$ will also tell us where the expansion it stands for maps (or not) in q. In other words, once we have an equivalence descriptor for a fragment name expansion, we can infer *all* mapping descriptors for it.

Example 4.2. Suppose that the data source publishes a modified version of the QSS from Example 3.4, enforcing the possibility of taking a walk on trips that contain tours. This translates into replacing the last rule for f_2 with the rule (unnormalized):

$$f_2(X) \rightarrow trip[.//f_6()]//f_3(X).$$

A client interface generates and sends a query identical to v_2 of Example 2.2 to this source.

The proof of expressibility will consist in finding an equivalence descriptor for the root of the tree pattern. To infer the existence of this descriptor, we compute descriptors going bottom up in the pattern and in the normalized QSS from Example 3.4.

We start with the leaves, for which we find $d_1 = equiv(f_5, n_{m2})$ and $d_2 = equiv(f_6, n_{w1})$, $d_2' = map(f_6, n_{w1})$. Using d_2, we can infer the descriptor $d_3 = equiv(f_4, n_{s1})$, which, together with the descriptor for n_{m2}, enables a descriptor $d_4 = equiv(f_3, n_{to1})$. Since n_{w1} is a descendant of n_{tr3}, we can use the mapping descriptor d_2' and the equivalence descriptor d_4 to build a descriptor $equiv(f_2, n_{tr3})$. This in turn enables a descriptor $equiv(f_1, n_{v2})$, which leads to inferring a descriptor for the root: $equiv(f_0, n_{d2})$.

Thus we can check that expressibility holds, even if v_2 is not isomorphic to any expansion of the QSS (since it has no predicate on the node labeled with *trip*).

Our algorithm for testing expressibility will mimic the two steps (A) and (B) above, applying them instead on QSS rules and fragment nodes via descriptors. Given descriptors for the fragment names in the RHS, we will infer new descriptors for the fragment name on the LHS. The only notable difference with respect to the approach for checking equivalence is for steps (B.1) and (B.2). For a fragment name f and node $n \in \text{NODES}(q)$, fragment names children of f in a rule may have several $equiv$ descriptors, referring to different nodes of q. We must choose one among them in a way that does not preclude the inference of a descriptor $equiv(f, n)$, when one exists. For that, we will use a function tf-cover, which takes as input a set of nodes N, a set of tree fragment names C and an array L such that for every $n \in N$, $L(n) \subseteq C$. It returns *true* if there is a way to pick a distinct tree fragment name from each $L(n)$, for all $n \in N$. We detail this function in Section 4.2. It is based on a max-flow computation and its running time is $O((|C| + |N|) * |C|)$.

The computation of descriptors (algorithm findDescExpr) starts with productions without tree fragment nodes on the RHS and continues inferring descriptors until a fixed point is reached. (This is close in spirit to bottom-up parsing as in the CYK algorithm [Hopcroft and Ullman 1979] or to bottom-up Datalog evaluation [Abiteboul et al. 1995].). It runs in polynomial time because (a) there are only polynomially many descriptors (their number is proportional to the size of the QSS multiplied by the size of the query), and (b) each incremental, bottom-up step for inferring a new descriptor runs in polynomial time.

THEOREM 4.3. *A Boolean tree pattern q is expressed by a QSS $\mathcal{P}$ iff* findDescExpr*$(q, \mathcal{P})$ outputs a descriptor $equiv(S, \text{ROOT}(q))$, for S being the start fragment name of $\mathcal{P}$.* findDescExpr *runs in polynomial time in the size of the query and of the QSS.*

PROOF. The fact that the input query q is minimized is important for our algorithm. Under this assumption, a view that is equivalent to the input query has to be isomorphic to it, modulo minimization of the view (i.e., redundant branches of the view). The steps (A) and (B) of the matrix-based approach for explicitly given views are then sound and complete for checking equivalence. Regarding the corresponding steps on the QSS program, the only notable difference is for steps (B.1) and (B.2) (step (E) in findDescExpr). We no longer have explicit view nodes/fragment and the query fragments to which they are equivalent, and tree fragment names appearing in a QSS rule may be expandable

ALGORITHM: findDescExpr(q, $\mathcal{P}$)

Input transformation: minimize q, normalize $\mathcal{P}$.

A. Start with an empty set of descriptors R.
B. For each rule $f() \rightarrow ()$, node $n \in$ NODES(q), add to R the descriptor $map(f, n)$.
C. For each rule $f() \rightarrow l$ (i.e., the RHS has only one node) and each node $n \in$ NODES(q) labeled by l, add to R the descriptors $equiv(f, n)$ and $map(f, n)$.

Repeat until R unchanged:

D. For each rule $f() \rightarrow l[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()]$, add to R a descriptor $map(f, n)$ if $n \in$ NODES(q) is labeled by l and

(1) for each fragment name c_i there exists a descriptor $map(c_i, n')$ s.t. n' is /-child of n,
(2) for each fragment name d_j there is a descriptor $map(d_j, n')$ s.t. n' is descendant of n.

E. for each rule $f() \rightarrow l[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()]$, add to R the descriptor $equiv(f, n)$, for $n \in$ NODES(q), and all $map(f, n')$ descriptors, for $n' \in$ NODES(q), that can be inferred from it when

(1) tf-cover(N_1, C, L) returns *true*, where N_1 is the set of /-children of n, $C \subseteq \{c_1, \ldots, c_n\}$ is the set of fragment names that have a descriptor $equiv(c_i, n')$ for $n' \in N_1$ and, for each $n' \in N_1$, $L(n') \subseteq C$ is the set of fragments names that have a descriptor $equiv(c_i, n')$.
(2) tf-cover(N_2, D, L) returns *true*, where N_2 is the set of //-children of n, $D \subseteq \{d_1, \ldots, d_m\}$ is the set of fragment names that have a descriptor $equiv(d_j, n')$ for $n' \in N_2$ and, for each $n' \in N_2$, $L(n') \subseteq D$ is the set of fragments names that have a descriptor $equiv(d_j, n')$.
(3) for each fragment name $c_i \notin C$, there is a descriptor $map(c_i, n')$ s.t. n' is /-child of n,
(4) for each fragment name $d_j \notin D$ there exists a descriptor $map(d_j, n')$ s.t. n' is descendant of n.

into several different view fragments. This is why in step (E) of findDescExpr, for a rule r, fragment name f and query node n, a given tree fragment name c_i or d_j might be candidate to cover (via an *equiv* descriptor) not just one but various /-children of n. The question then is which such node we pick to pair with the tree fragment name. In order to remain polynomial time, we must avoid trying all possible pairings of tree fragment names appearing in the RHS of r with nodes children of n. We thus check in bulk whether there exists such a pairing, via *equiv* descriptors, and this is the role of the tf-cover function. Given "links" between n's child nodes and r's RHS tree fragment names, we check if there exists a pairing yielding a total cover, i.e., a flow of size k, for k being the number of nodes. □

Remark. The fact that the input query q is minimized, which implies that no two nodes of q can have the same image under the ψ function we have described, allows us to avoid a bottom-up approach that might also have to bookkeep mappings from the query into the views. Such an approach would require descriptors that pair *a set of subtrees* of q with an expansion, leading to a worst-case exponentially large space for descriptors.

4.2. The Function tf-cover

We define here the helper function tf-cover, used in the algorithms for deciding expressibility. tf-cover takes as input a set of nodes N, a set of tree fragment names C and an array L such that for every $n' \in N$, $L(n') \subseteq C$. It returns true if there is a way to pick a distinct tree fragment name from each $L(i)$, for all $i \in N$.

The function is implemented by solving the following max-flow problem with integer values. The flow network has a source s and a sink t. Suppose $C = \{c_1, \ldots c_n\}$ and $L = \{L_1, \ldots L_k\}$. There are edges with capacity 1 from s to n nodes $c_1, c_2, \ldots c_n$. There are also k nodes $L_1, \ldots L_k$ and for each c_j such that $c_j \in L_i$, there is an edge with

capacity 1 from c_j to L_i. Finally, there is an edge with capacity 1 from each L_i node to the sink t.

As shown in Cormen et al. [2001], if all capacities are integers, the Ford-Fulkerson algorithm will find a max-flow that assigns an integer to every edge.

If the maximum flow returned is less than k (at least one $L(i)$ is not "covered"), tf-cover returns false. Otherwise (the max-flow is k), it returns true. If we want to also keep a view that witnesses expressibility, it is enough to know what edges between a c_j and an L_i have a flow of value 1. This is possible because the Ford-Fulkerson algorithm gives the value of the flow on each edge.

4.3. Expressibility for Tree Patterns

We now consider expressibility for standard tree pattern queries (patterns with an output node).

It is well known from previous literature that problems such as tree pattern containment and equivalence reduce to containment, respectively equivalence, for Boolean patterns. This is based on the following translation: let s be a new label (from *selection*), for a tree pattern p let p_0 denote the Boolean tree pattern obtained from p by (i) adding a /-child labeled s below the output node of p, and (ii) removing the output mark. From Miklau and Suciu [2004], for two tree patterns p and p', we have that $p \equiv p'$ iff $p_0 \equiv p'_0$.

A similar transformation can be applied for expressibility. Given a QSS $\mathcal{P}$, let $\mathcal{P}_0$ be the QSS obtained from $\mathcal{P}$ by (i) plugging a /-child labeled s below each node having an explicit label and the output mark, and (ii) making all rules and tree fragment names Boolean by removing their output mark. $\mathcal{P}_0$ generates Boolean tree patterns and, since $\mathcal{P}$'s sets of unary and Boolean tree fragment names were assumed disjoint, $\mathcal{P}_0$'s expansions have exactly one s-labeled node. We can prove the following.

THEOREM 4.4. *A tree pattern query q is expressed by a QSS $\mathcal{P}$ iff the Boolean tree pattern q_0 is expressed by the QSS $\mathcal{P}_0$.*

PROOF SKETCH. Follows from the reduction of containment (respectively equivalence) of tree pattern queries to containment (resp. equivalence) of boolean tree patterns. □

From Theorems 4.3 and 4.4 we have the following corollary.

COROLLARY 4.5. *Expressibility for tree pattern queries and QSS programs can be decided in PTIME.*

5. SUPPORT

For the problem of support, the fact whether the source enables persistent node ids (that are then exposed in query results) or not has a significant impact on the rewrite plans one can build. In both settings, with or without node ids, rewriting under an explicitly listed set of views has been studied in previous literature. We will now revisit them for support.

In the first setting, the identity of the nodes forming the result of a query is not exposed in the results. By consequence, the only possible rewrite plans consist in accessing a view result and maybe navigating inside it (via query *compensation*). This setting was considered in Xu and Özsoyoglu [2005], and the rewriting problem was shown to be in PTIME for *XP*. We study support in the absence of ids in Section 5.1. Our main result here is that support reduces to expressibility, which allows us to reuse the PTIME algorithm given in Section 4.

In the second setting, for which rewriting under an explicit set of views was studied in [Cautis et al. 2008], data sources expose persistent node ids. This enables more complex rewrite plans, in which the *intersection* of view results plays a crucial role. We

revisit this setting, for the support problem, in Section 6. As our general approach, we will apply the same kind of reasoning that was used for expressibility. We will group views into equivalence classes with respect to crucial tests for support and we will manipulate classes (encoded as *view descriptors*) instead of explicit views. This will enable us to avoid the enumeration of a potentially large space of views and rewrite plans.

Remark. As in the case of expressibility, naïve algorithms for support can be obtained by an enumeration approach: there are only finitely many distinct (w.r.t. isomorphism) views that might be useful for support (in either setting), since their mapping into q limits both the maximum length of a path (by the maximal path length in q) and the set of node labels (by the ones of q). One can decide support by enumerating all the possible candidate views, checking for each candidate if (a) it contains q (or a prefix of it) and (b) it is indeed an expansion of the QSS and, finally, for the set of views $\mathcal{V}$ given by these tests, check if it witnesses support. However, this solution has limited practical interest beyond the fact that it shows decidability for support, since it is non-elementary in time complexity.

5.1. Support in the Absence of Ids

When persistent identifiers are not exposed, a rewrite plan consists in accessing a view's result and maybe navigating inside it, and this navigation is called *compensation*. This is why expressibility and support in the absence of ids remain strongly related, as support simply amounts to finding a candidate view v which, via compensation, becomes equivalent with the input query.

Let us first introduce as notation for this operation the compensate function, which performs the concatenation operation from Xu and Özsoyoglu [2005], by copying extra navigation from the query into the rewrite plan. For a view $v \in XP$, an input query q, and a main branch rank k in q, compensate(v,q,k) returns the query obtained by deleting the first symbol from $x = xpath(q(k))$ and concatenating the rest to v. For instance, the result of compensating v = a/b with x = b[c][d]/e is the concatenation of a/b and [c][d]/e, that is, a/b[c][d]/e.

We can reformulate the result from previous literature as follows.

THEOREM 5.1 ([XU AND ÖZSOYOGLU 2005]). *Given a set of explicit views $\mathcal{V}$, a query q can be answered by $\mathcal{V}$ if and only if there exists a view v and main branch rank k in q such that compensate(v,q,k) $\equiv q$.*

Going now to views encoded as QSS expansions, we reduce the problem of support to expressibility, following the idea that support amounts to expressibility by a certain "compensated" specification.

From a given QSS $\mathcal{P}$, we will build a new QSS that generates, besides $\mathcal{P}$'s expansions, all their possible compensated versions with respect to q. More precisely, given an input query q and a QSS $\mathcal{P}$, let $comp(\mathcal{P}, q)$ denote the QSS obtained from $\mathcal{P}$ as follows:

For any rule yielding the output node, that is, of the form

$$f(X) \rightarrow l(X)[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()],$$

for each rank k in q, add a new rule, of the form (with a little departure from the normalized QSS syntax):

$$f(X) \rightarrow compensate(l[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()], q, k).$$

We can prove the following.

THEOREM 5.2. *A query q is supported in XP by a QSS $\mathcal{P}$ if and only if it is expressed by the QSS $comp(\mathcal{P}, q)$.*

PROOF. Follows easily from the restrictive "shape" of rewrite plans in the absence of node ids, namely one view plus eventually compensation with a prefix of q. It is straightforward that, from the views generated by $\mathcal{P}$, the new program $comp(\mathcal{P}, q)$ generates all their compensated versions, with any possible prefix of the input query q. If an equivalent rewrite plan using $\mathcal{P}$ exists, then it will be one of the views generated by $comp(\mathcal{P}, q)$. □

Example 5.3. An example of support in the absence of persistent ids has already been given in Example 3.3: q_2 can be rewritten by compensating v_1 with a *temp* predicate.

6. SUPPORT IN THE PRESENCE OF IDS

We consider in this section the problem of support in the presence of node ids, denoted in the following *id-based support*. First, deciding the existence of a rewriting for an *XP* query becomes coNP-hard even under an *explicit* set of *XP* views, as shown by Cautis et al. [2008].

THEOREM 6.1 ([CAUTIS ET AL. 2008]). *In the presence of ids, testing if an XP query can be rewritten using explicitly listed views is coNP-hard.*

Intuitively, the rewriting problem becomes hard because in rewrite plans that use (intersect) multiple views one may have to consider exponentially many interleavings of those views. This is a consequence of Lemmas 2.8 and 2.9.

As corollary, it follows immediately that the same lower-bound holds for id-based support.

COROLLARY 6.2. *Id-based support for XP is coNP-hard.*

Since our focus is on efficient algorithms for support, we next investigate the tightest restrictions for tractability. We consider the fragment of *extended skeletons* (XP_{es}), for which the rewriting problem was shown tractable in Cautis et al. [2008]. The restrictions imposed by the XP_{es} fragment on the input query were shown to be necessary for tractability, as their relaxation leads to coNP-hardness. It is therefore natural to ask whether the support problem is also tractable for input queries from this fragment. Note that one cannot do better, that is, obtain a decision procedure for queries outside this fragment, since the problem of support subsumes the rewriting problem.

The remainder of this article is thus dedicated to studying support for extended skeletons, focusing on efficient (PTIME) solutions that are sound in general (i.e., for any *XP* input query) and complete under fairly general conditions, and this without restricting the language of views (which remains *XP*).

We show that id-based support exhibits a complexity dichotomy: the sub-fragment of XP_{es} representing queries that have at least one //-edge in the main branch, denoted hereafter *multitoken*, continues to be in PTIME (Theorem 7.22), but the complementary subfragment that represents queries with only /-edges in the main branch, denoted hereafter *single-token*, interestingly, is NP-hard (see Theorem 8.1).

The fragment of multitoken queries is particularly useful in practice since often, for reasons such as conciseness or generality in the presence of schema heterogeneity, one does not want to write in a query all the navigation steps over a document (may skip some steps by //). After defining the fragment of extended skeletons, we consider in Section 7 support for multitoken queries and, in Section 8, support for single-token queries.

Extended skeletons (XP_{es}). Informally, this fragment limits the use of //-edges in predicates, in the following manner: a token t of a pattern p will not have predicates

that have //-edges and that may become redundant because of descendants of t and their respective predicates in some interleaving p might be involved in.

Let us first introduce some additional terminology. By a *//-subpredicate st* we denote a predicate subtree whose root is connected by a //-edge to a linear /-path l that comes from the main branch node n to which st is associated (as in $n[\ldots[.//st]]$). l is called the *incoming /-path* of st and can be empty.

Extended skeletons are patterns p having the following property: for any main branch node $n \neq \text{OUT}(p)$ and //-subpredicate st of n, there is no mapping (in either direction) between the code of the incoming /-path of st and the one of the /-path following n in the main branch (where the empty code is assumed to map in any code).

For instance, expressions such as a[b//c]/d//e or a[b//c//d]/e//d are in XP_{es}, while a[b//c]/b//d, a[b//c]//d, a[.//b]/c//d or a[.//b]//c are not. XP_{es} does not restrict in any way the usage of //-edges in the main branch or the usage of predicates with /-edges only.

Note that the above definition imposes no restrictions on predicates of the output node. This relaxation was not present in Cautis et al. [2008]'s definition of extended skeletons but it is easy to show that it does not affect any of the results that were obtained with the more restrictive definition. This is because there is only one choice for ordering the output nodes in interleavings of an intersection; they are collapsed into one output node.

7. MULTI-TOKEN QUERIES

We consider now id-based support for XP_{es} multi-token queries. For presentation simplicity, we first limit the discussion to rewrite plans that are intersections of views (no compensation before the intersection step). General $XP^{\cap}$ plans, that is, intersections of possibly compensated views, are considered in Section 7.5.

As for expressibility, we think of views as grouped into equivalence classes with respect to crucial tests for support. We manipulate such classes, represented by *view descriptors*, instead of explicit views, avoiding the enumeration of potentially many views and plans. As a QSS constructs views by putting together fragments, we construct view descriptors from *fragment descriptors*, which represent equivalence classes for fragment expansions.

This section is organized as follows. In order to clarify the role of view descriptors and the equivalence classes they stand for, we first revisit in Section 7.1 the PTIME algorithm of [Cautis et al. 2008] for deciding if an XP_{es} multi-token query q can be rewritten by an intersection of explicit XP views $\mathcal{V}$ already known to contain q. That algorithm was based on applying DAG-pattern rewrite steps towards a tree pattern and then checking equivalence with q. We reformulate it into an algorithm (testEquiv) that applies individual tests on the view definitions instead. Then, in Section 7.2, we introduce equivalence classes for views with respect to the tests of testEquiv, and *view descriptors* as a means to represent such classes. We reformulate testEquiv into a new algorithm, testEquivDesc, that runs on view descriptors instead of explicit view definitions. Then, in Section 7.3 we give a PTIME sound and complete algorithm for computing descriptors for the expansions of a QSS.

7.1. Rewriting with an Explicit Set of Views

Let the input XP_{es} multitoken query q be of the form $q = ft//m//lt$, where ft denotes the first token, lt denotes the last token and m denotes the rest (m may be empty).

Let $\mathcal{V} = \{v_1, \ldots, v_n\}$ denote a set of XP views such that $q \sqsubseteq v_i$ for each v_i. Let each view v_i be of the form $v_i = ft_i//m_i//lt_i$.

For an XP query v, by its *extended skeleton*, we denote the XP_{es} query obtained by pruning out all the //-subpredicates violating the XP_{es} condition. We can prove the following auxiliary lemma.

LEMMA 7.1. *An XP_{es} query is equivalent to an intersection of views iff equivalent to the intersection of their extended skeletons.*

PROOF. For an *XP* query v, let $s(v)$ denote the XP_{es} query obtained by pruning out the //-subpredicates that do not obey the XP_{es} condition. $s(v)$ is called the extended skeleton of v. For showing that a query q from XP_{es} is equivalent to an intersection of *XP* views $v_1, \ldots, v_n$ if and only if q is equivalent to the intersection of their extended skeletons, the *if* direction is immediate since $\cap_i v_i \sqsubseteq \cap_i s(v_i)$. For the *only if* direction, it suffices to see that since q is an extended skeleton, any containment mapping from q into $\cap_i v_i$ will actually use only parts that are not violating the XP_{es} condition. This means that a containment mapping from q into $\cap_i v_i$ gives also a containment mapping from q into $\cap_i s(v_i)$. □

By Lemma 7.1, without loss of generality all views are assumed in the rest of the Section 7.1 from XP_{es}.

Notation. Let $ft_{\mathcal{V}}$ denote the query obtained by "combining" the first tokens $ft_1, \ldots, ft_n$ as follows: start by coalescing the roots, then continue coalescing top-down any pair of main branch nodes that have the same parent and label. This process yields a tree because each first token ft_i maps in the first token of q, ft, hence each MB(ft_i) is a prefix of MB(ft). Let $lt_{\mathcal{V}}$ denote the query obtained by "combining" $lt_1, \ldots, lt_n$ similarly: start by coalescing the output nodes, then continue by coalescing bottom-up any pair of main branch nodes that have a common child and the same label.

Example 7.2. For instance, for two views $\mathcal{V} = \{v', v''\}$,

$$v' = doc(T)/vacation/trip[guide]//tour/museum,$$
$$v'' = doc(T)/vacation[.//walk]//museum[gallery],$$

the result of combining their first tokens, respectively last tokens is

$$ft_{\mathcal{V}} = doc(T)/vacation[.//walk]/trip[guide],$$
$$lt_{\mathcal{V}} = tour/museum[gallery].$$

Given MB(ft), MB(lt), if there exists a minimal (nonempty) prefix of MB(lt) that is isomorphic with a suffix of MB(ft), let MB(lt)$'$ denote the pattern obtained from MB(lt) by cutting out this prefix. Then, let l_q denote the linear pattern MB(ft)/MB(lt)$'$. If l_q is undefined by the above, by convention it is the empty pattern.

Example 7.3. For instance, for the query

$$q = doc(T)/vacation[.//walk]/tour//tour/museum,$$

l_q is welldefined, as $l_q = doc(T)/vacation/tour/museum$.

Given MB(ft) and MB(m), if there exists a minimal (nonempty) suffix of MB(ft) that is isomorphic with a prefix of MB(m), let MB(ft)$_m$ denote the pattern obtained from MB(ft) by cutting out this suffix. If MB(ft)$_m$ is undefined by the above, by convention it is the empty pattern. Similarly, given MB(lt) and MB(m), if there exists a minimal (nonempty) prefix of MB(lt) that is isomorphic with a suffix of MB(m), let MB(lt)$_m$ denote the pattern obtained from MB(lt) by cutting out this prefix. If MB(lt)$_m$ is undefined by the above, by convention it is the empty pattern.

We are now ready to give our reformulation of the PTIME algorithm of Cautis et al. [2008], which will test if $\cap\mathcal{V} \sqsubseteq q$. By Lemma 2.9, q must contain each possible interleaving i of the set $\mathcal{V}$ or, in other words, for each $i \in interleave(\mathcal{V})$ the following should hold:

(1) the first token of q can be mapped in the first token of i such that the image of ROOT(q) is ROOT(i),
(2) the last token of q can be mapped in the last token of i such that the image of OUT(q) is OUT(i),
(3) the images of these two tokens in i are disjoint,
(4) the intermediary part m (if nonempty) of q can be mapped somewhere between these two images in i.

ALGORITHM: testEquiv($\mathcal{V}$, q)

```
1  let each v_i = ft_i//m_i//lt_i, let q = ft//m//lt
2  compute the patterns ft_V, lt_V, l_q, MB(ft)_m and MB(lt)_m
3  if ft_V ≡ ft and lt_V ≡ lt
4     then if m is empty
5             then for each v_i in V
6                     do if MB(v_i) does not map into l_q
7                           then return true
8             else (m non-empty) for each v_j in V
9                     do if v_j can be seen as prefix_j//m'//suffix_j s.t.
10                          m' ≡ m
11                          prefix_j root-maps into ft, suffix_j output-maps into lt
12                          MB(prefix)_j does not root-map into MB(ft)_m
13                          MB(suffix)_j does not output-map into MB(lt)_m
14                          then return true
```

We can prove the following.

THEOREM 7.4. *For a multi-token XP query q and a set of XP views $\mathcal{V}$,* testEquiv *is a sound PTIME procedure for testing $q \equiv \cap\mathcal{V}$.*

PROOF. Line 3 ensures that any interleaving i starts by a /-pattern into which ft has a containment mapping and ends by a /-pattern into which lt has a containment mapping.

Let us now consider the case when q's intermediary part m is empty, that is, q is of the form $ft//lt$.

In this case, condition (line 6) guarantees that in any interleaving i the images of ft and lt (by the containment mappings mentioned above) are disjoint: If l_q is the empty pattern, this is immediate. Otherwise, since $l_q \not\sqsubseteq$ MB(v_i), this means that (a) no interleavings with main branch l_q can be built, and furthermore (b) no interleavings with an even shorter main branch (that would be obtained by cutting a bigger prefix from MB(lt)) can be built either. By the fact that these two containment mappings have disjoint images, their union yields a containment mapping from q into i, hence $i \sqsubseteq q$.

We now consider the case when m is not empty.

For this case, besides the fact that in any interleaving i the images of ft and lt must be disjoint, the rest of q (the m part) must also map somewhere between these images. All this is guaranteed by the conditions of lines 9–13.

First, v_j has a sub-query m' which, considered in isolation, is equivalent (i.e., isomorphic modulo minimization) with m. Then, conditions (lines 12–13) imply that in any interleaving i of the views, nodes from the m' part of v_j cannot be collapsed with nodes from the first or last tokens of the various views. More precisely, they imply that the minimal prefix (resp. suffix) of MB(lt) (resp. MB(ft)) cannot be collapsed with the part of MB(m') to which it is isomorphic (by the definition of MB(ft)$_m$ and MB(lt)$_m$). By the minimality property, if there are some coalescing opportunities, the ones that are ruled out here must be among them. Hence the part $ft_{\mathcal{V}}$ (by which i starts) and the

part $lt_{\mathcal{V}}$ (by which i ends) are disjoint, and there are at least $|m'|$ main branch nodes in between.

Then, the rest of q, m, will also map in between, since m maps in any pattern resulting from the interleaving of m' with other view parts (we can compose the mapping from m to m' with the onto function by which i is built). It follows easily that q has a containment mapping into any interleaving i of $\cap\mathcal{V}$.

We now consider how one can verify conditions (lines 6, 9–13) in polynomial time. For (line 6), the non-existence of a containment mapping between two linear paths could be easily translated into a containment mapping test.

Then, conditions (lines 9–13) amount to the following:

(1) finding the views that have a subquery equivalent to m (an equivalence test) and, for each of them,
(2) checking the nonexistence of the two mappings (even though $prefix_j$ root-maps into ft, hence MB($prefix_j$) also root-maps into MB(ft), and $suffix_j$ output-maps into lt, hence MB($suffix_j$) also output-maps into MB(ft)).

The first item is immediate. Then, for lines 12–13, since we are dealing again with linear patterns, testing if the two mappings fail can be done using a bottom-up (in the case of MB($suffix_j$)) respectively top-down (in the case of MB($prefix_j$)) procedure as the one described in Section 7.3, advancing one token at a time. □

For input queries from XP_{es} we can also prove completeness.

THEOREM 7.5. *For an XP_{es} multitoken query q and a set of XP views $\mathcal{V}$,* testEquiv *is complete for testing $q \equiv \cap\mathcal{V}$.*

PROOF. With respect to the approach of Cautis et al. [2008], our algorithm testEquiv does not apply rewriting rules on an intersection (denoted DAG pattern) of tree patterns (views), but verifies directly on the views conditions that are necessary for the existence of a rewriting.

For an XP query v, let $s(v)$ denote the XP_{es} query obtained by pruning out the //-subpredicates not obeying the XP_{es} condition. $s(v)$ is called the extended skeleton of v.

First, we rely on the result of Lemma 7.1, which guarantees that a query q from XP_{es} is equivalent to an intersection of XP views $v_1, \ldots, v_n$ if and only if q is equivalent to the intersection of their extended skeletons.

Then, if the conditions of line 3 do not hold, one can build interleavings i of the views that are not contained into q, for which either ft does not root map into i's first token or lt does not root map into i's last token or both. This, by Lemmas 2.8 and 2.9, would imply that q is not supported.

If q has only 2 tokens (i.e., the middle part m is empty), then MB(q) is of the form MB(ft)//MB(lt). If the test of line 6 does not hold, it means that l_q is nonempty and, moreover, all the main branches of the views have containment mappings into it. But these containment mappings point to an interleaving i of the views having a main branch shorter in length than MB(q), i.e. $|\text{MB}(i)| < |\text{MB}(q)|$. This implies that q does not contain i, hence q cannot have a rewriting using the intersection of the views.

Assuming now that q has more than 2 tokens (m is nonempty), each of the views v_i can be seen as $prefix_j//mid_j//suffix_j$, where $prefix_j$ root-maps into ft, $suffix_j$ output-maps into lt and mid_j maps into m while m may or may not map back into mid_j. Now, if m does not map back into any of the mid_j patterns, we can again exhibit an interleaving i of the views $v_1, \ldots, v_n$ not contained into q. This is based on the following.

LEMMA 7.6. *If the XP_{es} patterns are of the form $v_i = ft//p_i//lt$, $1 \leq i \leq n$, then $\cap_i v_i$ is equivalent to one tree pattern (one of its interleavings) iff there is a query among them, v_j, having an intermediary part p_j such that all other p_i map into p_j.*

The lemma implies that the intersection $\cap_i v_i$ is equivalent to the tree pattern q only if at least one middle part mid_j is equivalent to m. Otherwise, an interleaving i of $v_1, \ldots, v_n$ contradicting support can be built by looking at the intersection $\cap_j ft//mid_j//lt$ and its interleavings.

Without loss of generality, let us assume there is only one such view v_j of the form $v_j = prefix_j//mid_j//suffix_j$, with $mid_j \equiv m$. Let us also assume that $\text{MB}(ft)_m$ is not empty, hence there is an overlap between $\text{MB}(ft)$ and $\text{MB}(m)$ and that $\text{MB}(prefix_j)$ does root-map into $\text{MB}(ft)_m$. We use the overlap to construct an interleaving i that cannot be contained into q; the construction will be detailed in the remainder of this proof.

By the above assumptions, $\text{MB}(v_j)$ will have a containment mapping in the main branch of a query q' obtained from q by using the overlap of ft with m, i.e. coalescing the end of the former with the start of the latter. q' is of the form $ft'//m'//lt$, where ft' is ft plus maybe some other /-steps, and m' is only a suffix of m (more precisely, m minus its first token). It is easy to see that $\text{MB}(q') \not\sqsubseteq \text{MB}(q)$ since the former has fewer main branch nodes but, by the assumptions, $\text{MB}(q') \sqsubseteq \text{MB}(v_j)$.

Finally, we consider the intersection

$$I = (\cap_{i \neq j} ft'//mid_i//lt) \cap q'.$$

We have $ft'//m//lt \sqsubset I \sqsubset \cap_j v_j$. Each mid_i maps into m, but m does not have an inverse mapping, while m' is a proper suffix of m. By Lemma 7.6, it follows that

$$I \not\equiv ft'//m//lt,$$

hence $I \not\sqsubset ft'//m//lt$. This means that the tree patterns in I must have some interleaving $i = ft'//mid//lt$ that is not contained in $ft'//m//lt$, hence m does not map into mid. But this i will not be contained into $q = ft//m//lt$ either, since $ft//m$ cannot be mapped into $ft'//mid$. This is because at most m except its first token (call it t) can be mapped in the mid part but not m entirely, while $ft//t$ cannot be mapped into ft' (the latter was obtained by coalescing some nodes of t with nodes of ft, hence has fewer main branch nodes than $ft//t$).

Since $i \sqsubset \cap_j v_j$ but $i \not\sqsubset q$, we have that $\cap_j v_j \not\sqsubseteq q$.

We can deal in similar manner with the general case when several views v_j might be such that $mid_j \equiv m$ and when the second condition of Line 13 is the one that is not verified by some of them. □

7.2. View Descriptors

We detail now how one can perform the tests of algorithm testEquiv even when abstracting away from the view definitions. The key idea is that the complete definitions are not needed but only the details used in these tests. With respect to these details, views can be seen as grouped into equivalence classes and views from the same class will be equally useful in the execution of the algorithm. This idea will be exploited by our *view descriptors*. We then reformulate testEquiv in terms of descriptors in algorithm testEquivDesc. More precisely, assuming we are dealing with expansions of a QSS $\mathcal{P}$ with start fragment name S,

For line 3 *of* testEquiv. For the part $ft_v \equiv ft$: a *first-token descriptor* will be a tuple **ft**(**S**, **p**), where p denotes any pattern that can be built from a prefix of q's first token ft by removing all its predicates, except eventually for one. Such a descriptor indicates

that there exists an expansion v s.t. $q \sqsubseteq v$ and v's first token is of the form p, plus eventually other predicates (ignored in the descriptor). These descriptors represent partitions (equivalence classes) of the space of views containing q w.r.t. their first tokens and the predicates on them. Each view belongs to at least one such class, but may belong to several of them (for different choices of predicates).

For the part $lt_{\mathcal{V}} \equiv lt$: a *last-token descriptor* is a tuple $\mathbf{lt}(\mathbf{S}, \mathbf{p})$, where p denotes any pattern that can be built from a suffix of q's last token lt by removing all its predicates, except eventually for one. Such a descriptor says that there is an expansion v s.t. $q \sqsubseteq v$ and v's last token is of the form p, plus eventually other predicates.

It is easy to see that the **ft** and **lt** view descriptors allow us to compute the patterns $ft_{\mathcal{V}}$ and $lt_{\mathcal{V}}$, provided they verify $ft_{\mathcal{V}} \equiv ft$ and $lt_{\mathcal{V}} \equiv lt$, without requiring the actual first and last tokens. The domain of these descriptors is quadratic in the size of q.

For line 6 *of* testEquiv. An *l-descriptor* is a tuple $\mathbf{l}(\mathbf{S})$, indicating that there exists an expansion v s.t. $q \sqsubseteq v$ and $l_q \not\sqsubseteq \mathrm{MB}(v)$. (This type of descriptor is an alias for the condition of line 6, denoting a partition of the space of views in two complementary classes.)

For lines 9–13 *of* testEquiv. An *m-descriptor* is a tuple $\mathbf{m}(\mathbf{S})$, indicating that there exists an expansion v verifying $q \sqsubseteq v$ and all the conditions of lines 9–13.

We now reformulate testEquiv into an algorithm that runs on a set of view descriptors $\mathcal{D}$, instead of the explicit views $\mathcal{V}$ to which they correspond. Unsurprisingly, the new algorithm follows closely the steps of testEquiv, since descriptors are tailored to its tests.

ALGORITHM: testEquivDesc($\mathcal{D}, q$)

```
1 from all descriptors ft(S, p) ∈ D compute the pattern ft_V
2 from all descriptors lt(S,p) ∈ D compute the pattern lt_V
3 if ft_V ≡ ft and lt_V ≡ lt
4    then if m is empty
5            then if there exists a descriptor l(S) ∈ D
6                    then return true
7            else if there exists a descriptor m(S) ∈ D
8                    then return true
```

We can prove the following.

THEOREM 7.7. *For an XP query q, a finite set of XP views $\mathcal{V}$ and their corresponding descriptors $\mathcal{D}$,* testEquiv*(q,$\mathcal{V}$) outputs* true *if and only if* testEquivDesc*(q, $\mathcal{D}$) does so.*

PROOF. The fragments of patterns in the set of **ft** and **lt** descriptors computed by testEquivDesc contains all the main branches and *relevant* predicates of first and last tokens of views. By relevant, we mean here those that appear in q (in ft and lt), since they are all required if $ft_{\mathcal{V}} \equiv ft$ and $lt_{\mathcal{V}} \equiv lt$. (The fact that query equivalence means isomorphism modulo minimization is important here.)

It follows immediately that testEquiv and testEquivDesc compute the same $ft_{\mathcal{V}}$ and $lt_{\mathcal{V}}$ patterns when these two equivalence tests hold. The following tests are trivially equivalent, by the definition of **l** and **m** descriptors. □

Example 7.8. For the query q_1 in Example 1.1, $m = vacation//trip/trip[guide]$, $ft = doc(T)$ and $lt = tour[schedule//walk]/museum$.

For the QSS $\mathcal{P}$ from Example 3.3 and its two expansions v_1 and v_2, v_1 can be represented by the descriptors $\mathbf{ft}(S, doc(T))$, $\mathbf{lt}(S, museum)$, $\mathbf{m}(S)$ too since v_1 has the form $pref_1//m//suff_1$, with $pref_1 = doc(T)$ and $suff_1 = museum$. Similarly, v_2 is represented by $\mathbf{ft}(S, doc(T))$ and $\mathbf{lt}(S, tour[schedule//walk]/museum)$.

Running on these descriptors, testEquivDesc will confirm that there exists an equivalent rewriting for q_1 using $\{v_1, v_2\}$.

7.3. View Descriptors from a QSS

We present in this section a bottom-up algorithm (findDescSupp) that runs on a QSS $\mathcal{P}$ and a multitoken query q, computing the view descriptors (w.r.t. q) for the expansions of $\mathcal{P}$. Our algorithm is sound and complete, running in polynomial time. Via Theorems 7.7 and 7.4, findDescSupp delivers a sound PTIME algorithm for support when the input queries are multitoken from *XP*. Moreover, via Theorems 7.7 and 7.5, it delivers a PTIME decision procedure for support when the input queries are multitoken from XP_{es}.

We will describe findDescSupp by separate subroutines, one for each of the four kinds of view descriptors (first-token descriptors in Section 7.3.1, last-token descriptors in Section 7.3.2, l-descriptors in Section 7.3.3 and m-descriptors in Section 7.3.4).

Since a QSS constructs views by putting together fragments, we construct our view descriptors via *fragment descriptors*, which represent equivalence classes for fragment expansions. Intuitively, fragment descriptors bookkeep in the bottom-up procedure certain partial details, on the expansions of fragment names, details that allow us to test incrementally the various conditions of testEquiv.

To better clarify our choices for fragment descriptors, let us first detail how the tests of testEquiv can be done in incremental manner.

Mapping and equivalence tests are naturally done bottom-up, one node at time, and this translates easily into procedures that run on the QSS and rely on fragment descriptors. We already presented in Section 4 how one can test in this way the existence of containment or equivalence with q or parts thereof. We will handle the tests of lines 3, 10, and 11 in testEquiv similarly, by descriptors which record mapping or equivalence details.

For line 6, the nonexistence of a containment mapping between linear paths needs a slightly different approach. One can test incrementally if a linear path l_1 is contained in a linear path l_2 as follows.

(1) Test if the last token of l_2 maps in the last token of l_1, such that OUT(l_1) is the image of OUT(l_2). Let k denote the start rank (the upmost node) of this mapping image.
(2) Bottom-up, for each intermediary token t of l_2, map t in the lowest possible[1] available (i.e. above k) part of l_1. If no such mapping exists, we can conclude the nonexistence of a containment mapping from l_2 in l_1. At each step, bookkeep as k the start rank of that image of t in l_1.
(3) Finally, if the previous set of steps did not yield a negative answer already, a containment mapping of l_2 in l_1 does not exist if and only if the first token of l_2 cannot be mapped in l_1 s.t. (i) ROOT(l_1) is the image of ROOT(l_2), and (ii) the image of this first token of l_2 is above the current rank k.

A similar incremental approach, advancing one token at a time, can be used for the tests in lines 12 and 13, as we are dealing again with linear patterns. More precisely, a bottom-up approach as above can be used in the case of MB($suffix_j$) and, symmetrically, a top-down one can be used in the case of MB($prefix_j$).

Note that the approach above advances one token at a time, and not one node at a time (which would have fitted nicely with how views are built in a QSS). This is because we need to check that all possible partial mappings fail sooner or later to go through

[1]As we handle one token at a time, choosing the lowest available mapping image preserves all opportunities to find containment.

to a full containment mapping (for line 6), root-mapping (for line 12), respectively output-mapping (for line 13). And the only way to ensure that no mapping opportunity is prematurely discarded is to settle on a mapping image in a descriptor, the lowest possible one, only when a token is complete (i.e., its incoming edge is //).

Example 7.9. Consider the following two patterns:

$$q_1 = doc(T)/vacation/trip//tour/tour//museum,$$
$$q_2 = doc(T)//vacation//tour//tour//museum.$$

We can test that $q_1 \sqsubset q_2$ by the procedure described above as follows: start by mapping q_2's last token, *museum* into q_1's last token. The start rank k of this mapping is $|q_1|$, that is, $k = 6$. Then, bottom-up (or right to left on the *XP* expression) we pass to q_2's token *tour*/*tour* and we map it in the lowest possible image above rank 6. This is at rank 4, which becomes the new value of k. Similarly, we map the other token *vacation* of q_2 and k becomes 2 and finally we map the first token of q_2, $doc(T)$ ar rank 1 in q_1.

Consider now the query

$$q_3 = doc(T)//activity//vacation//museum.$$

When testing whether $q_1 \sqsubset q_3$, the bottom-up procedure stops outputting *false* when the mapping of q_3's token *activity* is not possible above the rank 2, where its *vacation* token had a mapping image.

We are now ready to detail how the set R of view descriptors is computed in the algorithm findDescSupp.

Preliminaries. We start by assuming that all *equiv* or *map* descriptors are precomputed for boolean fragment names (as described in Section 4). In the same style, we compute *containment* and *equivalence descriptors* for *unary fragment names* (i.e., those with an output mark). More precisely, a descriptor $contain(f, n)$, for $n \in \text{MBN}(q)$, (resp. $equiv(f, n)$) denotes that some expansion v_f contains (resp. is equivalent to) the suffix of q rooted at the main branch node n. Other types of fragment descriptors will be introduced next.

In the following sections, when we refer to a mapping from some expansion v_f into a prefix or subquery p of q, we denote a mapping from v_f into q itself which takes the main branch nodes of v_f into main branch nodes of the p part. This means that predicate nodes of v_f may have images below the p part itself. Similarly, equivalence between an expansion v_f and a prefix or subquery p is defined based on a containment mapping from the view into the query as described. With these ajustements, we do not need to (i) test if QSS generated views and fragments thereof are in XP_{es}, and (ii) prune out unnecessary subpredicates violating the XP_{es} conditions (only the extended skeletons of views are useful in the rewriting). Indeed, some useful views may not be in this fragment, as explained in Lemma 7.1.

7.3.1. Computing first-token descriptors. For this part, we will use *prefix descriptors* for fragment names.

Definition 7.10. *Syntax.* For a unary fragment name f, a *prefix descriptor* is a tuple *pref(f, p, k)*, for k being a rank in the range 1 to $|\text{MB}(ft)|$ and p denoting any pattern that can be obtained from ft by keeping (a) a substring of the main branch, starting from rank k, and (b) eventually, one predicate on that substring.

Semantics. There exists an expansion v_f such that (a) v_f has a containment mapping in the subtree of q rooted at the ft node of rank k, and (b) v_f has a first token which is of the form p plus additional predicates, if any (they are ignored in the descriptor).

Part 1 **of** findDescSupp(q, $\mathcal{P}$)**.**

Iterate the following steps.

(1) For $f(X) \rightarrow l[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]//g(X)$,
add a prefix descriptor

$$\mathbf{pref}(\mathbf{f}, \mathbf{l}, \mathbf{k})$$

for each rank k, $1 \leq k \leq |\mathrm{MB}(ft)|$, such that $node_q(k)$ has label l, for which we can infer that v_f contains the pattern $q(k)$, by the following tests:
(a) for each fragment name c_i there exists a descriptor $map(c_i, n)$, for n being a /-child of $node_q(k)$,
(b) for each fragment name d_j there exists a descriptor $map(d_j, n)$, for n being a descendant of $node_q(k)$
(c) there exists a containment descriptor $contain(g, n)$ for n being any main branch node of rank $k' > k$ in q.
Add also the prefix descriptor

$$\mathbf{pref}(\mathbf{f}, \mathbf{l}[\mathbf{P}], \mathbf{k})$$

if for a /-predicate (respectively //-predicate) P on $node_q(k)$ we have a descriptor $equiv(c_i, root_P)$ (resp. $equiv(d_j, root_P)$).

(2) For $f(X) \rightarrow l[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]/g(X)$,
given a prefix descriptor $pref(g, p', k')$, add a prefix descriptor

$$\mathbf{pref}(\mathbf{f}, \mathbf{l}/\mathbf{p}', \mathbf{k}),$$

for $k = k' - 1$, if $node_q(k)$ has label l and we can infer that v_f contains $q(k)$ by:
(a) for each fragment name c_i there exists a descriptor $map(c_i, n)$, for n being a /-child of $node_q(k)$,
(b) for each fragment name d_j there exists a descriptor $map(d_j, n)$, for n being a descendant of $node_q(k)$
Add also the prefix descriptor

$$\mathbf{pref}(\mathbf{f}, \mathbf{l}[\mathbf{P}]/\mathrm{MB}(p'), \mathbf{k}),$$

if for a /-predicate (respectively //-predicate) P on $node_q(k)$ we have a descriptor $equiv(c_i, root_P)$ (respectively $equiv(d_j, root_P)$),

(3) When a descriptor *pref(f,p,1)* is obtained, for $f = S$, add to R the view descriptor

$$\mathbf{ft}(\mathbf{S}, \mathbf{p}).$$

Example 7.11. On our running example, that is, query q_1 from Example 1.1 and the normalized QSS $\mathcal{P}$ from Example 3.4, we obtain first-token descriptors as follows:
On the rule $f_0(X) \rightarrow doc(T)//f_1(X)$,

(1) for q's main branch node at rank $k = 1$,
(2) we already have a fragment descriptor $contain(f_1, n_{v_3})$,
(3) we have no c_i, d_j branches,
(4) hence we can infer a fragment descriptor $pref(f_0, doc(T), 1)$.

Then, by Step (1.3) of findDescSupp we get the view descriptor $\mathbf{ft}(\mathbf{doc}(\mathbf{T}), \mathbf{f_0})$.

7.3.2. Computing last-token descriptors. We use for this part two kinds of fragment descriptors: *suffix descriptors* and *full-suffix descriptors*.

Definition 7.12. *Syntax.* For a unary fragment name f, a *suffix descriptor* is a tuple *suff(f, p)*, for p denoting any pattern that can be obtained from lt by keeping (a) a suffix of its main branch, and (b) eventually, one predicate on that suffix.

Semantics. This descriptor says that (a) v_f is a single-token query, of the form p plus maybe other predicates (ignored by the descriptor), and (b) v_f contains the subtree of lt rooted at the main branch node of rank $|\mathrm{MB}(lt)| - |\mathrm{MB}(p)| + 1$.

Definition 7.13. *Syntax.* For a unary fragment name f, a *full-suffix descriptor* is a tuple *fsuff(f, p, k)*, for k denoting a rank in q, and p being a pattern as defined in Definition 7.12 above.

Semantics. There exists an expansion v_f s.t. (a) v_f has a last token of the form p plus other predicates (if any), and (b) v_f maps in the subtree of q rooted at the main branch node of rank k.

Part 2 of findDescSupp*(q, $\mathcal{P}$)*

Main idea. We compute *suffix descriptors* similarly to the prefix ones. From them, *full-suffix descriptors* are then computed bottom-up, by simple containment mapping checks. If a descriptor *fsuff(f, p, 1)* is obtained, for $f = S$, we add $\mathbf{lt}(\mathbf{S}, \mathbf{p})$ to the set of view descriptors.

A. We compute *suffix descriptors* by iterating the following steps:

(1) For $f(X) \rightarrow l(X)[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]$,
add a suffix descriptor

$$\mathbf{suff}(\mathbf{f}, \mathbf{l})$$

if we can infer that v_f contains the subtree of lt rooted at OUT(lt) (this containment test is done similarly to Step (1.1)).
Add also the suffix descriptor

$$\mathbf{suff}(\mathbf{f}, \mathbf{l}[\mathbf{P}])$$

if for a /-predicate (resp. //-predicate) P on OUT(lt) we have a descriptor $equiv(c_i, root_P)$ (resp. $equiv(d_j, root_P)$),

(2) For $f(X) \rightarrow l[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]/g(X)$,
given a descriptor $suff(g, p')$, add a suffix descriptor

$$\mathbf{suff}(\mathbf{f}, \mathbf{l}/\mathbf{p}')$$

if, for $k = |\mathrm{MB}(q)| - |p'|$, we can infer that v_f contains the pattern $lt(k)$.
Add also the suffix descriptor

$$\mathbf{suff}(\mathbf{f}, \mathbf{l}[\mathbf{P}]/\mathrm{MB}(p'))$$

if for a /-predicate (resp. //-predicate) P on $node_{lt}(k)$ we have a descriptor $equiv(c_i, root_P)$ (resp. $equiv(d_j, root_P)$).

B. Compute *full-suffix descriptors* by iterating the following:

(1) For $f(X) \rightarrow l[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]//g(X)$,
given a suffix descriptor $suff(g, p)$, add a full suffix descriptor

$$\mathbf{fsuff}(\mathbf{f}, \mathbf{p}, \mathbf{k})$$

for each rank $k < |\mathrm{MB}(q)| - |\mathrm{MB}(p)|$ s.t. we can infer that v_f contains the pattern $q(k)$.

(2) For $f(X) \rightarrow l[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]/g(X)$,
given a full-suffix descriptor $fsuff(g, p, k')$, add a full-suffix descriptor

$$\mathbf{fsuff}(\mathbf{f}, \mathbf{p}, \mathbf{k}),$$

for $k = k' - 1$, if we can infer that v_f contains the pattern $q(k)$.

(3) For $f(X) \rightarrow l[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]//g(X)$,
given a full-suffix descriptor $fsuff(g, p, k')$, add a full-suffix descriptor

$$\mathbf{fsuff}(\mathbf{f}, \mathbf{p}, \mathbf{k})$$

for each rank $k < k'$ s.t. we can infer that v_f contains the pattern $q(k)$.

(4) When a descriptor $fsuff(f, p, 1)$ is obtained, for $f = S$, add to R the view descriptor

$$\mathbf{lt}(\mathbf{S}, \mathbf{p}).$$

Example 7.14. On our running example, we obtain last-token descriptors as follows.

(1) We compute *suffix* descriptors as follows:
(a) on the rule $f_5(X) \rightarrow museum(X)$,

(b) by the step 2.A.1 of findDescSupp we get the fragment descriptor

$$d_1 = suff(f_5, museum),$$

(2) We also have the mapping descriptors $d_2 = equiv(f_7, n_{g_2})$, $d_2' = map(f_4, n_{s_2})$, $d_2'' = equiv(f_4, n_{s_2})$

(3) From d_1 and d_2, for the rule $f_2(X) \rightarrow trip[f_7()]//f(5)(X)$ we get the full-suffix descriptor $d_3 = fsuff(f_2, museum, 4)$, as the rank of node n_{tr_2} is 4.

From d_3 and the rule $f_1(X) \rightarrow vacation//f_2(X)$ we then get

$$d_4 = fsuff(f_1, museum, 2).$$

Finally, from d_4 and the rule $f_0(X) \rightarrow doc(T)//f_1(X)$ we then get

$$d_5 = fsuff(f_0, museum, 1),$$

which leads to the view descriptor $\mathbf{lt(f_0, museum)}$.

(4) On another thread of computation, for the rule $f_3(X) \rightarrow tour[f_4]/f_5(X)$, using d_1 and d_2', by the step 2.A.2 of findDescSupp we get:
(a) the suffix descriptor $d_6 = suff(f_3, tour/museum)$,
(b) since we have d_2'', for the predicate $P = [schedule//walk]$, the suffix descriptor

$$d_6' = suff(f_3, tour[schedule//walk]/museum)$$

Then, from d_6' and the rule $f_2(X) \rightarrow tour//f_4(X)$ we get the full-suffix descriptor

$$d_7 = fsuff(f_2, tour[schedule//walk]/museum, 4).$$

From d_7 and the rule $f_1(X) \rightarrow vacation//f_2(X)$ we then get

$$d_8 = fsuff(f_1, trip[schedule//walk]/museum, 2).$$

Finally, from d_8 and the rule $f_0(X) \rightarrow doc(T)//f_1(X)$ we then get

$$d_9 = fsuff(f_0, trip[schedule//walk]/museum, 1),$$

which leads to the view descriptor $\mathbf{lt(f_0, trip[schedule//walk]/museum)}$.

7.3.3. Computing l-descriptors. We have seen in Section 7.3 an incremental procedure that tests the non-existence of a containment mapping for linear patterns bottom-up, one token at a time. To run a similar test directly on the QSS (whose expansions are revealed one node at a time), we need additional bookkeeping, allowing us to choose mapping images one token at a time. For this, we record at each step in the bottom-up process the following: (i) the current first token of v_f, (ii) the *lowest possible* mapping image for the rest of v_f (except its first token). This allows us to settle on the lowest possible mapping (in a descriptor) only when the token is complete (we have its incoming edge and it is a //-edge). To this end, we use *partial l-descriptors*.

Definition 7.15. *Syntax.* For a unary fragment name f, a *partial l-descriptor* is a tuple $pl[f, k_1, (k_2, p)]$, where k_1 is a rank in q, k_2 is a rank in l_q and p is any substring of l_q.

Semantics. There exists an expansion v_f s.t. (a) v_f contains the subtree of q rooted at the main branch node of rank k_1, (b) the main branch of the first token of v_f is p, and (c) k_2 is the start (upmost rank) of the *lowest possible output-mapping image* of the rest of the main branch of v_f (i.e., except the first token, represented by p) into l_q. By convention, this rank is $|l_q| + 1$ when v_f has only one token (the one described by p) and is 0 when there is no such mapping.

Part 3 **of** findDescSupp(q, $\mathcal{P}$)

Iterate the following steps:

(1) For rules $f(X) \rightarrow l(X)[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]$,
Add a partial l-descriptor

$$\mathbf{pl}[\mathbf{f}, |\mathrm{MB}(q)|, (|\mathbf{l_q}| + \mathbf{1}, \mathbf{l})]$$

if we can infer that v_f contains the subtree of q rooted at OUT(q),
(2) For $f(X) \rightarrow l[\ldots]/g(X)$,
given a descriptor $pl[g, k'_1, (k'_2, p')]$, if we can infer that v_f contains $q(k'_1 - 1)$:
(a) Add the partial l-descriptor

$$\mathbf{pl}[\mathbf{f}, \mathbf{k'_1} - \mathbf{1}, (\mathbf{k'_2}, \mathbf{l/p'})]$$

if f is not the start fragment name.
(b) Otherwise, add to R the view descriptor

$$\mathbf{l}(\mathbf{S})$$

if there is no mapping of l/p' into l_q whose image starts at ROOT(l_q) and ends above k'_2.
(3) For $f(X) \rightarrow l[\ldots]//g(X)$,
given a descriptor $pl[g, k'_1, (k'_2, p')]$, for each rank k_1, $1 \leq k_1 < k'_1$ s.t. v_f contains $q(k_1)$:
(a) if f is not the start fragment name, find the lowest rank k_2, s.t. p' has a mapping into l_q whose image starts at k_2 and ends *above* k'_2, where if $k'_2 = |l_q| + 1$ above means at $k'_2 - 1$; if no such value exists, set k_2 to 0. Output the partial l-descriptor

$$\mathbf{pl}[\mathbf{f}, \mathbf{k_1}, (\mathbf{k_2}, \mathbf{l})].$$

(b) otherwise, add to R the view descriptor

$$\mathbf{l}(\mathbf{S})$$

if there is no mapping of $l//p'$ into l_q whose image starts at ROOT(l_q) and ends above k'_2.

7.3.4. Computing m-descriptors. For this part, we need to check that some view v_j can be seen as being of the form $prefix_j//m'//suffix_j$, s. t. $m' \equiv m$ and

(1) $prefix_j$ root-maps into ft but MB($prefix_j$) cannot root-map into MB($ft)_m$,
(2) $suffix_j$ output-maps into lt, but MB($suffix_j$) cannot output-map into MB($lt)_m$.

Each of these aspects of an expansion is captured by a different type of fragment descriptor. We will output a view descriptor $\mathbf{m}(\mathbf{S})$ when a rule $f(X) \rightarrow l[\ldots]//g(X)$ is available and when (via fragment descriptors) we have that:

(1) g has an expansion v_g that gives us the part $m'//suffix_j$,
(2) there exist views generated via that rule and v_g, s.t. the part above v_g (in other words, the view obtained by expanding g in the empty pattern) has the properties for $prefix_j$.

We can use separate subroutines for each of these two items, and then the overall step above will combine their individual results.

For the $suffix_j$ part, we use *below m-descriptors*:

Definition 7.16. *Syntax.* For a unary fragment name f, a *below m-descriptors* is a tuple $bm[f, k_1, (k_2, p)]$, where k_1 and k_2 denote ranks in q, and p denotes any substring of MB(q).

Semantics. There exists an expansion v_f s.t. (a) v_f contains the subtree of ft rooted at the node of rank k_1, (b) p is the main branch of the first token of v_f, and (c) k_2 is the start of the *lowest possible output-mapping image* of the main branch of the rest of v_f (besides p) into MB($lt)_m$; by convention, k_2 is $|\mathrm{MB}(q)| + 1$ when v_f has only one token and is 0 when there is no such mapping.

Then, for the m part, we use *partial m-descriptors*:

Definition 7.17. *Syntax.* For a unary fragment name f, a *partial m-descriptor* is a tuple $pm(f, k)$, where k is a number in the range 1 to $|\text{MB}(m)|$, indicating a suffix of m.

Semantics. This descriptor says that (a) v_f is of the form $m'//suffix_j$, s.t. m' is equivalent with m's suffix having k main branch nodes, and (b) $suffix_j$ has the properties already described.

For the $prefix_j$ part, we use *above m-descriptors*:

Definition 7.18. *Syntax.* For a unary fragment name f, an *above m-descriptor* is a tuple $am[f, k_1, (k_2, p)]$, where k_1, k_2 denote ranks in q and p is any substring of $\text{MB}(q)$.

Semantics when p is empty (denoted hereafter '–'). there exists an expansion v of the QSS s.t. (a) v is of the form $rest//v_f$, for v_f being an expansion of f (b) *rest* root-maps into ft such that its image ends at the rank k_1, and (c) the end (bottommost node) of the highest possible root-mapping image of $\text{MB}(rest)$ into $\text{MB}(ft)_m$ is k_2; if no such mapping exists, by convention k_2 is $|\text{MB}(ft)_m| + 1$.

Semantics when $p \neq$ '–'. there exists an expansion v of the QSS s.t. (a) v is of the form $rest//p'/v_f$, for $p = \text{MB}(p')$, (b) $rest//p'$ root-maps into ft such that the image of p' ends at the rank k_1, and (c) the end (bottommost node) of the highest possible root-mapping image of $\text{MB}(rest)$ into $\text{MB}(ft)_m$ is k_2; by convention, if no such mapping exists, k_2 is $|\text{MB}(ft)_m| + 1$; when *rest* is empty k_2 is 0.

Given a rule $f(X) \rightarrow l[\ldots]/g(X)$ or $f(X) \rightarrow l[\ldots]//g(X)$, we will use an am-descriptor for f to infer one for g.

Main idea. Below m-descriptors are computed by a similar approach (one token at time) as the one used for partial l-descriptors. The *above m-descriptors* are obtained similarly, but in top-down manner. Starting from below-m descriptors, the *partial m-descriptors* are computed bottom-up, by simple equivalence checks. If for some fragment name g we computed both an above m-descriptor $am[g, k_1, (|\text{MB}(ft)_m| + 1, -)]$ and a partial m-descriptor $pm(g, |\text{MB}(m)|)$, we can add a descriptor $\mathbf{m}(\mathbf{S})$ to the set of view descriptors.

Part 4 **of** findDescSupp(q, $\mathcal{P}$)

Apply the the following steps:

A. Compute *below m-descriptors* by iterating the following steps:

(1) For rules $f(X) \rightarrow l(X)[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]$,
Add a below m-descriptor

$$\mathbf{bm}[\mathbf{f}, |\text{MB}(q)|, (|\text{MB}(q)| + \mathbf{1}, \mathbf{l})]$$

if we can infer that v_f contains the subtree of q rooted at $\text{OUT}(q)$.

(2) For $f(X) \rightarrow l[\ldots]/g(X)$,
given a descriptor $bm[g, k_1', (k_2', p')]$, if we can infer that v_f contains the pattern $lt(k_1' - 1)$, add the below m-descriptor

$$\mathbf{bm}[\mathbf{f}, \mathbf{k_1' - 1}, (\mathbf{k_2'}, \mathbf{l/p'})].$$

(3) For $f(X) \rightarrow l[\ldots]//g(X)$,
given a descriptor $bm[g, k_1', (k_2', p')]$, for each rank k_1, $k_1 < k_1'$, s.t. we can infer that v_f contains the pattern $lt(k_1)$,

find the lowest possible rank k_2 in $\text{MB}(lt)_m$ s.t. the token p' has a mapping into $\text{MB}(lt)_m$ starting at k_2 and ending *above* k_2' (if $k_2' = |\text{MB}(q)| + 1$ above means at $k_2' - 1$). If no such rank is found, set k_2 to 0.
Add a below m-descriptor

$$\mathbf{bm}[\mathbf{f}, \mathbf{k_1}, (\mathbf{k_2}, \mathbf{l})].$$

B. Compute *partial m-descriptors* by iterating the following steps:

(1) For $f(X) \rightarrow l[\dots]//g(X)$,
given a below m-descriptor $bm[g, k'_1, (k'_2, p')]$ s.t. (i) k'_2 is already 0, or (ii) p' cannot be mapped anywhere above k'_2 in $\text{MB}(lt)_m$, add the partial m-descriptor

$$\mathbf{pm}(\mathbf{f}, \mathbf{1})$$

if we can infer that v'_f is equivalent with m's suffix of size 1.

(2) For either $f(X) \rightarrow l[\dots]/g(X)$ or $f(X) \rightarrow l[\dots]//g(X)$,
given a partial m-descriptor $pm(g, n')$, add a partial m-descriptor

$$\mathbf{pm}(\mathbf{f}, \mathbf{n})$$

if we can infer that query $cut(v_f, n' + 1)$ is equivalent with m's suffix of size $n = n' + 1$.

C. Compute *above m-descriptors* by iterating the following steps:

(1) From start fragment names f and either $f(X) \rightarrow l[\dots]/g(X)$ or $f(X) \rightarrow l[\dots]//g(X)$,
Add an above m-descriptor

$$\mathbf{am}[\mathbf{g}, \mathbf{1}, (\mathbf{0}, \mathbf{l})] \ \textit{respectively} \ \mathbf{am}[\mathbf{g}, \mathbf{1}, (\mathbf{1}, -)]$$

if we can infer that v'_f root-maps into ft.

(2) For $f(X) \rightarrow l[\dots]/g(X)$,
given a descriptor $am[f, k'_1, (k'_2, p')]$:
(a) *if p' is not '-':* add an above m-descriptor

$$\mathbf{am}[\mathbf{g}, \mathbf{k'_1 + 1}, (\mathbf{k'_2}, \mathbf{p'/l})]$$

if we can infer that v'_f root-maps in $ft(k'_1 + 1)$.
(b) *if p' is '-':* add an above m-descriptor

$$\mathbf{am}[\mathbf{g}, \mathbf{k_1}, (\mathbf{k'_2}, \mathbf{l})]$$

for each rank k_1, $k_1 > k'_1$, s.t. we can infer that v'_f root-maps in the pattern $ft(k_1)$.

(3) For $f(X) \rightarrow l[\dots]//g(X)$,
given a descriptor $am[f, k'_1, (k'_2, p')]$:
(a) *if p' is not '-':* add an above m-descriptor

$$\mathbf{am}[\mathbf{g}, \mathbf{k'_1 + 1}, (\mathbf{k_2}, -)]$$

if we can infer that v'_f root-maps in the pattern $ft(k'_1 + 1)$, for the highest rank k_2, $k'_2 < k_2 \leq |\text{MB}(ft)_m|$, s.t. p'/l has a mapping into $\text{MB}(ft)_m$ starting *below* k'_2, where if $k'_2 = 0$ below means rank 1, and ending at k_2; if no such mapping exists set k_2 to $|\text{MB}(ft)_m| + 1$.
(b) *if p' is '-':* add an above m-descriptor

$$\mathbf{am}[\mathbf{g}, \mathbf{k_1}, (\mathbf{k_2}, -)]$$

for each rank k_1, $k_1 > k'_1$, s.t. v'_f root-maps in the pattern $ft(k_1)$, for the highest rank k_2, $k'_2 < k_2 \leq |\text{MB}(ft)_m|$, s.t. the token l has a mapping into $\text{MB}(ft)_m$ at rank k_2; if no such mapping exists set k_2 to $|\text{MB}(ft)_m| + 1$.

D. Finally, for a fragment name g, add to R the view descriptor $\mathbf{m}(\mathbf{S})$ if we have both
(a) a partial m-descriptor $pm(g, |\text{MB}(m)|)$
(b) an above m-descriptor $am[g, k_1, (|\text{MB}(ft)_m| + 1, -)]$.

Example 7.19. On our running example, we obtain m-descriptors as follows.

(1) We first obtain the below m-descriptor $bm(f_5, 6, (7, museum))$, as $|\text{MB}(q)| = 6$.
(2) We then get to partial m-descriptors:
From the rule $f_2(X) \rightarrow trip[f_7()]/f_5(X)$, since $\text{MB}(lt)_m$ is the empty pattern, from $d_2 = equiv(f_7, n_{g_2})$ we get the fragment descriptor $pm(f_2, 1)$ (as the size of the main branch of m's prefix $trip[guide]$ is 1).
Then, from the rule $f_2(X) \rightarrow trip/f_2(X)$ and from $pm(f_2, 1)$ we get another partial m-descriptor, $pm(f_2, 2)$.

Finally, from the rule $f_1(X) \rightarrow vacation//f_2(X)$ and from $pm(f_2, 2)$ we get the partial m-descriptor $pm(f_1, 3)$. Note that $|\text{MB}(m)| = 3$.

(3) We compute above m-descriptors as follows:
Starting with rule $f_0 \rightarrow doc(T)//f_1(X)$, since v'_{f_0} is $doc(T)$, we get the above m-descriptor $am(f_1, 1, (1, -))$. Note that since $\text{MB}(ft)_m$ is empty, $|\text{MB}(ft)_m| + 1 = 1$.

(4) Using descriptors $pm(f_1, 3)$, $am(f_1, 1, (1, -))$, we obtain the view descriptor $\mathbf{m}(\mathbf{f_0})$.

7.4. Formal Results

We can now prove the following.

THEOREM 7.20. *Given a QSS $\mathcal{P}$ and a multitoken query q, algorithm* findDescSupp *is sound and complete for computing the descriptors for $\mathcal{P}$'s expansions.* findDescSupp *runs in polynomial time in the size of the query and of the QSS.*

PROOF. We start by considering first-token descriptors. Step 1 of findDescSupp computes, bottom up, all suffixes of a prefix plus, eventually, one predicate. It stores them in the second field of prefix descriptors. When the position $k = 1$ is reached, it means that a main branch of a first token, plus maybe a predicate, has been computed. This justifies the inference of an first-token view descriptor (ft) that has the same p pattern as the $pref(f, p, 1)$ descriptor. We infer all such descriptors because we explore bottom-up all paths that could represent the first-token in an expansion of the QSS.

The case for last-token descriptors is symmetrical to the one of first-token descriptors.

Partial l-descriptors are computed starting from the base case of rules that have no tree fragment names (Step 1), then recording in bottom-up manner patterns and ranks in q and l_q, inferring partial l-descriptors that satisfy Definition 7.15. The procedure infers all such descriptors because, intuitively, a partial l-descriptor for smaller ranks in q and l_q exists only if there are partial l-descriptors for higher ranks, that is, corresponding to "lower" fragments of the main branches of the tree patterns. And, by the same reasoning, if no mapping can be inferred while going "up" in the pattern (case in which an **l** descriptor is inferred), it guarantees the non-existence of a containment mapping.

Below m-descriptors are computed in a very similar way to partial l-descriptors. And, using similar arguments, it can be shown that the algorithm computes all below m-descriptors. The computation of partial m-descriptors follows exactly the conditions in their definition. Above m-descriptors are also computed as a bottom-up evaluation, checking the conditions from Definition 7.18. Finally, having a partial m-descriptor $pm(g, |\text{MB}(m)|)$ and an above m-descriptor $am[g, k_1, (|\text{MB}(ft)_m| + 1, -)]$ justifies the introduction of an **m** descriptor, as it guarantees that some views generated by the QSS satisfy the conditions from lines 9–13 in Algorithm testEquiv.

It can easily be verified that the number of descriptors is polynomial, as they are defined using positions, subpatterns or patterns constructed in PTIME from the query and the specification. The computation of each descriptor is PTIME, as it amounts to simple tests on polynomial size patterns. Hence the computation of all descriptors is done in PTIME. □

Observation. By Theorems 7.20, 7.7 and 7.4, for a multi-token *XP* query q and QSS $\mathcal{P}$, given the descriptor set $\mathcal{D} :=$ findDescSupp$(q, \mathcal{P})$, q is supported by $\mathcal{P}$ if testEquivDesc$(q, \mathcal{D})$ outputs true.

Moreover, by Theorem 7.5, if q is in XP_{es}, it is supported by $\mathcal{P}$ (considering for now only rewrite plans that intersect views) if and only if testEquivDesc$(q, \mathcal{D})$ outputs true. We generalize these two observations to support in $XP^{\cap}$ in the next section.

7.5. Support with Compensated Views

We consider in this section general $XP^{\cap}$ rewrite plans for support that, before performing the intersection step, might compensate (some of) the views.

We show that support in this new setting can be reduced to support by rewrite plans which only intersect expansions of a QSS. This allows us to reuse the PTIME algorithms given in Section 7 (testEquivDesc and findDescSupp) and to find strictly more rewritings, namely those that would not be feasible without compensation. Thus we obtain a sound algorithm for support on XP multitoken queries in the rewrite language $XP^{\cap}$. This algorithm becomes complete when the input query is from XP_{es}.

Our reduction relies on the same QSS transformation, $comp(\mathcal{P}, q)$, used in Section 5.1.

Example 7.21. Suppose that the QSS in Example 3.3 is modified to return the guided trips themselves instead of their museums, by changing the third rule into rule R_3:

$$(R_3): \quad f_1(X) \rightarrow trip(X)[guide].$$

and obtaining a new QSS $\mathcal{P}_2$. Then, one of the expansions of $\mathcal{P}_2$ is:

$$v_3 : doc(T)//vacation//trip/trip[guide].$$

A query plan that rewrites q_2 using compensated views is

$$doc(v_3)/v_3/trip/museum \cap doc(v_2)/v_2/museum.$$

We can infer this rewriting by compensating R_3 with a navigation to a *museum* child, which leads to a QSS identical to $\mathcal{P}$.

We can prove the following.

THEOREM 7.22. *Given a QSS $\mathcal{P}$ and a multitoken XP query q, let* $\mathcal{D} :=$ findDescSupp$(q, comp(\mathcal{P}, q))$.

(1) *Algorithm* testEquivDesc*$(q,\mathcal{D})$ is sound for support in $XP^{\cap}$, that is, q is supported by $\mathcal{P}$ in $XP^{\cap}$ if* testEquivDesc*$(q,\mathcal{D})$ outputs true.*
(2) testEquivDesc*$(q,\mathcal{D})$ is also complete if q belongs to XP_{es}, that is, q is supported by $\mathcal{P}$ in $XP^{\cap}$ iff* testEquivDesc*$(q,\mathcal{D})$ outputs true.*

PROOF. Soundness follows from the observation given after Theorem 7.20 and from the fact that $comp(\mathcal{P}, q)$ generates compensated views. Completeness follows from Theorem 7.5 and the fact that $comp(\mathcal{P}, q)$ generates views augmented by any compensation that may be used by a mapping from q into an interleaving of views (i.e., suffixes of q). □

Summarizing these results, the following algorithm, denoted testSupp, can be used as a decision procedure for support in $XP^{\cap}$ on multitoken input queries.

ALGORITHM: testSupp$(\mathcal{P}, q)$

1. **let** $\mathcal{P}' = comp(\mathcal{P}, q)$
2. **let** $\mathcal{D} :=$ findDescSupp$(q, \mathcal{P}')$
3. **if** testEquivDesc$(q, \mathcal{D})$
4. **then return** *true*

7.6. Finding the Actual Views

We have so far only provided algorithms for deciding support and expressibility. To turn them into algorithms exhibiting the actual views generated by a QSS as well as the

rewriting using them requires extra bookkeeping. All we need to do is to carry along with a view or fragment descriptor one *witness*, that is, one actual expansion built during its derivation. Importantly, these witnesses will not be an additional dimension for the space of equivalence classes (i.e., two descriptors with same components but different witnesses will be considered identical and either one can be safely discarded). For instance, a prefix descriptor will now be of the form $pref(f, p, k, v)$ where v is some expansion of f (i.e., a view fragment) in the equivalence class described by p and k.

At each bottom-up step of findDescSupp, for each newly inferred descriptor d, its witness can be built from the witnesses of the existing descriptors that yielded d. For example, at Step (1.2) of findDescSupp, we would obtain a descriptor

$$pref(f, l/p', k, l[v_{c_1}, \ldots, v_{c_n}, v_{d_1}, \ldots, v_{d_m}]/v_g),$$

where each v_{c_i} (resp. v_{d_j}) denotes the witness associated with the mapping descriptor for c_i (resp. d_j) that was used in step (a) (resp. step (b)) and v_g denotes the expansion witnessing the prefix descriptor for g (with components p' and $k+1$).

So in the end view descriptors will also give us the necessary views and the equivalent rewrite plan will be their intersection. If compensated views are allowed, one needs to have on one hand the view definition and on the other the additional compensation. For that, we can simply mark in the $comp(\mathcal{P}, q)$ rules where compensation starts, for instance by using special names for the tree fragment names that were added to the original program $\mathcal{P}$. We can then keep both the view definitions and their compensations as witnesses.

8. SINGLE-TOKEN QUERIES

We consider in this section the remaining subfragment of XP_{es}, namely single-token queries. We show that id-based support becomes NP-hard (Theorem 8.1). Contrast this with both id-support for queries that have at least one //-edge in the main branch, and the rewriting problem for single-token XP_{es} queries under an explicit set of views, for which PTIME decision procedures exist.

THEOREM 8.1. *For an XP_{es} single-token query q and a QSS $\mathcal{P}$, deciding if q is supported by $\mathcal{P}$ in $XP^{\cap}$ is NP-hard.*

PROOF. We use a reduction from the minimum set-cover problem [Garey and Johnson 1979]. Let $(\mathcal{U}, \mathcal{S}, k)$ be an instance of this problem, with $\mathcal{U} = \{e_1, \ldots, e_n\}$ denoting the universe, $\mathcal{S} = \{S_1, \ldots S_m\}$ denoting the sets s.t. $S_i \subset \mathcal{U}$ for each S_i. We want to know whether there exists a subset S' of S, of size at most k, that can cover $\mathcal{U}$ (i.e., each element of $\mathcal{U}$ belongs to at least one set of S').

The reduction takes as input the set $\mathcal{U}$ and $\mathcal{S}$ (size $|\mathcal{S}| \times |\mathcal{S}|$) and the value k (size $lg(k)$). Let p be the biggest exponent s.t. $2^p \leq k$ and let $b_p b_{p-1} \ldots b_0$ be the binary representation of k. We build an instance of the support problem, with the QSS defined as follows.

(1) the tree fragments names are
$F = \{S, set, f, g, f_s, f_p^p, f_p^{p-1}, \ldots, f_p^0, f_{p-1}^{p-1}, f_{p-1}^{p-2}, \ldots, f_{p-1}^0, \ldots, f_1^1, f_1^0, f_0^0\}$.
(2) the alphabet $\Sigma = \{root, out, a, b, e_1, \ldots, e_m\}$.
(3) S is the start fragment.
(4) the set of productions P defined as follows:
   (a) the start productions
   $$S(X) \rightarrow doc(T)/f(X) \text{ and } S(X) \rightarrow doc(T)//g(X)$$
   (b) $f(X) \rightarrow a/a/a[Q]/a/out(X)$
   Where Q denotes the pattern obtained as follows: $Q \rightarrow b[h_p^p, h_{p-1}^{p-1} \ldots, h_1^1, h_0^0]$

(c) $\forall i = \overline{0, p}$ s.t. $b_i = 1$, we have the productions
$$h_i^k \rightarrow b\big[h_i^{k-1}\big]\big[h_i^{k-1}\big] \forall k = \overline{0, i}, \quad h_i^0 \rightarrow b[all]$$
(d) $all \rightarrow b[e_n, \ldots, e_n]$
(e) $g(X) \rightarrow a[b[//e_1, \ldots, //e_n]]/a[\,f_s]/a//out(X)$
(f) $f_s \rightarrow b[\,f_p^p,\, f_{p-1}^{p-1} \ldots,\, f_1^1,\, f_0^0]$
(g) $\forall i = \overline{0, p}$ s.t. $b_i = 1$, we have the productions
$$f_i^k \rightarrow b\big[\,f_i^{k-1}\big]\big[\,f_i^{k-1}\big] \forall k = \overline{0, i}, \quad f_i^0 \rightarrow b[set]$$
(h) $\forall i = \overline{0, p}$ s.t. $b_i = 0$, we have the production $f_i^i \rightarrow ()$
(i) $\forall S_j = \{e_{l_1}, \ldots, e_{l_j}\} \in S$ we have the production $set \rightarrow b[e_{l_1}, \ldots, e_{l_j}]$
(j) finally we have the production $set \rightarrow ()$.

This QSS produces two families of views. The first one, obtained via the tree fragment name f, contains only one member (v_f), which has a main branch $doc(T)/a/a/a/a/out$, and a Q predicate on the third a-node. Q has the set $e_1, \ldots e_n$ k times in total.

The views from the second family, obtained via the tree fragment name g, have the main branch $doc(T)//a/a/a//out$, the predicate $b[//e_1, //e_2, \ldots, //e_n]$ on the first a-node and a predicate produced by f_s on the second a-node. f_s produces branches of length at most $p + 2$ followed by one of the given subsets of $\{e_1, \ldots, e_n\}$.

Now let q be the single-token XP_{es} query
$$q = doc(T)/a/a[b[//e_1, //e_2, \ldots, //e_n]]/a[Q]/a/out.$$

It is easy to see that all the views generated by the QSS contain q: v_f maps obviously in q and all the other views have a containment mapping into q since, by construction, any pattern produced by f_s can be mapped into Q.

We now consider if q is supported, which amounts here to testing if the intersection of all the views is contained in q. Note that q will contain any interleaving which, for at least one view v_g, collapses the third a-node of v_g with the fourth a-node of v_f. This is because such an interleaving would be of the form
$$doc(T)/a/a[b[//e_1, //e_2, \ldots, //e_n]][\ldots]/a[Q][Q'][\ldots]/a/out$$
where Q' is produced by f_s and is actually redundant (can be minimized away).

So the only interleavings that remain to consider are those in which all the third a-nodes of v_g views are collapsed with the third a-node of v_f. These interleavings are of the form
$$doc(T)/a[b[//e_1, //e_2, \ldots, //e_n]]/a[Q'][\ldots]/a[Q]/a/out.$$

We can see now that q contains these interleavings if and only if among the predicates Q' produced by f_s there exists one into which the pattern $b[//e_1, //e_2, \ldots, //e_n]$ can map. But this is possible if and only if all the elements $e_1, \ldots, e_n$ are present in Q', and this happens if and only if there exists a cover of maximal size k for the set $\mathcal{U}$. □

Discussion. The surprising dichotomy between support for single-token and multitoken extended skeletons is rooted in their differences on the respective tests for equivalence with an intersection of views. First, for the single-token case, it is easy to see that support can hold only if some view's main branch is equivalent to q's /-edges only main branch. Otherwise, one could easily exhibit interleavings that do have //-edges in their main branch, hence cannot be contained in q. With this, building interleavings amounts basically to deciding where to collapse main branch nodes from the various views on a linear path with /-edges only. Intuitively, it is now less a matter of how to order main branch nodes of the views, and more of choosing for each node a

coalescing option among the few available. By consequence, a candidate interleaving i (i.e., one that is equivalent to q and contains all other interleavings) might combine (put under the same main branch node) predicates contributed by different views *at all levels of the main branch*. When q has several tokens, this is true only for the candidate's first and last tokens (built by combining in the unique possible way the first and last tokens of the views), while the section in between has to be entirely present (isomorphic modulo minimization) in some view.

In the electronic appendix of this paper we describe an exponential-time decision procedure for support for single-token XP_{es} queries. In short, it is based on descriptors that keep track of view "contributions"—namely predicates—in interleavings, and then checking if these contributions are enough for equivalence. For the sake of brevity, and since single-token queries are of less interest in practice—moreover, support is intractable in this fragment—a sketch of this algorithm and further details are relegated to the appendix.

9. QSS WITH PARAMETERS

We consider now an extension to QSS with input parameters for text values (denoted $QSS^{\#}$) and correspondingly, an extension of XP to text conditions. We modify the grammar of XP as follows:

$$pred \ ::= \ \epsilon \mid [rpath] \mid [rpath = C] \mid [.//rpath] \mid [.//rpath = C] \mid pred \ pred,$$

where C terminals stand for text constants. Every node in an XML tree t is now assumed to have a text value *text(t)*, possibly empty. The duality with tree patterns is maintained by associating to every predicate node n in a pattern p a test of equality *test(n)*, that is either the empty word or a constant C. The notions of embedding, mapping and containment can be adapted in straightforward manner to take into account text equality conditions.

The definition of $QSS^{\#}$ can be obtained from Definition 3.1 by adding the following: "a leaf element nodes may be additionally labeled with a parameterized equality predicate of the form $= \#i$, where $\#i$ is a *parameter* and i is an integer."

Example 9.1. Let us add to $\mathcal{P}$ from Example 3.3 the rule

$$f_1(X) \rightarrow trip[maxprice = \#1]//museum(X)$$

Using this rule, we can generate the view v_4 that retrieves museums on trips for which the maximum price is a parameter #1:

$$v_4 : \ doc(T)//vacation//trip/trip[maxprice = \#1]//museum.$$

A user query q_3 that asks for museums with temporary exhibitions on secondary trips that cost at most $1000

$$q_3 : \ doc(T)//vacation//trip/trip[maxprice = 1000]//museum[temp].$$

is then supported by the QSS, because it can be rewritten as

$$doc(v_4)/v_4/museum[temp](1000),$$

where parameter #1 is bound to the value in parenthesis (1000).

We can show that all the tractability and hardness results presented in the previous sections remain valid when text conditions and parameters are added to the setting.

Only minor adjustments are necessary in order to reuse the same PTIME algorithm for expressibility (modulo the new XP syntax and the adapted definitions of mapping and containment). For support as well, our techniques extend in straightforward manner, since the algorithms for deciding equivalence between a tree pattern and a possibly

intersection of tree patterns remain essentially the same, as was shown in Cautis et al. [2008]. Given a query q, the input $QSS^{\#}$ will be transformed into a QSS $\mathcal{P}'$ by replacing each $= \#i$ parameter occurrence by an explicit text equality condition $= C$, for each constant C appearing in q. Further details are omitted.

10. TRACTABILITY BOUNDARIES: COMPENSATED REWRITE PLANS

We consider in this section and in the next one extensions to the rewrite language and to the query set specifications, asking whether the efficient algorithms of the previous sections can be adapted to deal with them.

In this section, we investigate more complex rewrite plans for support, beyond $XP^{\cap}$, taking the compensation idea one step further. More precisely, we consider the rewrite language $XP^{\cap,c}$ which, after the intersection step, might compensate again for equivalence with the input query. We capture $XP^{\cap,c}$ by adding the following rules to the XP grammar:

$$ipath ::= cpath \mid (cpath) \mid (cpath)/rpath \mid (cpath)//rpath$$
$$cpath ::= apath \mid apath \cap cpath.$$

Revisiting Definition 2.10, a rewriting r in the language $XP^{\cap,c}$ is now of the form $\mathcal{I} = (\bigcap_{i,j} u_{ij})$, $\mathcal{I}/rpath$ or $\mathcal{I}//rpath$, with each u_{ij} being of the form $doc(v_j)/v_j/p_i$ or $doc(v_j)/v_j//p_i$.

Example 10.1. Consider the query q_4 below that extracts the temporary exhibitions from the data about museums visited on the same tour trips as in query q_1:

$$q_4 : doc(T)//vacation//trip/trip[guide]//tour[schedule//walk]/museum/temp.$$

There is no rewriting of q_4 using only an intersection of views generated by $\mathcal{P}$, since there is no mention of temporary exhibitions in $\mathcal{P}$. But if an intersection can be compensated, q_4 can be rewritten as the intersection of v_1 and v_2, followed by a one-step navigation:

$$(doc(v_1)/v_1/museum \cap doc(v_2)/v_2/museum)/temp.$$

We prove that support in $XP^{\cap,c}$ becomes NP-hard even for multi-token XP_{es} queries.

THEOREM 10.2. *For a multi-token XP_{es} query q and a QSS $\mathcal{P}$, deciding if q is supported by $\mathcal{P}$ in $XP^{\cap,c}$ is NP-hard.*

The proof of Theorem 10.2 is similar to the one of Theorem 8.1. The intuition behind this result is that an $XP^{\cap,c}$ rewriting r for a query q amounts to finding a rewriting r' in the simpler language $XP^{\cap}$ for a prefix of q and then compensating r' with the remainder of q. Even if q were multi-token, r' may correspond to a prefix of q that is in fact single-token, hence the complexity jump.

We describe next a sound PTIME procedure for support in $XP^{\cap,c}$, which becomes also complete under a certain restriction on the input query.

A *lossless prefix* p of q is any pattern obtained from q by "moving up" the output mark, that is, setting as the output node any main branch node and interpreting what follows that node as predicate (side) branches. Note that lossless prefixes of an XP_{es} query are also in XP_{es}. We can prove the following result, which holds for XP input queries in general.

THEOREM 10.3. *An XP query q is supported by a QSS $\mathcal{P}$ in $XP^{\cap,c}$ iff some lossless prefix of q is supported by $\mathcal{P}$ in $XP^{\cap}$.*

This result enables the following PTIME sound procedure for support in $XP^{\cap,c}$, based on the PTIME decision procedure of Section 7:

ALGORITHM: testSupp$^c(\mathcal{P}, q)$

1. **for each** lossless prefix q' of q
2. **do** $\mathcal{D}$ = findDescSupp($q', comp(\mathcal{P}, q')$)
3. **if** testEquivDesc($q', \mathcal{D}$) **then return** *true*

As a corollary of Theorems 10.3 and 7.22 we obtain that algorithm testSupp^c is also complete when the input query has only multitoken prefixes (this was the case for the input query q_1 of our running example).

COROLLARY 10.4. *Given a QSS $\mathcal{P}$ and an XP_{es} multi-token query q having only multi-token prefixes (i.e., of the form doc(T)//...), algorithm* testSupp^c *is sound and complete for support in $XP^{\cap,c}$.*

Remark. A sound and complete exponential-time algorithm for $XP^{\cap,c}$ support on XP_{es} input queries can be obtained by combining the PTIME algorithm for support in $XP^{\cap}$ for multitoken queries (Section 6) with the exponential-time one for single-token queries (Section A of the appendix). For each lossless prefix of the input query, depending on whether it is multitoken or single-token, the former or the latter procedure will be applied.

11. TRACTABILITY BOUNDARIES: QSS WITH FOREST RIGHT-HAND SIDE

We consider now an extension to the query set specifications, which allows *forests* of tree fragments on the RHS, that is, expansion rules of the form

$$f \rightarrow tf_1, \ldots, tf_k$$

for $tf_1, \ldots, tf_k$ denoting tree fragments. Recall that until now expansion rules were only of the form $f \rightarrow tf$. We call the set specifications in this language QSS^+. This is in line with the original Query Set Specification language of Petropoulos et al. [2003], which also allows forests on the RHS. (The language presented in Petropoulos et al. [2003] has a different syntax from the one we adopted here and in Section C of the electronic appendix we show how that syntax can be compiled into ours.) With this added feature, we show that expressibility and support become NP-hard, even for very restricted tree patterns.

THEOREM 11.1. *Expressibility and support (either in XP or $XP^{\cap}$) are NP-hard for QSS^+, even for input queries and views without //-edges.*

PROOF. *Proof for expressibility.* We detail our proof by considering Boolean tree patterns. The one for patterns of arity 1 is similar. We use once again a reduction from the minimum set-cover problem. Let $(\mathcal{U}, \mathcal{S}, k)$ be an instance of this problem, with $\mathcal{U} = \{e_1, \ldots, e_n\}$ denoting the universe, $\mathcal{S} = \{S_1, \ldots S_m\}$ denoting the sets s.t. $S_i \subset \mathcal{U}$ for each S_i. We want to know whether there exists a subset S' of S, of size at most k, that can cover the entire $\mathcal{U}$ (i.e. each element of $\mathcal{U}$ belongs to at least one set of S').

The reduction takes as input the set $\mathcal{U}$ and $\mathcal{S}$ (size $|\mathcal{S}| \times |\mathcal{S}|$) and the value k (size $lg(k)$).

Let p be the biggest exponent s.t. $2^p \leq k$ and let $b_p b_{p-1} \ldots b_0$ be k's binary representation. We build an instance of the expressibility problem by defining the QSS as follows.

(1) the tree fragments are
$F = \{S, set, f_p^p, f_p^{p-1}, \ldots, f_p^0, f_{p-1}^{p-1}, f_{p-1}^{p-2}, \ldots, f_{p-1}^0, \ldots, f_1^1, f_1^0, f_0^0\}$.
(2) the alphabet $\Sigma = \{a, b, e_1, \ldots, e_m\}$.
(3) S is the start fragment.

(4) the set of productions P defined as follows:
   (a) the start production $S \rightarrow a(X)[b[f_p^p, f_{p-1}^{p-1} \ldots, f_1^1, f_0^0]]$
   (b) $\forall i = \overline{0, p}$ s.t. $b_i = 1$, we have the productions

$$f_i^k \rightarrow f_i^{k-1}, f_i^{k-1}, \forall k = \overline{0, i}, \quad f_i^0 \rightarrow set$$

   (c) $\forall i = \overline{0, p}$ s.t. $b_i = 0$, we have the production $f_i^i \rightarrow ()$
   (d) $\forall S_j = \{e_{l_1}, \ldots, e_{l_j}\} \in S$ we have the production $set \rightarrow e_{l_1}, \ldots, e_{l_j}$
   (e) finally we have the production $set \rightarrow ()$.

The QSS grammar builds Boolean queries having a root node labeled a, with an unique child node labeled b. The children of this b node have e-labels. It is easy to see that the generated queries have branches corresponding to a choice of at most k sets from S (outputted by the expansion of the at most k *set* fragment names).

Let now q be the Boolean tree pattern $a[b[e_1][e_2] \ldots [e_n]]$. First, note that all the queries generated by the QSS program contain q. In order for q to be expressed by this program, there must exist a choice of at most k sets from S that covers all the elements, $e_1, \ldots, e_n$. Hence expressibility holds if and only if we can find in S a cover of $\mathcal{U}$ of maximal size k.

Proof for support. Follows immediately by the same construction. A rewrite candidate is the one that intersects all the possible expansions of the program. □

In Section B of the electronic appendix (available in the ACM Digital Library) we show that the NP lower bound for QSS^+ expressibility and support (both in the absence and presence of ids, under XP_{es}restrictions) is tight for practical purposes, since they can be decided in exponential time. Further details can be found in the appendix.

12. EXPRESSIBILITY AND SUPPORT FOR OTHER XPATH FRAGMENTS

We briefly discuss in this section support and expressibility for other fragments of XPath for which query containment, a crucial test in these problems, is tractable. These have to do with the special node label *wildcard*, denoted $*$, which can be matched with any label in an embedding. It was shown in Miklau and Suciu [2004] that query containment becomes NP-complete when wildcard is used in conjunction with //-edges and predicates. However, the complexity drops to PTIME when any of this three constructions is not used.

The fragment disallowing //-edges leads to less interesting rewrite plans, since intersections of patterns have at most a single possible interleaving. For this fragment, QSS expressibility and support become trivial, as the containment test still requires exhibiting one containment mapping. Note however that they become hard for QSS^+ (Theorem 11.1).

The complexity of support, as the one of rewriting using multiple views, remain open for the fragment that disallows predicates but allows both wildcard and //-edges. The rewriting problem for such linear paths could be solved using automata-based techniques (as in Cautis et al. [2009]), but these are worst-case exponential in the number of views and in the maximal length of sequences alternating wildcard and /-edges (the star length [Miklau and Suciu 2004]). However, we conjecture that our techniques for expressibility apply to this fragment as well, since containment is witnessed by a single containment mapping (with a slight extension to *adorned patterns*, as defined in Miklau and Suciu [2004]).

13. RELATED WORK

XPath rewriting using only one view [Xu and Özsoyoglu 2005; Mandhani and Suciu 2005] or a finite, explicitly given set of views [Balmin et al. 2004; Arion et al. 2007; Tang et al. 2008; Cautis et al. 2008] was the topic of several studies. To the best of our knowledge, we are the first to address the problem of rewriting XPath queries using a compactly specified set of views. The specifications are written in the Query Set Specification(QSS) language [Petropoulos et al. 2003], which was also the basis for building a QBE-like XPath interface in a system for managing biological data [Newman and Özsoyoglu 2004]. The QSS language presented in Petropoulos et al. [2003] has a different syntax from the one we adopted here and in Section C of the electronic appendix we show how that syntax can be compiled into ours.

Expressibility and support were studied in the past for relational queries and sets of relational views specified by Datalog programs [Levy et al. 1999; Vassalos and Papakonstantinou 2000; Cautis et al. 2009]. The work on relational views [Cautis et al. 2009] shares with our article the idea of grouping the views in a finite number of equivalence classes w.r.t their behavior in a rewriting algorithm. Similar is also the strategy of computing these classes (represented by *descriptors*) bottom-up from the specification of the sets of views.

However, relational and XPath queries exhibit very different behaviors. For instance, support and expressibility were shown to be inter-reducible in PTIME for relational queries and views [Cautis et al. 2009], and thus share the same complexity (EXPTIME-complete). This is no longer the case for *XP* queries in the presence of node ids: expressibility is in PTIME (see Section 4), while support is coNP-hard. The PTIME results we obtain make crucial use of the tree shape of XPath queries and require problem-specific restrictions that do not follow from the relational work. Indeed, it was shown in [Cautis et al. 2009] that support and expressibility on relational queries are undecidable under integrity constraints. Note that integrity constraints are a necessary ingredient in order to capture explicitly tree-shape constraints that are implicit for XML data.

For implementing security policies, a complementary approach to specifying sets of views consists in annotating the DTD of the source with *access annotations* that can be used to allow/disallow access to parts of the data [Fan et al. 2004, 2007]. The system infers *one view* over the input document that conforms to the annotations and publishes the DTD of this view. Clients are allowed to ask any queries over the view DTD. This architecture is designed for security scenarios and does not extend to querying sources with limited capabilities.

Comparison with prior publication. This article is based on an earlier extended abstract [Cautis et al. 2009], whose main focus was on tractable algorithms for support and expressibility. In this paper we provide a more complete study on the computational complexity of these problems, describing also decision procedures for settings in which support or expressibility were shown to be hard. In addition to providing the complete proofs for all our results, this article also makes several new nontrivial contributions. We describe how our algorithms for expressibility and support can be modified to also provide an equivalent view or rewrite plan. We added a detailed description of the handling of richer query set specifications, adapting the previous definitions of view descriptors and giving a procedure for finding them. We present the handling of compensated rewrite plans for support, which was only claimed in the extended abstract: we describe both a tractable algorithm, that is sound in general and complete under fairly permissive restrictions, and a complete algorithm that runs in exponential time. Then, for single-token input queries, we describe an algorithm that decides support in exponentialtime, that is, the best we can hope for given the NP lower bound. We also discuss here how the query set specifications (QSS) of Petropoulos et al. [2003]

Table I. The Complexity of the Expressibility Problem

data model	input queries	views encoding (XP)	
XML	XP	QSS	in PTIME
XML	XP	QSS$^+$	in EXPTIME NP-hard

Table II. The Complexity of the Support Problem

data model	input queries	views encoding (XP)	rewritings	
XML with copy semantics	XP	QSS	XP	in PTIME
XML with copy semantics	XP	QSS$^+$	XP	in EXPTIME NP-hard
XML with node ids	XP	QSS	$XP^\cap$	coNP-hard
XML with node ids	XP_{es} single-token	QSS	$XP^\cap$	in EXPTIME NP-hard
XML with node ids	XP_{es} multi-token	QSS	$XP^\cap$	in PTIME
XML with node ids	XP_{es}	QSS	$XP^{\cap,c}$	in EXPTIME NP-hard
XML with node ids	XP_{es}	QSS$^+$	$XP^{\cap,c}$	in EXPTIME NP-hard

Table III. The Complexity of the Rewriting Problem with Explicit Views

data model	input queries	views	rewritings	
XML with copy semantics	XP	XP	XP	in PTIME
XML with node ids	XP	XP	$XP^\cap$ or $XP^{\cap,c}$	in EXPTIME coNP-hard
XML with node ids	XP_{es}	XP	$XP^{\cap,c}$	in PTIME

can be compiled into our syntax, by making explicit the output node and eliminating all occurrence constraints.

Moreover, we provide examples illustrating the step-by-step computation of descriptors for support. We also give complete details on techniques for which only a high-level description was given in the extended abstract: we detail an important component of the expressibility algorithm, namely the tf-cover function, as well as the steps for computing support descriptors, which were mostly sketched before.

14. CONCLUSION

We study the problems of expressibility and support of an XPath query by XPath views generated as expansions of a Query Set Specification. We focus on efficiency and PTIME algorithms, ensuring that they are sound in general and identifying the most permissive restrictions under which they become complete.

We find that for XPaths corresponding to the fragment having child and descendant navigation and no wildcard, expressibility can be solved in PTIME.

For support, the complexity analysis is more refined, as it depends on the rewriting language. In the case in which the XML nodes in the result of the views lose their original identity, we are able to give a PTIME algorithm for support. If the source exposes persistent node ids, which enable rewritings that intersect several views with compensation, we show that the problem becomes NP-hard unless fairly permissive restrictions on the user query are placed. We present a sound PTIME algorithm that also becomes complete under the restrictions. For the hard cases, we prove that the lower

bounds are tight for practical purposes, by describing exponential-time algorithms for support.

We consider extensions to the rewrite language and to the query set specifications. We show that our algorithms can be extended to deal with parameters, but richer rewrite languages and specifications exhibit a complexity gap: expressibility and support become NP-hard, even under strong restrictions on the queries and views. This lower-bound is proven tight for practical purposes, as the initial algorithms can be adapted for these extensions, with worst-case exponential time complexity.

We summarize our results on the computational complexity of expressibility and support for XPath in Tables I and II. For comparison, we also chart in Table III the complexity of rewriting using explicit views, both in the absence of ids [Xu and Özsoyoglu 2005] and in their presence [Cautis et al. 2008].

An interesting direction for future work is to investigate even richer rewrite languages, that can have arbitrarily many levels of intersection and compensation. This direction is indeed promising, yielding strictly more rewrite opportunities, yet challenging, since even the question whether view-based rewriting for XP_{es} input queries allows tractable decision procedures remains open under such plans.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

REFERENCES

ABITEBOUL, S., HULL, R., AND VIANU, V. 1995. *Foundations of Databases*. Addison-Wesley.

AMER-YAHIA, S., CHO, S., LAKSHMANAN, L., AND SRIVASTAVA, D. 2002. Tree pattern query minimization. *VLDB J. 11,* 4.

ARION, A., BENZAKEN, V., MANOLESCU, I., AND PAPAKONSTANTINOU, Y. 2007. Structured materialized views for XML queries. In *Proceedings of the International Conference on Very Large Databeses*. 87–98.

BALMIN, A., ÖZCAN, F., BEYER, K. S., COCHRANE, R., AND PIRAHESH, H. 2004. A framework for using materialized XPath views in XML query processing. In *Proceedings of the International Conference on Very Large Databeses*. 60–71.

BENEDIKT, M., FAN, W., AND KUPER, G. M. 2005. Structural properties of XPath fragments. *Theor. Comput. Sci. 336,* 1, 3–31.

CAUTIS, B., ABITEBOUL, S., AND MILO, T. 2009. Reasoning about XML update constraints. *J. Comput. Syst. Sci. 75,* 6.

CAUTIS, B., DEUTSCH, A., AND ONOSE, N. 2008. XPath rewriting using multiple views: Achieving completeness and efficiency. In *Proceedings of the International Workshop on Web and Databases*.

CAUTIS, B., DEUTSCH, A., AND ONOSE, N. 2009. Querying data sources that export infinite sets of views. In *Proceedings of the International Conference on Database Theory*. 84–97.

CAUTIS, B., DEUTSCH, A., ONOSE, N., AND VASSALOS, V. 2009. Efficient rewriting of XPath queries using query set specifications. *Proc. VLDB Endow. 2,* 1.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms, 2nd Ed.* The MIT Press.

FAN, W., CHAN, C. Y., AND GAROFALAKIS, M. N. 2004. Secure XML querying with security views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 587–598.

FAN, W., GEERTS, F., JIA, X., AND KEMENTSIETSIDIS, A. 2007. Rewriting regular XPath queries on XML views. In *Proceedings of the International Conference on Data Engineering*. 666–675.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

HOPCROFT, J. E. AND ULLMAN, J. D. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

LEVY, A. Y., RAJARAMAN, A., AND ULLMAN, J. D. 1999. Answering queries using limited external query processors. *J. Comput. Syst. Sci. 58,* 1, 69–82.

MANDHANI, B. AND SUCIU, D. 2005. Query caching and view selection for xml databases. In *Proceedings of the International Conference on Very Large Databeses*. 469–480.

MIKLAU, G. AND SUCIU, D. 2004. Containment and equivalence for a fragment of xpath. *J. ACM 51,* 1, 2–45.

NEWMAN, S. AND ÖZSOYOGLU, Z. M. 2004. A tree-structured query interface for querying semi-structured data. In *Proceedings of the International Conference on Statistical and Scientific Database Management*. 127–130.

PAPAKONSTANTINOU, Y., GUPTA, A., GARCIA-MOLINA, H., AND ULLMAN, J. D. 1995. A query translation scheme for rapid implementation of wrappers. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*. 161–186.

PETROPOULOS, M., DEUTSCH, A., AND PAPAKONSTANTINOU, Y. 2003. The Query Set Specification Language (QSSL). In *Proceedings of the International Workshop on Web and Databases*. 99–104.

TANG, N., YU, J. X., ÖZSU, M. T., CHOI, B., AND WONG, K.-F. 2008. Multiple materialized view selection for xpath query rewriting. In *Proceedings of the International Conference on Data Engineering*. 873–882.

VASSALOS, V. AND PAPAKONSTANTINOU, Y. 2000. Expressive capabilities description languages and query rewriting algorithms. *J. Log. Program. 43,* 1, 75–122.

XU, W. AND ÖZSOYOGLU, Z. M. 2005. Rewriting XPath queries using materialized views. In *Proceedings of the International Conference on Very Large Databeses*. 121–132.

Received January 2010; accepted June 2010