Efficient Rewriting of XPath Queries Using Query Set Specifications

Bogdan Cautis
Telecom ParisTech
cautis@telecom-paristech.fr

Alin Deutsch
UC San Diego
deutsch@cs.ucsd.edu

Nicola Onose
UC San Diego
nicola@cs.ucsd.edu

Vasilis Vassalos
Athens Univ. of Economics and Business
vassalos@aueb.gr

ABSTRACT

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 [19], 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.

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 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 [14, 21, 8], 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 [16]. As a formalism for compactly representing large sets of such views, we adopt a variation of the Query Set Specification Language(QSSL) [19], a grammar-like formalism for specifying XPath view families (see also [17]).

Expressibility and support. As in the literature on sources exporting sets of legal relational queries [21, 8], 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 post-processing of its result. The corresponding decision problem is called *expressibility* [8]: we say that query q is *expressible* if it is equivalent to a view published by the source. The second setting is when the capabilities of the source are extended by a *source wrapper* [18] 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* [8]: 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 [22, 15] on XPath rewriting studied the problem of equivalently rewriting an XPath 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 [5]. 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 paper, 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 paper we consider the example*

VLDB '09, August 24-28, 2009, Lyon, France

of a tourism agency that allows to find organized trips matching user 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 [7], 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 [8]. 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 [14, 21], where already the simple language of conjunctive queries leads to EXPTIME completeness of expressivity and support [8], but on the other hand is closed under intersection, which poses no additional problem.

Outline of the paper. The paper 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 presence (Sections 6, 7, 8). QSS and rewriting language extensions are presented in Sections 9, 10. Section 11 discussed related work and Section 12 concludes.

2. XPATH AND TREE PATTERNS

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

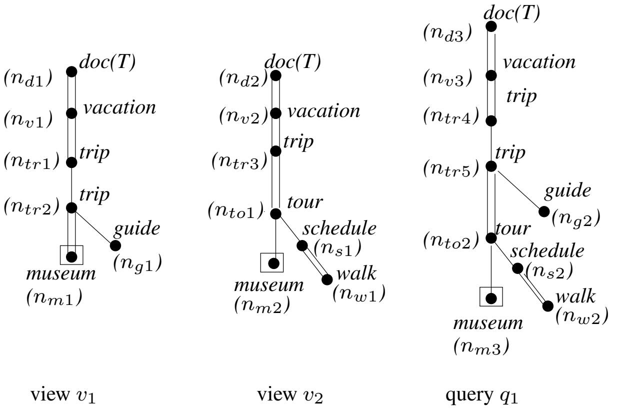

Figure 1: The tree patterns of queries v_1, v_2 and q_1

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

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

The sub-expressions 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 paper, and 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* [16]:

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

Any *XP* expression can be translated into a tree pattern query and vice versa (see, for instance [16]). For a given tree pattern query p, *xpath*(p) is the associated *XP* expression.

EXAMPLE 2.1. *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 2.2. *An* embedding *of a tree pattern p into a tree t over Σ is a function e from* $\text{NODES}(p)$ *to* $\text{NODES}(t)$ *that has the following properties: (1)* $e(\text{ROOT}(p)) = \text{ROOT}(t)$*; (2) for any* $n \in \text{NODES}(p)$, $\text{LABEL}(e(n)) = \text{LABEL}(n)$*; (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: $\{e(\text{OUT}(p)) \mid$ *e is an embedding of p into t* $\}$

DEFINITION 2.3. *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* [3] if no pattern $p' \equiv p$ can have fewer nodes than p.

DEFINITION 2.4. *A mapping between two tree patterns p_1, p_2 is a function* $h : \text{NODES}(p_1) \rightarrow \text{NODES}(p_2)$ *satisfying properties (2),(4) of an embedding (allowing the target to be a pattern) plus three others: (5) for any* $n \in \text{MBN}(p_1)$, $h(n) \in \text{MBN}(p_2)$*; (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* $h(\text{OUT}(p_1)) = \text{OUT}(p_2)$*. A* containment mapping *denotes a mapping that is simultaneously a root-mapping and an output-mapping.*

Previous studies [3, 16] 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.

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

Intersection. We consider in this paper the extension *XP*$^\cap$ of *XP* with respect to intersection, having a straightforward semantics. Its grammar is obtained from that of *XP* by adding the following rule:

$$ipath ::= apath \mid apath \cap ipath$$

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 [6], a *code* is a string of symbols from Σ, alternating with either / or //.

DEFINITION 2.5 (INTERLEAVING). *A* interleaving *of a finite set of tree patterns* $\mathcal{S}$ *is any tree pattern* p_i *produced as follows:*

1. *let* $M = \cup_{p \in S}\text{MBN}(p)$,
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* $n \in M, \text{LABEL}(f_i(n)) = \text{LABEL}(n)$
   (b) *for any* $p \in \mathcal{S}$, $f_i(\text{ROOT}(p))$ *is the first symbol of* i,
   (c) *for any* $p \in \mathcal{S}$, $f_i(\text{OUT}(p))$ *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*
   $$\ldots f_i(n_1)\ldots f_i(n_2)\ldots.$$
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 above. The tree patterns* p_i *thus obtained are called* interleavings *of* $\mathcal{S}$ *and we denote their set by* interleave($\mathcal{S}$).

EXAMPLE 2.2. *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 2.1. *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 2.2. *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 (see [6]).

View-based rewriting. Given 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 $\{doc(v) | v \in \mathcal{V}\}$, in which the topmost element is labeled with the view name. 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 2.6. *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

- each query *unfold*(u_{ij}) contains q, and
- by Lemmas 2.1 and 2.2, q contains all the tree patterns (interleavings) in $interleave(\{unfold(u_{ij})\})$.

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* (resp. *//-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 |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 descendants of n along with their predicates. A *suffix* p' of a tree pattern p is any subtree of p rooted at a node in $\text{MBN}(p)$.

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, i.e., 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 sub-sequences separated by //-edges, and each sub-pattern corresponding to such a sub-sequence 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 [19].

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

- F *is the set of tree fragment names*
- Σ*, with* $\Sigma \cap F = \emptyset$ *is the set of element names*
- $S \in F$ *is the start tree fragment name*
- 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:

- *element nodes, labeled with symbols from Σ,*
- *tree fragment nodes n labeled with symbols from F,*
- *edges either of child type, denoted by simple lines, or of descendant type, denoted by double lines,*
- *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:

- starting from a rule $S(X) \rightarrow tf$,
- 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 $f(X)$.
- 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* expressible *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 of view results. In this case, one can choose $\mathcal{L}_R$ to be $XP^{\cap}$ and use several views in more complex rewritings.

EXAMPLE 3.1. *The QSS $\mathcal{P}$ below 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.*

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

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:

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

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 $\mathcal{P}$:

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

Consider the query q_1 of Example 1.1. One can check that q_1 cannot be answered by navigating into a single view. Suppose now that the views expose persistent node ids. By using Lemmas 2.1, 2.2, one can check that the support of q_1 is witnessed by v_1 and v_2:

$$q_1 \equiv doc(v_1)/v_1/museum \cap doc(v_2)/v_2/museum.$$

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.

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:

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

For any fragment name f and rule

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

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 3.2. *The result of normalizing the QSS $\mathcal{P}$ from Example 3.1 is the following specification:*

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

4. EXPRESSIBILITY

We consider in this section the problem of expressibility: given a query q and a QSS $\mathcal{P}$ encoding a set of 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 unfeasible 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 (w.r.t. 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* w.r.t. their behavior in the algorithm for checking equivalence with q. Since there are fewer (only polynomially many) possible behaviors, manipulating such equivalence classes instead of explicit views or fragments thereof enables our PTIME solution.

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

- *mapping descriptors*, which record if some expansion of a tree fragment name maps into a subtree of q,
- *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* ([16]). 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 4.2, we show how expressibility for tree patterns (arity 1) can be reduced to expressibility for boolean tree patters.

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 $f() \to l[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()]$ 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, i.e.

- q is isomorphic to (part of) v, i.e. there is a containment mapping ψ from q into v, and the inverse ψ^{-1} is a partial mapping from v into q,
- 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 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) Similarly, for each pair of nodes (n_1, n_2), $n_1 \in \text{NODES}(v)$ $n_2 \in \text{NODES}(q)$ 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 descendant 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. More precisely,

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 4.1. *Suppose that the data source publishes a modified version of the QSS from Example 3.2, 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):*

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

A client interface generates and sends a query identical to v_2 of Example 2.1 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 3.2.

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_{w1}, 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 $n \in N$. This function is based on a max-flow computation and its running time is $O((|C| + |N|) * |C|)$. We refer the reader to the extended version of this paper [9], Appendix A, for the detailed definition of tf-cover.

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. 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.

```
Algorithm findDescExpr($q$, $\mathcal{P}$):
A. Start with an empty set of descriptors $R$.
B. For each rule $f() \to ()$, node $n \in \text{NODES}(q)$, add to $R$ the
   descriptor $map(f, n)$.
C. For each rule $f() \to l$ (i.e., the RHS has only one node) and
   each node $n \in \text{NODES}(q)$ labeled by $l$, add to $R$ the descriptors
   $equiv(f, n)$ and $map(f, n)$.
Repeat until $R$ unchanged:
D. For each rule $f() \to l[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()]$,
   add to $R$ a descriptor $map(f, n)$ if $n$ is labeled by $l$ and
   • for each fragment name $c_i$ there exists a descriptor
     $map(c_i, n')$ s.t. $n'$ is a /-child of $n$,
   • for each fragment name $d_j$ there exists a descriptor
     $map(d_j, n')$ s.t. $n'$ is a descendant of $n$.
E. for each rule $f() \to l[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()]$:
   add to $R$ the descriptors $equiv(f, n)$ and $map(f, n)$ if
   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 exists a descriptor
      $map(c_i, n')$ s.t. $n'$ is a /-child of $n$,
   4. for each fragment name $d_j \notin D$ there exists a descriptor
      $map(d_j, n')$ s.t. $n'$ is a descendant of $n$.
```

THEOREM 4.1. *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.*

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

4.2 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 [16], 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 4.2. *A tree pattern query q is expressed by a QSS $\mathcal{P}$ iff the boolean tree pattern q_0 is expressed by the QSS $\mathcal{P}_0$.*

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 query 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 [22], 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 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 [7], 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 w.r.t. 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.

5.1 Support in the absence of ids

When persistent identifiers are not exposed, a rewrite plan consist 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 [22], 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, $\text{compensate}(v, q, k)$ 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, i.e. a/b[c][d]/e.

We can reformulate the result from previous literature as follows:

THEOREM 5.1 ([22]). *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* $\text{compensate}(v, q, k) \equiv q$*.*

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 w.r.t. 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, i.e., of the form

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

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

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

We can prove the following:

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

EXAMPLE 5.1. *An example of support in the absence of persistent ids has already been given in Example 3.1: 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 problems 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 under an explicit set of *XP* views becomes coNP-hard, as it was shown in [7].

THEOREM 6.1 ([7]). *Testing if an XP query can be rewritten using explicitly listed views, in the presence of ids, is coNP-hard.*

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

COROLLARY 6.1. *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 [7]. 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, i.e., obtain a decision procedure for queries outside this fragment, since the problem of support subsumes the rewriting problem.

The remainder of this paper 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 *multi-token*, continues to be in PTIME (Theorem 7.5), but the complementary sub-fragment 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 multi-token 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 all the navigation steps on a document. After defining the fragment of extended skeletons, we consider in Section 7 support for multi-token queries and, in Section 8, support for single-token queries.

Extended skeletons (*XP*$_{es}$). Intuitively, this fragment limits the use of //-edges in predicates, in a way which can be summarized as follows: a token t of a pattern p will not have predicates 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 having the following property: for any main branch node n 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.

7. MULTI-TOKEN QUERIES

We consider in this section 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). The extension to general *XP*$^{\cap}$ plans, i.e., intersections of (possibly compensated) views, is detailed in Section 7.4.

As in the case of expressibility, we think of views as grouped into equivalence classes w.r.t. to crucial tests for support. We manipulate such classes, which are represented by *view descriptors*, instead of explicit views, avoiding the enumeration of a potentially large space of views and rewrite plans. Given that a QSS constructs views by putting together fragments, we have to construct our 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 [7] 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 w.r.t. the tests of testEquiv, and *view descriptors* as a means to represent such classes. We reformulate the testEquiv algorithm into a new algorithm, testEquivDesc, that runs on view descriptors instead of explicit view definitions. Finally, 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 multi-token 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 intermediary part (m may be empty). Let $\mathcal{V} = \{v_1, \ldots, v_k\}$ denote a set of views such that $q \sqsubseteq v_i$ for each view v_i. Let each view v_i be of the form $v_i = ft_i//m_i//lt_i$.

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 $\text{MB}(ft_i)$ is a prefix of $\text{MB}(ft)$. 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 7.1. *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

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

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

EXAMPLE 7.2. *For instance, for the query*

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

l_q is well-defined, as $l_q = \textit{doc}(T)/\textit{vacation}/\textit{tour}/\textit{museum}$.

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

We are now ready to present our reformulation of the PTIME algorithm of [7], which will test that $\cap\mathcal{V} \sqsubseteq q$. By Lemma 2.2, 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:

- the first token of q can be mapped in the first token of i s.t. the image of $\text{ROOT}(q)$ is $\text{ROOT}(i)$,
- the last token of q can be mapped in the last token of i s.t. the image of $\text{OUT}(q)$ is $\text{OUT}(i)$,
- the images of these two tokens in i are disjoint,
- the intermediary part m (if non-empty) of q can be mapped somewhere between the two images in i.

Algorithm 1 testEquiv($\mathcal{V}$, q)

```
 1: let each $v_i = ft_i//m_i//lt_i$, let $q = ft//m//lt$
 2: compute the patterns $ft_{\mathcal{V}}$ and $lt_{\mathcal{V}}$
 3: compute the pattern $l_q$
 4: compute the patterns $\text{MB}(ft)_m$ and $\text{MB}(lt)_m$
 5: if $ft_{\mathcal{V}} \equiv ft$ and $lt_{\mathcal{V}} \equiv lt$ then
 6:     if $m$ is empty then for each $v_i \in \mathcal{V}$
 7:         if $\text{MB}(v_i)$ does not map into $l_q$ then output true
 8:     else ($m$ non-empty) for each $v_j \in \mathcal{V}$
 9:         if $v_j$ can be seen as $prefix_j//m'//suffix_j$ s.t.
10:             $m' \equiv m$
11:             $prefix_j$ root-maps into $ft$, $suffix_j$ output-maps into $lt$
12:             $\text{MB}(prefix_j)$ does not root-map into $\text{MB}(ft)_m$
13:             $\text{MB}(suffix_j)$ does not output-map into $\text{MB}(lt)_m$
14:         then output true
```

We can prove the following:

THEOREM 7.1. *For a multi-token XP query q and a set of XP views $\mathcal{V}$,* testEquiv *is a sound PTIME procedure for testing $q \equiv \cap\mathcal{V}$.*

For XP_{es} multi-token queries, we can also prove completeness:

THEOREM 7.2. *For an XP_{es} multi-token query q and a set of XP views $\mathcal{V}$,* testEquiv *is complete for testing $q \equiv \cap\mathcal{V}$.*

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 one does not need the complete definitions 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 view descriptors in algorithm testEquivDesc. More precisely, assuming we are dealing with expansions of a QSS $\mathcal{P}$ with start fragment name S,

For line 5 of testEquiv**.** For the part $ft_{\mathcal{V}} \equiv ft$: a *first-token descriptor* will be a tuple $\mathbf{ft}(\mathbf{S}, \mathbf{p})$, 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 will belong to at least one such class, but may be belong to several (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 $\mathbf{ft}$ and $\mathbf{lt}$ 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 7 of testEquiv**.** An *l-descriptor* is a tuple $\mathbf{l}(\mathbf{S})$, indicating that there exists an expansion v verifying $q \sqsubseteq v$ and $l_q \not\sqsubseteq \text{MB}(v)$. (This type of descriptor is an alias for the condition of line 7, 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 various tests.

Algorithm 2 testEquivDesc($\mathcal{D}$, q)

```
from all descriptors $\mathbf{ft}(\mathbf{S}, \mathbf{p}) \in \mathcal{D}$ compute the pattern $ft_{\mathcal{V}}$
from all descriptors $\mathbf{lt}(\mathbf{S}, \mathbf{p}) \in \mathcal{D}$ compute the pattern $lt_{\mathcal{V}}$
if $lt_{\mathcal{V}} \equiv ft$ and $lt_{\mathcal{V}} \equiv ft$ then
    if $m$ is empty then
        if there exists a descriptor $\mathbf{l}(\mathbf{S}) \in \mathcal{D}$ then output true
    else if there exists a descriptor $\mathbf{m}(\mathbf{S}) \in \mathcal{D}$ then output true
```

THEOREM 7.3. *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.*

EXAMPLE 7.3. *For the query q_1 in Example 1.1, ft = doc(T), m = vacation//trip/trip[guide], lt = tour[schedule//walk]/museum.*

For the QSS $\mathcal{P}$ from Example 3.1 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 multi-token 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 7.3 and 7.1, findDescSupp delivers a sound PTIME algorithm for support when the input queries are multi-token from *XP*. Moreover, via Theorems 7.3 and 7.2, it delivers a PTIME decision procedure for support when the input queries are multi-token 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 5, 10 and 11 in testEquiv similarly, by descriptors which record mapping or equivalence details.

For line 7, the non-existence 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:

- test if the last token of l_2 maps in the last token of l_1, such that $\mathrm{OUT}(l_1)$ is the image of $\mathrm{OUT}(l_2)$. Let k denote the start rank (the upmost node) of this mapping image.
- 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 non-existence 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.
- 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) $\mathrm{ROOT}(l_1)$ is the image of $\mathrm{ROOT}(l_2)$, 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 $\mathrm{MB}(\textit{suffix}_j)$ and, symmetrically, a top-down one can be used in the case of $\mathrm{MB}(\textit{prefix}_j)$.

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 to a full containment mapping (for line 7), 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 //).

We are now ready to detail how view descriptors are computed in the algorithm findDescSupp. We start by assuming that all *equiv* and *map* descriptors are pre-computed for the 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 $n \in \mathrm{MBN}(q)$, (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. Examples illustrating the step-by-step computation of descriptors are given in [9], Appendix G.

7.3.1 Computing first-token descriptors

For this part, we will use *prefix descriptors* for fragment names:

DEFINITION 7.1. 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* $|\mathrm{MB}(ft)|$ *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 *s.t. (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).*

Step 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 $\mathbf{pref(f, l, k)}$ for each rank k, $1 \leq k \leq |\mathrm{MB}(ft)|$, s.t. $node_q(k)$ has label l, for which we can infer that v_f contains the pattern $q(k)$, by the following tests:
   - for each fragment name c_i there exists a descriptor $map(c_i, n)$, for n being a /-child of $node_q(k)$,
   - for each fragment name d_j there exists a descriptor $map(d_j, n)$, for n being a descendant of $node_q(k)$
   - there exists a containment descriptor $contain(g, n)$ for n being any main branch node of rank $k' > k$ in q.

   Moreover, if for a /-predicate (resp. //-predicate) P on $node_q(k)$ we have a descriptor $equiv(c_i, root_P)$ (resp. $equiv(d_j, root_P)$), add the descriptor $\mathbf{pref(f, l[P], k)}$.
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 $\mathbf{pref(f, l/p', k)}$, for $k = k' - 1$, if $node_q(k)$ has label l and we can infer that v_f contains $q(k)$, as follows:
   - for each fragment name c_i there exists a descriptor $map(c_i, n)$, for n being a /-child of $node_q(k)$,
   - for each fragment name d_j there exists a descriptor $map(d_j, n)$, for n being a descendant of $node_q(k)$

   Moreover, if for a /-predicate (resp. //-predicate) P on $node_q(k)$ we have a descriptor $equiv(c_i, root_P)$ (resp. $equiv(d_j, root_P)$), add also $\mathbf{pref(f, l[P]/}\mathrm{MB}(p')\mathbf{, k)}$.
3. Whenever a descriptor *pref(f,p,1)* is obtained, for $f = S$, add $\mathbf{ft(S, p)}$ to the set of view descriptors.

7.3.2 Computing last-token descriptors

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

DEFINITION 7.2. 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 7.3. 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 7.2 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.*

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

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 $\mathbf{lt(S, p)}$ to the set of view descriptors. (For the explicit steps we refer the reader to [9], Appendix B.1.)

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

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 chose 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 7.4. 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.*

Step 3 **of** findDescSupp($q, \mathcal{P}$)**.** Iterate the following steps:

1. For rules $f(X) \rightarrow l(X)[c_1(), \dots c_n(), .//d_1(), \dots, .//d_m()]$, if we can infer that v_f contains the subtree of q rooted at $\text{OUT}(q)$, add a descriptor $\mathbf{pl}[\mathbf{f}, |\text{MB}(q)|, (|\mathbf{l_q}| + \mathbf{1}, \mathbf{l})]$
2. For $f(X) \rightarrow l[\dots]/g(X)$, given a descriptor $pl[g, k_1', (k_2', p')]$, if we can infer that v_f contains the pattern $q(k_1' - 1)$:
   - if f is not the start fragment name, add the descriptor $\mathbf{pl}[\mathbf{f}, \mathbf{k_1'} - \mathbf{1}, (\mathbf{k_2'}, \mathbf{l/p'})]$.
   - otherwise, if there is no mapping of l/p' into l_q whose image starts at $\text{ROOT}(l_q)$ and ends above k_2', add the descriptor $\mathbf{l(S)}$ to the set of view descriptors.
3. For $f(X) \rightarrow l[\dots]//g(X)$, descriptor $pl[g, k_1', (k_2', p')]$, for each rank k_1, $1 \leq k_1 < k_1'$, s.t. we can infer that
   - 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 descriptor $\mathbf{pl}[\mathbf{f}, \mathbf{k_1}, (\mathbf{k_2}, \mathbf{l})]$.
   - otherwise, if there is no mapping of $l//p'$ into l_q whose image starts at $\text{ROOT}(l_q)$ and ends above k_2', add the descriptor $\mathbf{l(S)}$ to the set of view descriptors.

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

- $prefix_j$ root-maps into ft but $\text{MB}(prefix_j)$ cannot root-map into $\text{MB}(ft)_m$,
- $suffix_j$ output-maps into lt, but $\text{MB}(suffix_j)$ cannot output-map into $\text{MB}(lt)_m$.

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

- g has an expansion v_g that gives us the part $m'//suffix_j$,
- 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 7.5. 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* $\text{MB}(q)$.

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 $\text{MB}(lt)_m$; by convention, k_2 is $|\text{MB}(q)| + 1$ 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 7.6. 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 described above.*

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

DEFINITION 7.7. 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 $f(X) \rightarrow l[\dots]/g(X)$ or $f(X) \rightarrow l[\dots]//g(X)$, we will use an am-descriptor for f to infer one for g.

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

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 $\mathbf{m(S)}$ to the set of view descriptors. (For more details we refer the reader to [9], Appendix B.2.)

We can now prove the following:

THEOREM 7.4. *Given a QSS $\mathcal{P}$ and a multi-token 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.*

By Theorems 7.4, 7.3 and 7.1, 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 7.2, 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.

7.4 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* multi-token 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, which builds expansions with compensation.

EXAMPLE 7.4. *Suppose that the QSS of the source in Example 3.1 is modified to return the guided trips themselves instead of the museums of those trips, 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 7.5. *Given a QSS $\mathcal{P}$ and a multi-token XP query q, let $\mathcal{D}$:=* findDescSupp$(q, comp(\mathcal{P}, q))$.

1. *Algorithm* testEquivDesc*(q,$\mathcal{D}$) is sound for support in $XP^{\cap}$, i.e., 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}, i.e. q is supported by $\mathcal{P}$ in $XP^{\cap}$ iff* testEquivDesc*(q,$\mathcal{D}$) outputs true.*

Remark. In a setting in which one needs to also find a witness for support, this can be done by bookkeeping at each step besides descriptors a *representative*, an arbitrarily chosen view or view fragment from the equivalence class. More details can be found in [9].

8. SINGLE-TOKEN QUERIES

We consider in this section the remaining sub-fragment 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.*

The surprising dichotomy between support for single-token and multi-token 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 coming from 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 only way possible the first and last tokens of the views), while the section in between has to be entirely present (isomorphic modulo minimization) in some view.

The proof of Theorem 8.1 is given in [9]. We also give an exponential-time algorithm that decides support for XP_{es} single-token queries (the best we can hope for given the NP lower bound). Finally, we give in a sound PTIME algorithm for this problem.

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:

$pred$ ::= ϵ | [*rpath*] | [*rpath* $= C$] | [.//*rpath*] | [.//*rpath* $= C$] | *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 obtain 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 3.1 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 algorithms for expressibility and support (modulo the new *XP* syntax and the adapted definitions of mapping and containment). 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

We consider now 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.

Compensated rewriting plans. We consider in this section 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 grammar of *XP*:

$$\begin{array}{lll} ipath & ::= & cpath \mid (cpath) \mid (cpath)/rpath \mid (cpath)//rpath \\ cpath & ::= & apath \mid apath \cap cpath \end{array}$$

Revisiting Definition 2.6, 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 $\mathcal{P}$. However, if we allow the intersection to 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 10.1. *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 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.

In [9] we also show that support in $XP^{\cap,c}$ can be solved in exponential-time for XP_{es} input queries, i.e. optimal for practical purposes.

QSS with forest RHS. We consider now an extension to the query set specifications, which allows *forests* of tree fragments on the RHS, i.e., expansion rules of the form $f \rightarrow tf_1, \ldots, tf_k$.

We call the set specifications in this language QSS^+. With this added feature, we show that expressibility and support become NP-hard, even for very restricted tree patterns, without //-edges.

THEOREM 10.2. *Expressibility is NP-hard for QSS^+, even for XP queries and views without //-edges. Support is NP-hard for QSS^+, for XP queries and views without //-edges in predicates.*

We refer the reader to [9] for further details. There, we also show that QSS^+ expressibility can be solved in exponential time.

11. RELATED WORK

XPath rewriting using only one view [22, 15] or a finite, explicitly given set of views [5, 4, 20, 7] was the object 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 [19], which was also the basis for building a QBE-like XPath interface in a software system for managing biological data [17]. The QSS language presented in [19] has a different syntax from the one we adopted here and in [9] 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 [14, 21, 8]. The work on relational views [8] shares with our paper 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 [8], 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.

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 [11]. 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.

12. CONCLUSION

We study the problems of expressibility and support of an XPath query by XPath views generated as expansions of a Query Set Specification. Since we focus on efficiency, we consider only 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, 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.

13. REFERENCES

[1] S. Abiteboul, Z. Abrams, S. Haar, and T. Milo. Diagnosis of asynchronous discrete event systems: Datalog to the rescue! In *PODS*, pages 358–367, 2005.

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

[3] S. Amer-Yahia, S. Cho, L. Lakshmanan, and D. Srivastava. Tree pattern query minimization. *VLDB J.*, 11(4), 2002.

[4] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou. Structured materialized views for XML queries. In *VLDB*, 2007.

[5] A. Balmin, F. Özcan, K. S. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, 2004.

[6] M. Benedikt, W. Fan, and G. Kuper. Structural properties of XPath fragments. *Theor. Comput. Sci.*, 336(1), 2005.

[7] B. Cautis, A. Deutsch, and N. Onose. XPath rewriting using multiple views: Achieving completeness and efficiency. In *WebDB*, 2008.

[8] B. Cautis, A. Deutsch, and N. Onose. Querying data sources that export infinite sets of views. In *ICDT*, 2009.

[9] B. Cautis, A. Deutsch, N. Onose, and V. Vassalos. Efficient rewriting of XPath queries using Query Set Specifications, 2009. TR CS2009-0941, UCSD. Available from http://db.ucsd.edu/index.jsp?pageStr=publications.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[11] W. Fan, C. Y. Chan, and M. N. Garofalakis. Secure XML querying with security views. In *SIGMOD Conference*, pages 587–598, 2004.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[14] A. Y. Levy, A. Rajaraman, and J. D. Ullman. Answering queries using limited external query processors. *JCSS*, 58(1), 1999.

[15] B. Mandhani and D. Suciu. Query caching and view selection for XML databases. In *VLDB*, 2005.

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

[17] S. Newman and Z. M. Özsoyoglu. A tree-structured query interface for querying semi-structured data. In *SSDBM*, pages 127–130, 2004.

[18] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. D. Ullman. A query translation scheme for rapid implementation of wrappers. In *DOOD*, 1995.

[19] M. Petropoulos, A. Deutsch, and Y. Papakonstantinou. The Query Set Specification Language (QSSL). In *WebDB*, pages 99–104, 2003.

[20] N. Tang, J. Yu, T. Özsu, B. Choi, and K. Wong. Multiple materialized view selection for XPath query rewriting. In *ICDE*, 2008.

[21] V. Vassalos and Y. Papakonstantinou. Expressive capabilities description languages and query rewriting algorithms. *J. Log. Program.*, 43(1), 2000.

[22] W. Xu and Z. M. Özsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, 2005.

APPENDIX

A. 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 $C = \{c_1, \dots c_n\}$ and $L = \{L_1, \dots L_k\}$. There are edges with capacity 1 from s to n nodes $c_1, c_2, \dots c_n$. There are also k nodes $L_1, \dots L_k$ 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 [10], 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 n (at least one $L(i)$ is not "covered"), tf-cover returns false. Otherwise (the max-flow is n), 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.

B. MULTI-TOKEN QUERIES

We explain here in detail how we compute the last-token descriptors (defined in Section 7.3.2) and the m-descriptors (defined in Section 7.3.4).

B.1 Computing last-token descriptors

For any rank k in p, by $cut(p, k)$ we denote the prefix of p having k main branch nodes.

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

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

1. For $f(X) \to l(X)[c_1(), \dots c_n(), .//d_1(), \dots, .//d_m()]$, add a descriptor $\mathbf{suff(f, l)}$ if we can infer that v_f contains the subtree of lt rooted at $\mathrm{OUT}(lt)$.

   if for a /-predicate (resp. //-predicate) P on $\mathrm{OUT}(lt)$ we have a descriptor $equiv(c_i, root_P)$ (resp. $equiv(d_j, root_P)$), add also $\mathbf{suff(f, l[P])}$.

2. For $f(X) \to l[c_1(), \dots c_n(), .//d_1(), \dots, .//d_m()]/g(X)$, given a descriptor $suff(g, p')$, add a descriptor $\mathbf{suff(f, l/p')}$ if, for $k = |\mathrm{MB}(q)| - |p'|$, we can infer that v_f contains the pattern $lt(k)$.

   if for a /-predicate (resp. //-predicate) P on $node_{lt}(k)$ we have a descriptor $equiv(c_i, root_P)$ (resp. $equiv(d_j, root_P)$), add also $\mathbf{suff(f, l[P]/}\mathrm{MB}(p')\mathbf{)}$.

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

1. For $f(X) \to l[c_1(), \dots c_n(), .//d_1(), \dots, .//d_m()]//g(X)$, given a suffix descriptor $suff(g, p)$, add a full suffix descriptor $\mathbf{fsuff(f, p, k)}$ 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 $f(X) \to l[c_1(), \dots c_n(), .//d_1(), \dots, .//d_m()]/g(X)$, given a full-suffix descriptor $fsuff(g, p, k')$, add a descriptor $\mathbf{fsuff(f, p, k)}$, for $k = k' - 1$, if we can infer that v_f contains the pattern $q(k)$.

3. For $f(X) \to l[c_1(), \dots c_n(), .//d_1(), \dots, .//d_m()]//g(X)$, given a full-suffix descriptor $fsuff(g, p, k')$, add a descriptor $\mathbf{fsuff(f, p, k)}$ for each rank $k < k'$ s.t. we can infer that v_f contains the pattern $q(k)$.

4. Whenever a descriptor $fsuff(f, p, 1)$ is obtained, for $f = S$, add $\mathbf{lt(S, p)}$ to the set of view descriptors.

B.2 Computing m-descriptors

Step 4 **of** findDescSupp(q, $\mathcal{P}$)**.** Apply the the following steps:

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

1. For rules $f(X) \to l(X)[c_1(), \dots c_n(), .//d_1(), \dots, .//d_m()]$, if we can infer that v_f contains the subtree of q rooted at $\mathrm{OUT}(q)$, add a descriptor $\mathbf{bm[f, |}\mathrm{MB}(q)\mathbf{|, (|}\mathrm{MB}(q)\mathbf{| + 1, l)]}$.

2. For $f(X) \to l[\dots]/g(X)$ and 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 descriptor $\mathbf{bm[f, k'_1 - 1, (k'_2, l/p')]}$.

3. For $f(X) \to l[\dots]//g(X)$, 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 $\mathrm{MB}(lt)_m$ s.t. the token p' has a mapping into $\mathrm{MB}(lt)_m$ starting at k_2 and ending *above* k'_2, where if $k'_2 = |\mathrm{MB}(q)| + 1$ above means at $k'_2 - 1$; if no such rank is found, set k_2 to 0.

   add a below m-descriptor $\mathbf{bm[f, k_1, (k_2, l)]}$.

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

1. For $f(X) \to l[\dots]//g(X)$, 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 $\mathrm{MB}(lt)_m$

   if we can infer that v'_f is equivalent with m's suffix of size 1, add the partial m-descriptor $\mathbf{pm(f, 1)}$.

2. For either $f(X) \to l[\dots]/g(X)$ or $f(X) \to l[\dots]//g(X)$ given a partial m-descriptor $pm(g, n')$: if we can infer that the query $cut(v_f, n' + 1)$ is equivalent with the suffix of m of size $n = n' + 1$, add a descriptor $\mathbf{pm(f, n)}$.

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

1. From start fragment names f and either rules

   $f(X) \to l[\dots]/g(X)$ or $f(X) \to l[\dots]//g(X)$

   if we can infer that v'_f root-maps into ft, add a descriptor $\mathbf{am[g, 1, (0, l)]}$ (respectively $\mathbf{am[g, 1, (1, -)]}$).

2. For $f(X) \to l[\dots]/g(X)$ and a descriptor $am[f, k'_1, (k'_2, p')]$:
   - *if p' is not '-':* if we can infer that v'_f root-maps in $ft(k'_1 + 1)$, add a descriptor $\mathbf{am[g, k'_1 + 1, (k'_2, p'/l)]}$
   - *if p' is '-':* for each rank k_1, $k_1 > k'_1$, s.t. we can infer that v'_f root-maps in the pattern $ft(k_1)$,add a descriptor $\mathbf{am[g, k_1, (k'_2, l)]}$.

3. For $f(X) \to l[\dots]//g(X)$, a descriptor $am[f, k'_1, (k'_2, p')]$:
   - *if p' is not '-':* if we can infer that v'_f root-maps in the pattern $ft(k'_1 + 1)$, add a descriptor

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

     for the highest rank k_2, $k'_2 < k_2 \le |\mathrm{MB}(ft)_m|$, s.t. p'/l has a mapping into $\mathrm{MB}(ft)_m$ 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 $|\mathrm{MB}(ft)_m| + 1$.
   - *if p' is '-':* for each rank k_1, $k_1 > k'_1$, s.t. v'_f root-maps in the pattern $ft(k_1)$, add

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

     for the highest rank k_2, $k'_2 < k_2 \le |\mathrm{MB}(ft)_m|$, s.t. the token l has a mapping into $\mathrm{MB}(ft)_m$ at rank k_2; if no such mapping exists set k_2 to $|\mathrm{MB}(ft)_m| + 1$.

D. Finally, for a fragment name g, given both

- a partial m-descriptor $pm(g, |\mathrm{MB}(m)|)$
- an above m-descriptor $am[g, k_1, (|\mathrm{MB}(ft)_m| + 1, -)]$,

add a descriptor $\mathbf{m(S)}$ to the set of view descriptors.

C. SINGLE-TOKEN QUERIES

We first give the proof of NP-hardness for support when q is a single-token XP_{es} query. Then, we describe a sound, tractable algorithm for this problem, and we show how it can be extended to an exponential-time sound and complete algorithm for this problem.

Proof of Theorem 8.1

PROOF. We use a reduction from the minimum set-cover problem [12]. Let $(\mathcal{U}, \mathcal{S}, k)$ be an instance of this problem, with $\mathcal{U} = \{e_1, \dots, e_n\}$ denoting the universe, $\mathcal{S} = \{S_1, \dots S_m\}$ 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 $b_p b_{p-1} \dots b_0$ be the binary representation of k.

We build the following instance of the support problem. We define the QSS as follows:

- the tree fragments $F = \{S, set, f, g, f_s, f_p^p, f_p^{p-1}, \dots, f_p^0, f_{p-1}^{p-1}, f_{p-1}^{p-2}, \dots, f_{p-1}^0, \dots, f_1^1, f_1^0, f_0^0\}$
- the alphabet $\Sigma = \{root, out, a, b, e_1, \dots, e_m\}$
- S is the start fragment
- the set of productions P defined as follows:
  1. the start productions
     $$S(X) \rightarrow root/f(X)$$
     and
     $$S(X) \rightarrow root//g(X)$$
  2. $f(X) \rightarrow a/a/a[Q]/a/out(X)$
     Where Q denotes the pattern obtained as follows: let L_i, $i = \overline{0,p}$, denote the pattern formed by $i+2$ nodes labeled b followed by the predicate $[e_1, \dots, e_n]$. For example, $L_0 = b[b[e_1, \dots, e_n]]$.
     Then, we define Q as
     $$Q = b[L_p][L_{p-1}] \dots [L_1][L_0].$$
  3. $g(X) \rightarrow a[b[//e_1, \dots, //e_n]]/a[f_s]/a//out(X)$
  4. $f_s \rightarrow b[f_p^p, f_{p-1}^{p-1} \dots, f_1^1, f_0^0]$
  5. $\forall i = \overline{0,p}$ s.t. $b_i = 1$, we have the productions
     $$f_i^k \rightarrow b[f_i^{k-1}][f_i^{k-1}] \forall k = \overline{0,i}$$
     $$f_i^0 \rightarrow b[set]$$
  6. $\forall i = \overline{0,p}$ s.t. $b_i = 0$, we have the production
     $$f_i^i \rightarrow ()$$
  7. $\forall S_j = \{e_{l_1}, \dots, e_{l_j}\} \in S$ we have the production
     $$set \rightarrow b[e_{l_1}, \dots, e_{l_j}]$$
  8. finally we have the production
     $$set \rightarrow ()$$

This QSS produces two families of views. The first one, by the fragment name f, contains only one member (v_f), which has a main branch $root/a/a/a/a/out$, and a Q predicate on the third a-node.

The views from the second family of views, by the fragment name g, have the main branch $root//a/a/a//out$, the predicate $b[//e_1, //e_2, \dots, //e_n]$ 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 elements $e_1, \dots, e_n$.

Let now q be the single-token XP_{es} query

$$q = root/a/a[b[//e_1, //e_2, \dots, //e_n]]/a[Q]/a/out.$$

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

$$root/a/a[b[//e_1, //e_2, \dots, //e_n]][\dots]/a[Q][Q'][\dots]/a/out$$

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

$$root/a[b[//e_1, //e_2, \dots, //e_n]]/a[Q'][\dots]/a[Q]/a/out$$

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 $b[//e_1, //e_2, \dots, //e_n]$ can map. But this is possible if and only if all the elements $e_1, \dots, e_n$ are present in Q', and this happens if and only if there exists a cover of maximal size k for the set $\mathcal{U}$. □

We now discuss the sound, tractable algorithm for support in this setting. For presentation simplicity, we first limit the discussion to rewrite plans that are intersections of views (no compensation before the intersection step). We start by considering how one can check the existence of a rewriting using a finite set of explicitly listed views.

We introduce first some new notation. For a pattern p and a node $n \in \text{MBN}(p)$, by $\text{SP}_p(n)$ we denote the subtree rooted at n in p.

For the to-be-rewritten query q and a predicate P in q, for a view v let N_v^P denote the set of main branch nodes of v having a predicate P' s.t. (i) the pattern of P' is equivalent to some subtree of q, and (ii) the pattern q_P contains the pattern $v_{P'}$.

We are now ready to formulate a sound, tractable algorithm which, for a set of explicitly listed views $\mathcal{V}$, tests if an equivalent rewriting exists for a single-token query q.

The tests will ensure the opposite containment mapping, i.e., that q contains each possible interleaving of the views. Section C.1 will then detail how one can verify these properties even when abstracting away from the view definitions, by using view descriptors, and in Section C.2 we show how descriptors can be inferred even when views are defined as the expansions of a QSS.

For a view v_j verifying the conditions of lines 7-9, we say that v_j *contributes* P in the intersection. We can prove the following:

THEOREM C.1. *For an XP single-token query q and a set of XP views $\mathcal{V}$ containing q, **testEquiv** is a sound PTIME procedure for $q \equiv \cap\mathcal{V}$.*

PROOF SKETCH. When q has only /-edges in the main branch, among the views of $\mathcal{V}$ there must be at least one having the same

Algorithm 3 testEquiv($\mathcal{V}$, q)

```
1:  begin
2:  for each predicate $P$ in $q$ and each view $v_j \in \mathcal{V}$
3:      compute the set of nodes $N^P_{v_j}$
4:  if there exists a view $v_i \in \mathcal{V}$ s.t. $\text{MB}(v_i) \equiv \text{MB}(q)$ then
5:      if for all predicates $P$ in $q$
6:          there exists a view $v_j \in \mathcal{V}$ s.t.
7:              for all containment mappings $\psi$ from $\text{MB}(v_j)$ into $\text{MB}(q)$
8:                      the node $n_P$ is an image under $\psi$
9:                      the node $n \in v_j$ for which $\psi(n) = n_P$ is in $N^P_{v_j}$
10:     then output $\mathbf{true}$
11: end
```

main branch as q. With this, the intersection yields only interleavings having as main branch that linear path with /-edges only. The only remaining issue is whether all of q's predicates will be "present" at each interleaving, thus enabling a containment mapping from q. This follows from conditions (lines 7-9) which imply that, in any interleaving i, a predicate into which P can map will be present at i's main branch node of rank r_P.

Testing conditions (lines 7-9) for each predicate P in q can be easily translated into a containment mapping test. □

C.1 Descriptors

Again, the key idea for checking support when views are defined by a QSS is that, in order to test the conditions of the algorithm **testEquiv**, one does not need the detailed definitions of the views but only some particular details on them. This idea will be exploited by our *view descriptors*. More precisely, assuming we are dealing with expansions of a QSS $\mathcal{P}$ with start fragment name S,

For the condition of line 4. For a view v containing q, a view descriptor $\mathbf{mb(S)}$ will indicate that the pattern representing the main branch of v is equivalent with $\text{MB}(q)$.

For the conditions of lines 7-9. A direct but expensive (complexity-wise) solution for this part would be specify in a descriptor the set of predicates P the view contributes, according to lines $7-9$. But it suffices instead to consider predicates individually. By a descriptor $\mathbf{pred(S,P)}$ we denote a view v which contains q and contributes predicate P. The fact that a certain view may contribute several predicates is irrelevant, as it is enough to know that there exist covering views for each of them (not necessarily distinct).

Note that descriptors partition the set of views into equivalence classes with respect to the tests of **testEquiv**: two views having different definitions but yielding the same descriptors will be equally useful for these tests.

Now, we can easily rephrase **testEquiv** into an algorithm that runs directly on the set of view descriptors D, instead of the explicit views $\mathcal{V}$ to which they correspond.

Algorithm 4 testEquivDesc(D, q)

```
1: if there exists a descriptor $\mathbf{mb(S)} \in D$ then
2:     if for all predicates $P$ in $q$
3:         there exists a descriptor $\mathbf{pred(S,P)} \in D$
4:     then output $\mathbf{true}$
5: end
```

We can prove the following:

THEOREM C.2. *For a single-token query q, a finite set of views $\mathcal{V}$ containing q and their corresponding descriptors D,* **testEquiv***(q,V) outputs* true *if and only if* **testEquivDesc***(q,D) does so.*

C.2 Descriptors from a QSS program

We present in this section a bottom-up algorithm that computes, for a QSS and a single-token query q, all the view descriptors for all the QSS expansions.

As before, we need fragment descriptors in order to perform incrementally the tests of **testSupp**. While this is straightforward for the test of line 4, the conditions of lines $7-9$ require special treatment. More precisely, given a view v_j and a query q, we rephrase this part into testing the non-existence of a containment mapping (with a twist) between $\text{MB}(v_j)$ and $\text{MB}(q)$ (i.e., linear patterns). Note that this is something we already know how to do from the multi-token case. For that, since there is always at least one containment mapping from $\text{MB}(v_j)$ into $\text{MB}(q)$, we will change some of the labels in $\text{MB}(v_j)$ and $\text{MB}(q)$ in order to test if there are mappings violating the conditions of lines $7-9$.

Assuming that the set of nodes $N^P_{v_j}$ is not empty, let l_P denote the label of n_P and let l'_P denote a new label derived from it. For this new label, let us consider the following variation to the usual definition of a mapping: *a node of label l'_P can map into a node of label l_P*. This variation will be called hereafter *P-mapping*. Based on this, containment P-mapping is then defined in the usual way. Using this new label, let $relabel(P, v_j)$ denote the pattern obtained from v_j by relabeling with l'_P the nodes that were labeled l_P in v_j and were not in the set $N^P_{v_j}$. Also, let q' denote the pattern obtained from q by relabeling the node n_P by l'_P.

It is now easy to see that testing the conditions of lines $7-9$ amount to testing the non-existence of a containment P-mapping from $\text{MB}(relabel(P, v_j))$ into $\text{MB}(q')$. And this can be done on an explicit view v_j by the same bottom-up approach, which advances one token at a time, described in Section 7.3.3.

For each predicate P, we describe a subroutine that finds the descriptors $\mathbf{pred(P)}$. The $\mathbf{mb}()$ one will be obtained as a side-effect of these subroutines. We assume that the *map*, *equiv*, *contain* descriptors are already pre-computed. Besides those, we will use only one kind of fragment descriptors, called *intermediary descriptors*.

DEFINITION C.1. Syntax: *An intermediary descriptor w.r.t. a fixed predicate P in q for a unary fragment name f is a tuple $interm[f, k_1, (k_2, p)]$, where k_1 and k_2 are ranks in the main branch of q, and p denotes any linear substring of* $\text{MB}(q)$ *or* $\text{MB}(q')$.

Semantics: *Such a descriptor says that*

- *v_f contains the subtree of q rooted at the main branch node of rank k_1,*
- *p is the main branch of the first token of $relabel(P, v_f)$,*
- *k_2 is the start rank (i.e., the upmost node) of the* lowest possible output P-mapping image *of the rest of the main branch of $relabel(P, v_f)$ (besides t) into* $\text{MB}(q')$*; by convention, k_2 is* $|\text{MB}(q)| + 1$ *when v_f has only one token (the one described by t) and is* 0 *when there is no such P-mapping.*

Algorithm findDescriptors:

For each predicate P in q, repeat until fix-point:

1. for rules $f(X) \rightarrow l(X)[c_1(), \ldots c_n, .//d_1(), \ldots, .//d_m()]$, if we can infer that v_f contains the subtree of q rooted at $\text{OUT}(q)$, add a descriptor $\mathbf{interm}[\mathbf{f}, |\text{MB}(q)|, (|\text{MB}(q) + \mathbf{1}|, \mathbf{l})]$.
2. for rules $f(X) \rightarrow l[c_1(), \ldots c_n, .//d_1(), \ldots, .//d_m()]/g(X)$, given a descriptor $interm[g, k'_1, (k'_2, p')]$, for $k_1 = k'_1 - 1$, if we can infer that v_f contains the subtree $q(k_1)$, then
   (a) if $l \neq l_P$, add a descriptor $\mathbf{interm}[\mathbf{f}, \mathbf{k_1}, (\mathbf{k'_2}, \mathbf{l/p'})]$,
   (b) if $l = l_P$: add a descriptor $\mathbf{interm}[\mathbf{f}, \mathbf{k_1}, (\mathbf{k'_2}, \mathbf{l_P/p'}]$
   (i) if P is a /-predicate and there exists a descriptor $equiv(c_i, n)$ s.t.

- the pattern q_P contains the pattern $l/xpath(\mathrm{SP}_q(n))$,
- the pattern $\mathrm{SP}_q(n)$ root-maps in the subtree of q rooted at some /-child of $node_q(k_1)$,

or (ii) if P is a //-predicate and there is a descriptor $equiv(d_j, n)$ s.t.

- the pattern q_P contains the pattern $l//xpath(\mathrm{SP}_q(n))$,
- the pattern $\mathrm{SP}_q(n)$ root-maps in the subtree of q rooted at some descendant of $node_q(k_1)$.

(c) otherwise, add the descriptor $\mathbf{interm[f, k_1, (k_2', l_P'/p')]}$.

If f is the start fragment name S, if $k_1' - 1 = 1$ and if we can infer that v_f contains q,

(a) if $k_2' = 0$ or the token l/p' does not have a P-mapping image in $\mathrm{MB}(q')$ starting at $\mathrm{ROOT}(q')$ and ending *above* k_2', where if $k_2' = |\mathrm{MB}(q)| + 1$ above means at $k_2' - 1$, add a view descriptor $\mathbf{pred(P)}$

(b) moreover, if $|l/p'| = |\mathrm{MB}(q)|$ (which here means $\mathrm{MB}(v_f) \equiv \mathrm{MB}(q)$), add also the descriptor $\mathbf{mb(S)}$ to the set of view descriptors.

3. for $f(X) \to l[c_1(), \dots c_n, .//d_1(), \dots, .//d_m()]//g(X)$, given a descriptor $interm[g, k_1', (k_2', p')]$, for each rank k_1, $1 \leq k_1 < k_1'$, s.t. we can infer that v_f contains $q(k_1)$,

   find also the lowest possible rank k_2, $1 \leq k_2 < k_2'$, such that the token p' P-maps into $\mathrm{MB}(q')$ starting at k_2 and ending *above* k_2', where if $k_2' = |\mathrm{MB}(q)| + 1$ above means at $k_2' - 1$; if no such rank is found, set k_2 to 0.

(a) if $l \neq l_P$, add a descriptors $\mathbf{interm[f, k_1, (k_2, l)]}$,

(b) if $l = l_P$: add the descriptor $\mathbf{interm[f, k_1, (k_2', l_P]}$

(i) if P is a /-predicate and there exists a descriptor $equiv(c_i, n)$ s.t.

- the pattern q_P contains the pattern $l/xpath(\mathrm{SP}_q(n))$,
- the pattern $\mathrm{SP}_q(n)$ root-maps in the subtree of q rooted at some /-child of $node_q(k_1)$,

or (ii) if P is a //-predicate and there is a descriptor $equiv(d_j, n)$ s.t.

- the pattern q_P contains the pattern $l//xpath(\mathrm{SP}_q(n))$,
- the pattern $\mathrm{SP}_q(n)$ root-maps in the subtree of q rooted at some descendant of $node_q(k_1)$.

(c) otherwise, add the descriptor $\mathbf{interm[f, k_1, (k_2', l_P')]}$.

When $f = S$, if the token l does not P-map into $\mathrm{MB}(q')$ starting at rank 1 and ending above k_2 (i.e., $k_2 = 0$ or $k_2 = 1$), add a descriptor $\mathbf{pred(S, P)}$ to the set of view descriptors.

THEOREM C.3. *Given a QSS $\mathcal{P}$ and a single-token query q, algorithm* ***findDescriptors*** *is sound and complete for computing the descriptors for $\mathcal{P}$'s expansions. It runs in polynomial time in the size of the query and of the QSS.*

C.3 Decision procedure for support

We describe in this section a sound and complete algorithm for support on single-token XP_{es} queries, which will run in exponential time in the size of the query. For this, we will relax the definition of the N_v^P set as follows: for the to-be-rewritten query q and a predicate P in q, for a view v, N_v^P will now denote the set of main branch nodes of v having a predicate P' s.t. the pattern q_P contains the pattern $v_{P'}$.

With this adjustment, let $testEquiv'$ denote the corresponding procedure. We can prove the following:

THEOREM C.4. *For an XP_{es} single-token query q and a set of XP views $\mathcal{V}$ containing q,* ***testEquiv*** *is a sound and complete procedure for $q \equiv \cap\mathcal{V}$.*

Then, the approach based on view descriptors instead of view definitions remains the same. Regarding support, in order to access the necessary details w.r.t N_v^P, we will introduce a new kind a fragment descriptor, which will record what nodes of q map in an expansion of a tree fragment name. More precisely,

DEFINITION C.2. *For a fragment name f, a set C of /-siblings in q, a set D of //-siblings in q, and a node n from q, a* q-mapping descriptor *denotes a tuple $q - map(f, C, D, n)$. It says that (i) there exists an expansion v_f which root-maps in the subtree of q rooted at n, (ii) for each $n_i \in C$, there exists a root-mapping m_i of the subtree of q rooted at n_i into v_f, and (iii) for each $n_j \in D$, there exists a mapping m_j of the subtree of q rooted at n_j into v_f.*

Any of the three components C, D or n might be empty.

Note that the space of $q - map$ descriptors is exponential in the size of q. They can be computed, in worst-case exponential time, in bottom-up manner straightforwardly.

With these descriptors, we modify **findDescriptors** as follows:

- For the steps (2.b.i) and (3.b.i): if there is a descriptor $q - map(c_i, C, D, n)$ s.t. $root_P \in C$ and n is a child of $node_q(k_1)$.
- For the step (2.b.ii) and (3.b.ii): if there is a descriptor $q - map(d_j, C, D, n)$ s.t. $root_P \in D$ and n is a descendant of $node_q(k_1)$.

findDescriptors is sound and complete for computing the descriptors of the QSS expansions. It thus enables a sound and complete algorithm for support (via *testEquivDesc*).

Dealing with compensated views. We use the same $comp(\mathcal{P}, q)$ construction to deal with plans that intersect compensated views.

D. SUPPORT IN $XP^{\cap,c}$

We sketch in this section an EXPTIME sound and complete procedure for support on XP_{es} queries in $XP^{\cap,c}$.

We first introduce some additional notation. A *lossless prefix* p of q is any pattern obtained q by "moving up" the output mark, i.e., setting as the output node any main branch node and interpreting what follows that node as predicate (side) branches.

We can prove the following:

THEOREM D.1. *An XP_{es} query q is supported by a QSS in $XP^{\cap,c}$ iff some lossless prefix of q is supported by that QSS in $XP^{\cap}$.*

This enables the following EXPTIME decision procedure for support in $XP^{\cap,c}$: test $XP^{\cap}$ support for each lossless prefix, using either the PTIME decision procedure of Section 7 (if the prefix is multi-token), or the EXPTIME one of the previous section (if the prefix is single-token).

E. EXTENSION TO QSS^+

Proof of Theorem 10.2, first part.

PROOF. We start with expressibility. We detail our proof for boolean tree patterns. The one for patterns of arity 1 is similar.

We use a reduction from the minimum set-cover problem [12]. Let $(\mathcal{U}, \mathcal{S}, k)$ be an instance of this problem, with $\mathcal{U} = \{e_1, \dots, e_n\}$ denoting the universe, $\mathcal{S} = \{S_1, \dots S_m\}$ 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 $b_p b_{p-1} \dots b_0$ be the binary representation of k.

We build the following instance of the expressibility problem. We define the QSS as follows:

- the tree fragments
$F = \{S, set, f_p^p, f_p^{p-1}, \dots, f_p^0, f_{p-1}^{p-1}, f_{p-1}^{p-2}, \dots, f_{p-1}^0, \dots, f_1^1, f_1^0, f_0^0\}$
- the alphabet $\Sigma = \{a, e_1, \dots, e_m\}$
- S is the start fragment
- the set of productions P defined as follows:
  - the start production
  $$S \rightarrow a(X)[f_p^p, f_{p-1}^{p-1} \dots, f_1^1, f_0^0]$$
  - $\forall i = \overline{0, p}$ s.t. $b_i = 1$, we have the productions
  $$f_i^k \rightarrow f_i^{k-1}, f_i^{k-1}, \forall k = \overline{0, i}$$
  $$f_i^0 \rightarrow set$$
  - $\forall i = \overline{0, p}$ s.t. $b_i = 0$, we have the production
  $$f_i^i \rightarrow ()$$
  - $\forall S_j = \{e_{l_1}, \dots, e_{l_j}\} \in S$ we have the production
  $$set \rightarrow e_{l_1}, \dots, e_{l_j}$$
  - finally we have the production
  $$set \rightarrow ()$$

The QSS grammar builds boolean queries having a root node labeled a with children nodes having 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 $a[e_1][e_2] \dots [e_n]$.

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, $e_1, \dots, e_n$. Hence expressibility holds if and only if we can find in S a cover of $\mathcal{U}$ of maximal size k. □

Proof of Theorem 10.2, second part.

PROOF. For support, we adapt the previous reduction from the minimum set-cover problem as follows:

The start production is now

$$S(X) \rightarrow b//a[f_p^p][f_{p-1}^{p-1}] \dots [f_1^1][f_0^0]//c(X),$$

while fragment names expand as before. This program generates views having a main branch $b//a//c$ and having various $[e_i]$ predicates on the a node.

The query q for which we want to test support is

$$b//a[e1] \dots [e_n]//c.$$

It is easy to see that all the views contain q, hence q is supported if and only if the intersection of *all* the generated views is equivalent to q. By using results from [7], this is the case if and only if one of the views, call it v, contains all others views. When this condition holds, v is equivalent to q if and only if its a node has all the predicates $[e_1], \dots [e_n]$. But one such view exists if and only if there exists a cover of maximal size k. □

E.1 Expressibility for general QSS

We show that the NP lower bound for expressibility is tight for practical purposes, since expressibility can be decided in exponential time. We can prove the following:

THEOREM E.1. *Expressibility can be decided in exponential time in the size of the query and of the views.*

We discuss the approach for deciding QSS expressibility (and support) for boolean patterns. Dealing with tree patterns by the same algorithm can be then done in the style of Section 4.2.

We now use fragment descriptors defined as follows:

DEFINITION E.1. *For a fragment name f and a set N of nodes from q, a* mapping descriptor *for the pair (f, N) (written map(f, N)) says that there exists an expansion v_f (which can be a tree or a forest) and a mapping m of v_f into q such that the roots of trees in v_f are mapped into the nodes in N.*

DEFINITION E.2. *For a fragment name f, a node $n \in q$ and a set N of same-edge sibling nodes children of n, an* equivalence descriptor *for the pair (f, N) (written equiv(f, N)) says that there exists an expansion v_f (a tree or a forest) satisfying the following:*

- *some $|N|$ trees among those in the forest v_f are equivalent to the subtrees rooted at the nodes of N in q,*
- *if the nodes N are connected by a $/$-edge to n, the remaining trees in v_f map into q such that the images of their roots are either among the nodes of N or among other /-siblings of the nodes of N,*
- *otherwise (i.e. the nodes N are connected by a $//$-edge to n), the remaining trees in v_f map into q such that the images of their roots map below n in q.*

Note that in these descriptors we do not bookkeep the number of trees in the expansion v_f. We only keep track of the subtrees of q into which they can map (for mapping descriptors) or of the sibling subtrees in q with which their are equivalent (for equivalence descriptors). Also note that an equivalence descriptor implies a mapping one, for the same fragment name and set of nodes.

The space of distinct descriptors is O$(|\mathcal{G}| \times 2^{|q|})$, hence exponential in the size of the query and polynomial in the size of the program.

E.2 Computing descriptors

For a given query q and a QSS $\mathcal{G}$, we can compute all the corresponding descriptors as described below. The computation starts from the productions with no tree fragment nodes and continues inferring descriptors until a fixed point is reached, close in spirit to bottom-up parsing as in the CYK algorithm [13] or to bottom-up Datalog evaluation [2]. Each step of this process will run in time worst-case exponential in the size of q and $\mathcal{G}$.

Algorithm findDescriptors:

1. start with an empty set of descriptors D.
2. for each production $f \rightarrow ()$, add to D all the descriptors of *map*$(f, \{n\})$, for $n \in q$.
3. for each tree production $f \rightarrow tf$, such that *tf* has only element nodes, compute (by the definitions) and add to D all the possible descriptors *map*$(f, \{n_q\})$ and *equiv*$(f, \{n_q\})$.
4. for each tree production $f \rightarrow tf$:

(a) infer new mapping descriptors for f as follows:
For $f_1, \dots f_k$ being the tree fragment nodes appearing in tf, for all possible combinations of existing mapping descriptors, $c = (\mathit{map}(f_1, N_1), \dots, \mathit{map}(f_1, N_1))$, let tf_c denote the tree pattern obtained from tf by replacing each tree fragment node f_i by a set of trees that are isomorphic copies of the subtrees of q rooted at the nodes listed in N_i.
For each mapping ψ of tf_c into q, add to D the descriptor $\mathit{map}(f, \{\psi(\text{ROOT}(tf_c)\})$. Note that by a naive iteration over the space of descriptors this step can be executed in time exponential in the number of tree fragment nodes, k. But a polynomial time approach is possible by dealing with the mapping descriptors in bulk (similar to how mappings algorithms work).

(b) infer new equivalence descriptors for f as follows:
Let $c_1, \dots, c_n$ be the /-children of $\text{ROOT}(tf)$ and let $d_1, \dots d_m$ be its //-children (either list can be empty). Using their associated descriptors, build a new descriptor $\mathit{equiv}(f, \{n_q\})$ (and implicitly $\mathit{map}(f, \{n_q\})$) for each node $n_q \in q$ such that:

i. there exist some fragment names $C = \{c_{i_1}, \dots c_{i_j}\} \subseteq \{c_1, \dots, c_n\}$ and equivalence descriptors
$$\mathit{equiv}(c_{i_1}, N_{i_1}), \dots, \mathit{equiv}(c_{i_j}, N_{i_j}),$$
such that the set $N_/$ of /-children of n_q satisfies
$$N_/ = N_{i_1} \cup \dots \cup N_{i_j},$$

ii. there exist some fragment names $D = \{d_{i_1}, \dots d_{i_j}\} \subseteq \{d_1, \dots, d_m\}$, and equivalence descriptors
$$\mathit{equiv}(d_{i_1}, N_{i_1}), \dots, \mathit{equiv}(d_{i_j}, N_{i_j}),$$
such that the set $N_{//}$ of //-children of n_q satisfies
$$N_{//} = N_{i_1} \cup \dots \cup N_{i_j},$$

iii. all remaining fragment names $c_i \notin C$ have mapping descriptors of the form $\mathit{map}(c_i, N_q^i)$, for N_q^i being a set of /-children of n_q

iv. all remaining fragment names $d_j \notin D$ have mapping descriptors of the form $\mathit{map}(d_j, N_q^j)$, for N_q^j being a set of descendants of n_q.

The above step can be done by iterating over the descriptors of the cs and ds. Hence this step can be completed in worst-case exponential time in the size of the query and program.

5. for each forest production $f \to g_1, \dots g_k$:

(a) for any combination of mapping descriptors
$$\mathit{map}(g_1, N_1), \dots \mathit{map}(g_k, N_k)$$
add to D the descriptor
$$\mathit{map}(f, N_1 \cup \dots \cup N_k).$$

(b) for any combination of descriptors
$$c = \mathit{equiv}(g_{i_1}, N_{i_1}), \dots \mathit{equiv}(g_{i_l}, N_{i_l}), \\ \mathit{map}(g_{j_{l+1}}, N_{j_{l+1}}), \dots, \mathit{map}(g_{i_k}, N_{i_k})$$
for $(i_1, \dots, i_l, i_{l+1}, \dots, i_k)$ being any size k permutation, add to D the descriptors
$$\mathit{equiv}(f, N_{i_1} \cup \dots \cup N_{i_l})$$
$$\mathit{map}(f, N_{i_1} \cup \dots \cup N_{i_l})$$
when one of the following conditions is verified:

i. all the nodes in $N_{i_1}, \dots, N_{i_l}, N_{i_{l+1}}, \dots, N_{i_k}$ are /-siblings in q

ii. all the nodes in $N_{i_1}, \dots, N_{i_l}$ are //-siblings in q, children of some node n_q, while the nodes in the sets $N_{i_{l+1}}, \dots, N_{i_k}$ are all somewhere below n_q in q.

This step can executed in time worst-case exponential in the size of the query and program.

6. if any new descriptors have been inferred, go back to step 4.

We can prove the following:

THEOREM E.2. *Expressibility holds iff* ***findDescriptors*** *outputs an equivalence descriptor for the start tree fragment, of the form* $\mathit{equiv}(S, \text{ROOT}(q))$. ***findDescriptors*** *runs in exponential time in the size of the QSS and of the query.*

F. THE REWRITE ALGORITHM FOR MULTI-TOKEN QUERIES IS SOUND

Proof of Theorem 7.1:

PROOF SKETCH. The first condition 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, i.e., q is of the form $ft//lt$.

In this case, condition (line 7) 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 $l_q \not\sqsubseteq \text{MB}(v_i)$, 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 $\text{MB}(lt)$) 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 $\text{MB}(lt)$ (resp. $\text{MB}(ft)$) cannot be collapsed with the part of $\text{MB}(m')$ to which it is isomorphic (by the definition of $\text{MB}(ft)_m$ and $\text{MB}(lt)_m$). 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 7, 9- 13) in polynomial time. For (line 7), 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:

- finding the views that have a sub-query equivalent to m (an equivalence test) and, for each of them,
- checking the non-existence of the two mappings (even though $prefix_j$ root-maps into ft, hence $\text{MB}(prefix_j)$ also root-maps into $\text{MB}(ft)$, and $suffix_j$ output-maps into lt, hence $\text{MB}(suffix_j)$ also output-maps into $\text{MB}(ft)$).

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 $\text{MB}(suffix_j)$) respectively top-down (in the case of $\text{MB}(prefix_j)$) procedure as the one described in Section 7.3, advancing one token at a time. □

Proof of Theorem 7.2:

PROOF. The proof is essentially a reformulation of the part of the proof of Theorem 4.5 from [7] that concerns multi-token queries from *XP*$_{es}$. The difference is that 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 that do not obey the *XP*$_{es}$ condition. $s(v)$ is called the extended skeleton of v.

First, in order to reuse the proof of Theorem 4.5 from [7] we show 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 $\cap_i s(v_i)$.

Then, if the conditions of line 5 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 2.1 and 2.2, would imply that q is not supported.

If q has only 2 tokens (i.e., the middle part m is empty), then $\text{MB}(q)$ is of the form $\text{MB}(ft)//\text{MB}(lt)$. If the test of line 7 does not hold, it means that l_q is non-empty 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 $\text{MB}(q)$, 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 non-empty), 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 (used in the proof of Theorem 4.5 from [7]).

LEMMA F.1. *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.

W.l.o.g. 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 F.1, 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 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. □

Proof of Theorem 7.3:

PROOF SKETCH. 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. □

Proof of Theorem 7.4:

PROOF SKETCH.

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 the **pref** descriptor. 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 inferrence of an **ft** descriptor 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.

Last-token descriptors. This case is symmetrical to the first-token descriptors.

l-descriptors. Partial l-descriptors are computed starting from the base case of rules that have no tree fragment names (Step 1). Then it records bottom up patterns and ranks in q and l_q, inferring partial l-descriptors that satisfy Definition 7.4. It 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, i.e., 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.

m-descriptors. 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 7.7. Finally, having a partial m-descriptor $pm(g, |\mathrm{MB}(m)|)$ and an above m-descriptor $am[g, k_1, (|\mathrm{MB}(ft)_m| + 1, -)]$ 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. □

Proof of Theorem 7.5:

PROOF SKETCH. 1. Soundness follows from the observation that follows after Theorem 7.4 and from the fact that $comp(\mathcal{P}, q))$ generates compensated views.

2. Follows from Theorem 7.2 and from the fact that $comp(\mathcal{P}, q))$ generates views having all the compensation that may be used by a mapping from q into an interleaving of views.

□

G. OTHER EXAMPLES

EXAMPLE G.1. ***Testing non-containment for linear patterns.*** *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 in Section 7.3 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 $|q_1|$, i.e. $k = 6$. 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.*

EXAMPLE G.2. ***First-token descriptors.*** *On our running example, i.e., query q_1 from Example 1.1 and the normalized QSS $\mathcal{P}$ from Example 3.2, we obtain first-token descriptors as follows:*
On the rule $f_0(X) \rightarrow doc(T)//f_1(X)$,

- *for q's main branch node at rank $k = 1$,*
- *we already have a fragment descriptor $contain(f_1, n_{v_3})$,*
- *we have no c_i, d_j branches,*
- *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* $\mathbf{ft}(\mathbf{doc}(\mathbf{T}), \mathbf{f_0})$.

EXAMPLE G.3. ***Last-token descriptors.*** *On our running example, we obtain last-token descriptors as follows:*

- *we compute* suffix *descriptors as follows:*
  - *on the rule$f_5(X) \rightarrow museum(X)$,*
  - *by the step 2.A.1 of* findDescSupp *we get the fragment descriptor $d_1 = suff(f_5, museum)$,*
- *we also have the mapping descriptors $d_2 = equiv(f_7, n_{g_2})$, $d_2' = map(f_4, n_{s_2})$, $d_2'' = equiv(f_4, n_{s_2})$*
- *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 $\mathbf{lt}(\mathbf{f_0}, \mathbf{museum})$.
- *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*
  $d_6 = suff(f_3, tour/museum)$, and (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
  $\mathbf{lt}(\mathbf{f_0}, \mathbf{trip}[\mathbf{schedule}//\mathbf{walk}]/\mathbf{museum})$.

EXAMPLE G.4. ***m-descriptors.*** *On our running example, we obtain m-descriptors as follows:*

- *we first obtain the below m-descriptor $bm(f_5, 6, (7, museum))$, as $|\mathrm{MB}(q)| = 6$.*
- *we then get to partial m-descriptors:*

  From the rule $f_2(X) \rightarrow trip[f_7()]/f_5(X)$, since $\mathrm{MB}(lt)_m$ 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 $|\mathrm{MB}(m)| = 3$.

- *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 $\mathrm{MB}(ft)_m$ is empty, $|\mathrm{MB}(ft)_m| + 1$ is indeed 1.
- *finally, using the two descriptors $pm(f_1, 3)$, $am(f_1, 1, (1, -))$, we obtain the view descriptor $\mathbf{m}(\mathbf{f_0})$.*

H. COMPILATION AND NORMALIZATION

We assume Query Set Specifications given as in [19], i.e. specifying tree pattern queries with one output. We preprocess an input QSS in two stages. In the first stage, we make the output explicit and in the second one we normalize the specification such that there are no ?, $*$ or $+$ occurrence constraints.

The result of the compilation stage (subsequently used during normalization) is a *QSS with bindings*, which is similar to a QSS except that instead of result node names it uses variables to specify the result node (in the style of Datalog for trees introduced in [1]). Tree fragment names on the left hand side of a rule may carry a variables bound in the tree fragments from the right hand side. A variable from a tree fragment can either be bound to an element node or to a tree fragment node. If it is an element node, then the variable bindings are given by the matches of that node into a document and that node will be a result node when it appears in an expansion. If it is a tree fragment node f, then its variable bindings come from rules in which f appears on the left hand side. We will also use a syntax of the form

$$f(X) \rightarrow tf_1(Y_1), \ldots tf_k(Y_k)$$

denoting that Y_j is empty or it is a variable bound somewhere in the tree fragment tf_j.

An *expansion* of a QSS with bindings is obtained in a similar way to one for a regular QSS, by replacing non-terminals with the body of the rule in which they appear and keeping the correspondence between variables from the left hand side and from the right hand side.

THEOREM H.1. *For any QSS $\mathcal{G}$ there is a QSS with bindings $\mathcal{G}$ and without ?, $*$ or $+$ occurrence constraints such that $\mathcal{G}$ produces the same expansions as $\mathcal{G}'$.*

PROOF. We show below the compilation and normalization steps that build $\mathcal{G}'$ from $\mathcal{G}$. □

H.1 Compilation

For the first stage, we start by inferring which tree fragments may contribute to the output and record this using '#' annotations/flags. A tree fragment that contains an element whose name is among the result node names, gets the # and so does the tree fragment name on the left hand side of that production. Then we propagate # flags, until reaching a fix point: if a tree fragment name has #, then all its right-hand-side occurrences in the QSS get # (together with the tree fragment in which they appear); the tree fragment names on the left of those rules also get a #. The number of #'s is at most the number of tree fragments plus the number of rules. And each # can be computed in linear time in the size of the rule in which that tree fragment appears.

Then we replace # flags with variables in the following way.

A rule $f \rightarrow g_1, \ldots, g_n$, in which f has no #, is compiled into $f() \rightarrow g_1(), \ldots, g_n()$.

For a rule $f \rightarrow g_1, \ldots, g_n$ in which f has # set, it is slightly more complicated.

Let $g_{i1}, \ldots, g_{ik}$ be the tree fragments on the right that have #, and contain a tree fragment node that has a #. For each g_{ip} in this set having t tree fragment nodes with #, enumerated in some fixed order, we add the rule

$$f(X_{pi}) \rightarrow g_1(), \ldots, g_{ip}(X_{pi}), \ldots, g_n(), \text{ where } 1 \leq i \leq t$$

where the X_{pi} variable is bound to the i^{th} tree fragment node of g_{ip} having a #.

Let $g_{j1}, \ldots, g_{jm}$ the tree fragments on the right that have # and also have result element nodes.

Then, for every g_{jp} having r result nodes, enumerated in some fixed order, we create r rules

$$f(X_{pi}) \rightarrow g_1(), \ldots, g_{jp}(X_{pi}), \ldots, g_n(), \text{ where } 1 \leq i \leq r$$

in which the variable X_{pi} is bound to the i^{th} result node of g_{jp}.

The number of rules in the grammar obtained through this compilation process may increase by at most the number of nodes in the original grammar. Each new production can be built in linear time in the size of the original production it comes from.

H.2 Normalization

At this stage we have a QSS with output specified by variables, such that any tree fragment with ? or $*$ occurrence constraint has no output variable. (Those with $+$ may have output.) We then rewrite this grammar into one without ?, $*$ or $+$. The transformation rules are given below:

- ? constraint: Any tree fragment node of the form $g()$? is replaced by a new fragment name $f()$ and the rules below are added:

  $$f() \rightarrow g() \quad (1)$$
  $$f() \rightarrow \quad (2)$$

  Any rule containing a $g(X)$? such that the X is the output of the rule is replaced by a rule in which $g(X)$ has no occurrence constraint.

- $*$ constraint: Any tree fragment node of the form $g()*$ is replaced by a new fragment name *recg*() and the rules below are added:

  $$\mathit{recg}() \rightarrow g(), \mathit{recg}() \quad (3)$$
  $$\mathit{recg}() \rightarrow \quad (4)$$

  Any rule containing a $g(X)*$ such that the X is the output of the rule is replaced by a rule in which $g(X)$ has no occurrence constraint.

- $+$ constraint:
  Any tree fragment node of the form $g(X)+$ is replaced by a new fragment name *recg*(X) and the rules below are added:

  $$\mathit{recg}(X) \rightarrow g(Y), \mathit{recg}(X) \quad (5)$$
  $$\mathit{recg}(X) \rightarrow g(X), \mathit{rech}() \quad (6)$$
  $$\mathit{recg}(X) \rightarrow g(X) \quad (7)$$
  $$\mathit{rech}() \rightarrow g(Y), \mathit{rech}() \quad (8)$$
  $$\mathit{rech}() \rightarrow g(Y) \quad (9)$$

  where $Y \cap X = \emptyset$ and *rech* is a new fragment name.

Please note that transformations 1-9 can be done in linear time in the size of the QSS. In the following, we will consider that our specification is already normalized, i.e. there are no occurrence constraints.