Rewriting Regular XPath Queries on XML Views

Wenfei Fan [1,2] Floris Geerts [1,3] Xibei Jia [1] Anastasios Kementsietsidis [1]
[1] University of Edinburgh [2] Bell Laboratories [3] Hasselt University/Transnational Univ. of Limburg
{wenfei,fgeerts,xjia,akements}@inf.ed.ac.uk

Abstract

We study the problem of answering queries posed on virtual *views of* XML *documents, a problem commonly encountered when enforcing* XML *access control and integrating data. We approach the problem by rewriting queries on views into equivalent queries on the underlying document, and thus avoid the overhead of view materialization and maintenance. We consider possibly recursively defined* XML *views and study the rewriting of both XPath and regular XPath queries. We show that while rewriting is* not *always possible for XPath over recursive views, it is for regular XPath; however, the rewritten query may be of* exponential *size. To avoid this prohibitive cost we propose a rewriting algorithm that characterizes rewritten queries as a new form of automata, and an efficient algorithm to evaluate the automaton-represented queries. These allow us to answer queries on views in linear time. We have fully implemented a prototype system,* SMOQE*, which yields the first regular XPath engine and a practical solution for answering queries over possibly recursively defined* XML *views.*

1. Introduction

In many applications users are allowed to access an XML document only by querying a view of the data. The need for this is evident in, for example, enforcing access control on XML data [2, 5, 9]. To prevent improper disclosure of sensitive or confidential information of XML data residing in a server, the server defines an XML view for each group of users, consisting of all and only the information that the users are authorized to access. While the users may query the view, they are not allowed to directly query or access the underlying document (referred to as the *source*). With this comes the need to answer queries posed on the views. One way to do this is to first materialize the views and then directly evaluate queries on the views. However, it is often too costly to materialize and maintain a large number of views, a common scenario when many groups of users with different access privileges query the same source. A more realistic approach is to *rewrite* (aka. translate, reformulate) queries on the views into equivalent queries on the source, evaluate the rewritten queries on the source *without materializing the views*, and return the answers to the users.

We study how to rewrite XML queries posed on virtual XML views into equivalent queries on the underlying XML document. For XML queries we start with a fragment of XPath, which supports recursion (the descendant-or-self axis '//'), union and complex filters (predicates). This class of XPath queries is commonly used in practice and is essential to XQuery, XSLT and XML Schema. We consider XML views defined by annotating a view DTD with a collection of (regular) XPath expressions, along the same lines as how commercial systems specify XML views [15, 21, 20]. An XML view defined as above is a mapping $\sigma : D \to D_V$ in the global-as-view style, from XML documents of the *document* DTD D to documents of the *view* DTD D_V. When the view schema D_V is *recursively defined*, i.e. if some element type in D_V is defined in terms of itself, so is the view. The central technical problem studied in this paper is:

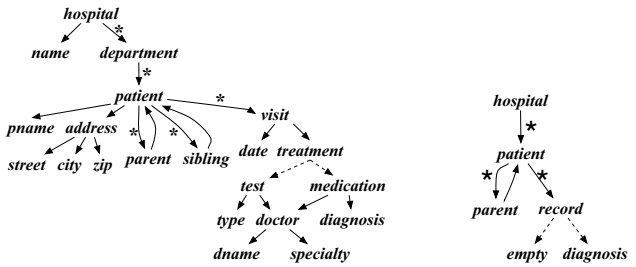
> The *rewriting problem* is to find an algorithm that, given a view definition σ and an XPath query Q over the view DTD D_V, computes an XPath query Q' over the document DTD D such that for any XML tree T of D, $Q(\sigma(T)) = Q'(T)$.

While there has been a host of work on rewriting XPath queries into SQL queries for XML views of relational data (see [17] for a survey), little previous work has considered rewriting XPath queries into XPath queries for XML views of XML data. In this context, query rewriting has only been studied for non-recursive XML views, over which XPath rewriting is always possible [9]. However, query rewriting for *recursive* views is still an *open* problem [17].

Recursive DTDs naturally arise when, e.g., specifying biomedical data (see the Gene Ontology database, GO [7]); in fact [3] shows that out of 60 real-world DTDs analyzed, more than half (35) of them were recursive. It is the reason that Oracle supports fully recursively defined XML views (AXSD [21]) and that IBM also allows a class of recursively defined XML view (DAD [15]). However desirable, the rewriting problem is more intriguing for recursively defined views, due to the interaction between recursion in XPath queries (e.g., '//') and recursion in the view definition.

Example 1.1: Consider a *hospital* DTD D shown as a graph in Fig. 1(a). A *hospital* document of D consists of a list of *department*s, and each *department* has a list of in-*patient*s (i.e. patients who are currently residing in the hospital; we

1-4244-0803-2/07/$20.00 ©2007 IEEE.



(a) document DTD D (b) view DTD D_V

production: *hospital* → *patient**
σ_0(*hospital,patient*) = *department/patient[visit/treatment/medication/diagnosis/text() = 'heart disease']* /*Q_1*/
production: *patient* → *parent**, *record**
σ_0(*patient, parent*) = *parent* /*Q_2*/
σ_0(*patient, record*) = *visit* /*Q_3*/
production: *parent* → *patient*
σ_0(*parent, patient*) = *patient* /*Q_4*/
production: *record* → *empty* + *diagnosis*
σ_0(*record, empty*) = *treatment/test* /*Q_5*/
σ_0(*record, diagnosis*) = *treatment/medication/diagnosis* /*Q_6*/

(c) view specification

Figure 1. Example: document and view DTDs and view specification.

use '$*$' on an edge to indicate a list). For each *patient*, the hospital maintains her name (*pname*), *address*, records of *visit*s, each including the visit *date* and *treatment* which is either a *test* or some *medication* (dashed edges indicate disjunction), as well as information about the treating *doctor*. Each *name*, *pname*, *street*, *city*, *zip*, *date*, *type*, *dname*, *specialty* has a single text node (PCDATA) as its child (omitted in the figure). The hospital also maintains family medical history by means of the recursively defined *parent* and *sibling*. It records the same information of ancestors with those of in-patients, by sharing the description for *patient*s.

A view σ_0 is defined for a research institute studying inherited patterns of heart disease, with the *view* DTD depicted in Fig. 1(b) (the view is defined in Example 2.2). Obliged by the Patient Privacy Act, the view reveals only those *patient*s who have heart disease, along with their *parent* hierarchy. While the institute may access *diagnosis* information of those patients and their ancestors, it is denied access to their *name*, *address*, *test* and *doctor* data.

Consider an XPath query Q posed on the view, which is to find patients whose ancestors also had heart disease:
Q: *patient*[*//record/diagnosis/text()*='heart disease'].
Here $*$ denotes a wildcard, i.e., any element. However, it is impossible to rewrite Q on the view to an equivalent query (in the XPath fragment mentioned above) on the underlying *hospital* document. This is because '//' in Q is supposed to traverse only the *parent* hierarchy on the view, i.e., a sequence of the *(parent/patient)* pattern; however, when translated to a query Q' on the source, Q' necessarily retains '//' since the view DTD is recursive, and '//' in Q' may access *sibling*s of those patients, although *sibling*s are not in the view and are not allowed to be accessed. An incorrect translation may lead to serious security breach. □

In response to this we develop both fundamental results and practical techniques for the rewriting problem. The main contributions of the paper include the following.

1. Closure Properties. On the theoretical side, we study the *closure property of XPath under query rewriting*: is it always possible to rewrite XPath queries on views to XPath queries on the source? We prove that XPath is *not* closed under query rewriting for *recursive* views. In light of this we consider a mild extension of XPath, *regular XPath [19]*, which uses the general Kleene closure E^* instead of the '//' axis. We show that regular XPath is *closed* under rewriting for arbitrary views, recursive or not. Since regular XPath subsumes XPath, any XPath queries on views can be rewritten to equivalent regular XPath queries on the source.

However, the rewriting problem is *EXPTIME-complete*: for a (regular) XPath query Q over even a (non-)recursive view, the rewritten regular XPath query on the source may be inherently *exponential in the size of Q and the view* DTD D_V. This tells us that rewriting is beyond reach in practice if Q is *directly* rewritten into regular XPath.

On the practical side, to avoid the exponential blow-up we develop the following techniques for answering (regular) XPath queries posed on XML views.

2. Automaton-based rewriting for (regular) XPath. We introduce a rewriting method based on a notion of *mixed finite state automata* (MFA) to represent rewritten regular XPath queries. An MFA is a nondeterministic finite automaton (NFA) *"annotated"* with alternating finite state automata (AFA), which characterize data-selection paths and filters of a regular XPath query Q, respectively. The algorithm rewrites Q into an equivalent MFA $\mathcal{M}$. In contrast to the exponential blowup, the size of $\mathcal{M}$ is bounded by $O(|Q||\sigma||D_V|)$. This makes it possible to answer queries on views via rewriting. To our knowledge, although a number of automaton formalisms were proposed for XPath and XML stream (e.g. [6, 13]), they cannot characterize regular XPath queries, as opposed to MFA.

3. Evaluation of rewritten query. We provide an efficient algorithm for evaluating MFA $\mathcal{M}$ (rewritten regular XPath queries) on XML source T. While there have been a number of evaluation algorithms developed for XPath, none is capable of processing regular XPath queries. Previous algorithms for XPath (e.g., [16]) require at least two passes of T: a bottom-up traversal of T to evaluate filters, followed by a top-down pass of T to select nodes in the query answer. In contrast, our evaluation algorithm combines the two passes into a single top-down pass of T during which it both evaluates filters and identifies potential answer nodes. The key idea is to use an auxiliary graph, often far smaller than T, to store potential answer nodes. Then, a single traversal of the graph suffices to find the actual answer nodes. The

algorithm effectively avoids unnecessary processing of subtrees of T that do not contribute to the query answer. It is not only the first efficient algorithm for evaluating regular XPath queries (MFA), but also provides an efficient (alternative) algorithm to evaluate XPath queries.

4. Implementation and experimental study. We have implemented a prototype system SMOQE (Secure MOdular Query Engine [10]) for answering queries on XML views, *fully* supporting the rewriting and evaluation techniques mentioned above. Using SMOQE we have conducted an experimental study, which clearly demonstrates that our evaluation techniques are efficient and scale well. For regular XPath queries, we compared the SMOQE evaluation of queries with that of their XQuery translation, and found that the latter requires considerably more time. Furthermore, SMOQE outperforms the widely used XPath engine Xalan (default XPath implementation in Java 5), whether Xalan uses its interpretive processor or its high performance compiling processor (XSLTC), when evaluating XPath queries.

In summary, we provide the first practical and complete solution to answering regular XPath queries posed on (virtual and possibly recursively defined) XML views. It is provably efficient: it has a *linear-time* data complexity and a quadratic combined complexity. Furthermore it yields the first efficient technique for processing regular XPath queries, whose need is evident since regular XPath is increasingly being used both as a stand-alone query language and as an intermediate language in query translation [11].

Organization. Section 2 reviews (regular) XPath and XML views. Section 3 discusses the closure property of (regular) XPath rewriting. Section 4 introduces MFA and Section 5 describes the rewriting algorithm. Section 6 presents the MFA evaluation algorithm, followed by experimental results in Section 7. Related work is discussed in Section 8, followed by conclusions in Section 9.

2. Background

In this section we review XPath [4], regular XPath [19], DTDs and XML views considered in this paper.

2.1. XPath and Regular XPath

We consider a class of *regular* XPath queries proposed and studied in [19], denoted by $\mathcal{X}_{\text{reg}}$ and defined as follows:

$$Q ::= \epsilon \mid A \mid Q/Q \mid Q \cup Q \mid Q^* \mid Q[q],$$
$$q ::= Q \mid Q/\textit{text()} = \text{`}c\text{'} \mid \neg Q \mid Q \wedge Q \mid Q \vee Q$$

where ϵ is the empty path (*self*), A is a label (tag), '$\cup$' represents *union*, '$/$' is the *child-axis*, and $*$ is the Kleene star; $[q]$ is referred to as a *filter*, in which Q is an $\mathcal{X}_{\text{reg}}$ expressions, c is a string constant, and $\neg, \wedge, \vee$ are the Boolean negation, conjunction and disjunction, respectively. Regular XPath extends regular expressions by allowing filters [19], and extends XPath by supporting Kleene closure Q^* as opposed to the restricted recursion '//' (the *descendant-or-self axis*).

Like XPath queries, when an $\mathcal{X}_{\text{reg}}$ query Q is evaluated at a node v in an XML tree T, it returns the set of nodes of T reachable via Q from v, denoted by $v[\![Q]\!]$.

We also study an *XPath fragment* of $\mathcal{X}_{\text{reg}}$, denoted by $\mathcal{X}$, which is defined by replacing Q^* with '//' in the definition above. Note that given a DTD D of the documents on which queries are posed, '//' is expressible in $\mathcal{X}_{\text{reg}}$ as $(\bigcup Ele)^*$, where $\bigcup Ele$ denotes the union of all the labels in D.

Example 2.1: Consider an XML document T conforming to the document DTD D in Fig. 1(a). The regular XPath query

Q = *department/patient*[$q_0 \wedge (q_1/(q_1)^*)$]/*pname*
q_0 = *visit/treatment/medication/diagnosis/text() = "heart disease"*
q_1 = *parent/patient*[$\neg q_0$]/*parent/patient*[q_0]

when evaluated on T, returns the names of patients who have heart disease and the disease appears in their ancestors but always skips a generation. Such queries, which look for certain patterns, are often encountered in medical research. Note that the query is in the fragment $\mathcal{X}_{\text{reg}}$, but is not expressible in the XPath fragment $\mathcal{X}$. □

In this work we focus on regular XPath queries with only downward modalities since they are most commonly used in practice. As will be seen shortly, rewriting queries is already challenging in this setting. It is thus necessary to understand rewriting of these basic queries before dealing with full-fledged XPath or XQuery.

2.2. DTD

Following [9], we represent a DTD D as a triple (Ele, P, r), where Ele is a finite set of *element types*; r is a distinguished type in Ele, called the *root type*; P defines the element types: for each A in Ele, $P(A)$ is a regular expression of the form: *str*, ϵ, $B_1, \ldots, B_n$, or $B_1 + \cdots + B_n$. Here *str* denotes PCDATA, ϵ is the empty word, B_i is either B or of the form B^* where B is in Ele (referred to as a *child* type of A), and '$+$', ',' and '$*$' denote *disjunction* (with $n > 1$), *concatenation* and the *Kleene star*, respectively. We refer to $A \rightarrow P(A)$ as the *production* of A. This form of DTD's does not lose generality since any DTD can be converted to a DTD of this form by using new element types.

A DTD can be represented as a graph, as shown in Fig. 1. It is *recursive* if the corresponding graph is *cyclic*. For example, both DTD's depicted in Fig. 1 are recursive.

2.3. XML Views

We consider views defined by annotating a DTD [9]. This is similar in spirit to XML view specification in commercial systems, e.g. annotated XSD's (AXSD) in Oracle XML DB [21] and Microsoft SQLServer 2000 SQLXML [20], and Document Access Definitions (DAD) of IBM DB2 XML Extender [15]. Specifically, we define an XML view as a mapping $\sigma : D \rightarrow D_V$, where D is a *document* DTD, D_V is a *view* DTD. Given an XML document T of D, the mapping generates an XML view $\sigma(T)$ that conforms to the view

Query rewriting	Views	Closure	Complexity
from $\mathcal{X}$ to $\mathcal{X}$	non-rec.	Yes [9]	EXPTIME-complete
from $\mathcal{X}$ to $\mathcal{X}$	recursive	No	NA
from $\mathcal{X}$ to $\mathcal{X}_{\text{reg}}$	arbitrary	Yes	EXPTIME-complete
from $\mathcal{X}_{\text{reg}}$ to $\mathcal{X}_{\text{reg}}$	arbitrary	Yes	EXPTIME-complete

Figure 2. Closure property and complexity

DTD D_V. More specifically, for each element type A and its child type B in D_V (i.e., each edge (A, B) in the DTD graph of D_V), σ maps (A, B) to a query $\sigma(A, B)$ defined on documents T of D. Intuitively, given an A element, $\sigma(A, B)$ generates its B children in the view by extracting data from T. The query $\sigma(A, B)$ is in the regular XPath fragment $\mathcal{X}_{\text{reg}}$ given above. The XML view is *recursive* if the view DTD D_V is recursive.

Example 2.2: Figure 1(c) defines the view σ_0 described in Example 1.1. The semantics of σ_0, informally presented, is as follows: Given a *hospital* document T, σ_0 generates a view $\sigma_0(T)$ top-down, which conforms to the view DTD of Fig. 1(b). The query Q_1 (i.e., $\sigma_0(\textit{hospital}, \textit{patient})$) extracts from T those *patient*s who have heart disease. For the patients extracted by Q_1, (a) Q_2 finds their *parent* nodes, which are in turn processed by Q_4 and then inductively by Q_2 and Q_3 to form the *parent* hierarchy, and (b) Q_3 finds the *record* (i.e., *visit*) data, which can be either be *empty* (i.e., *test*) or *diagnosis*, handled by Q_5, Q_6, respectively. □

3. The Closure Property of (Regular) XPath

We next study the closure property and complexity of XPath and regular XPath query rewriting. The main results of this section are summarized in Fig. 2.

Formally, an XML query language L is *closed under rewriting* if there exists a computable function $F : L \to L$ that, given any view definition $\sigma : D \to D_V$ and any query Q in L over D_V, computes query $Q' = F(Q)$ in L such that for any document T of D, $Q(\sigma(T)) = Q'(T)$. While one may consider translating an XPath query Q to an equivalent Q' in a richer language, e.g. XQuery or XSLT, it is vastly preferable to have an XPath translation since it is more efficient to evaluate XPath queries than queries in the aforementioned Turing-complete languages. The closure property is desirable since rewriting should not be penalized by paying the higher price for evaluating and optimizing queries in a richer language than that of the original query.

It was shown in [9] that the class $\mathcal{X}$ of XPath queries defined in Section 2 is closed under query rewriting for *non-recursive* views. However, below we show that in the presence of recursion in a view definition, this is no longer the case (even when the annotating queries are in $\mathcal{X}$).

Theorem 3.1: *For recursively defined* XML *views, the fragment $\mathcal{X}$ is not closed under query rewriting.* □

In contrast, the fragment $\mathcal{X}_{\text{reg}}$ of regular XPath given in the last section is closed under query rewriting:

Theorem 3.2: *For arbitrary* XML *views (recursive or non-recursive), $\mathcal{X}_{reg}$ is closed under rewriting.* □

Example 3.1: Recall the view $\sigma_0 : D \to D_V$ defined in Example 2.2 and the query Q given in Example 1.1. Using the queries Q_1, Q_2, Q_3, Q_4 and Q_6 from the view specification in Fig. 1(c), we can compute a correct rewriting Q' of query Q. Specifically: Q' = $Q_1[Q_2/Q_4/(Q_2/Q_4)^*/Q_3/Q_6/\textit{text()} =$'heart disease'$]$. For any document T that conforms to D, $Q'(T) = Q(\sigma_0(T))$. □

Although it is always possible to rewrite a (regular) XPath query on a view to an equivalent regular XPath query on the source, it is often prohibitively expensive if it is to directly compute $\mathcal{X}_{\text{reg}}$ queries as output. Indeed, the rewriting problem subsumes the problem for translation from NFA's to regular expressions. The latter problem is EXPTIME-complete [8]: the size of the explicit representation of a regular expression is exponential in the size of the NFA. Worse still, it remains exponential even if the NFA is acyclic.

Corollary 3.3: *There exist a view definition $\sigma : D \to D_V$ and a query Q in $\mathcal{X}$ such that for any Q' in $\mathcal{X}_{reg}$, if $Q(\sigma(T)) = Q'(T)$ for all* XML *trees T of D, then the size $|Q'|$ of Q', when represented as an $\mathcal{X}_{reg}$ query, is exponential in $|Q|$ and the size $|D_V|$ of D_V. The lower bound remains intact even when D_V is non-recursive.* □

4. Mixed Finite State Automata

The exponential lower bound of Corollary 3.3 tells us that a direct rewriting into (regular) XPath is beyond reach in practice. To overcome this, in this section we introduce a new representation of $\mathcal{X}_{\text{reg}}$ queries, referred to as *mixed finite state automata* (MFA). Along the same lines as NFA for regular expressions, MFA characterize $\mathcal{X}_{\text{reg}}$ queries and avoid the exponential blowup of rewriting. Leveraging MFA we shall present a practical solution to the rewriting problem by providing (a) a low polynomial-time algorithm for rewriting $\mathcal{X}_{\text{reg}}$ queries on a view into the MFA-presentation of equivalent $\mathcal{X}_{\text{reg}}$ queries on the source (Section 5), and (b) a linear-time algorithm for directly evaluating the MFA-presentation of $\mathcal{X}_{\text{reg}}$ queries on the source (Section 6).

While a regular expression can be efficiently represented as a graph or a NFA, for $\mathcal{X}_{\text{reg}}$ queries a notion of automaton representation is not yet available. The difficulties of characterizing an $\mathcal{X}_{\text{reg}}$ query Q as an automaton include the following: (a) Q typically involves both "selecting" paths that are to extract and return nodes, and filters that constrain the extraction; (b) a filter $[q]$ in Q may involve Boolean operators '$\wedge, \vee, \neg$' and constant test $p/\textit{text()} =$ 'c', which are not encountered in regular expressions; (c) worse still, it may be nested: q itself may be a query of the form $p[q_1]$; and (d) the sub-query p of p^* may itself contain Kleene closure.

Mixed finite state automata (MFA). In light of this we define an MFA $\mathcal{M}$ as a *nondeterministic finite automaton*

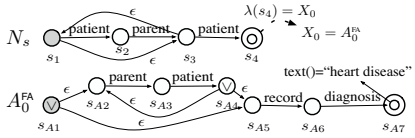


Figure 3. NFA N_s **and** AFA A_0^{FA} **in Example 4.1**

(NFA) in which a state may be annotated with an *alternating finite state automaton* (AFA). Intuitively, the NFA in $\mathcal{M}$ is to capture the selecting paths of an $\mathcal{X}_{\text{reg}}$ query Q and the AFA's are to characterize the filters in Q.

Formally, an MFA $\mathcal{M}$ is defined to be $(N_s, \vec{\mathcal{A}})$, where (a) $\vec{\mathcal{A}}$ is a set of bindings $X_i = A_i^{\text{FA}}$, X_i is a name and A_i^{FA} is an AFA as defined below; (b) $N_s = (K_s, \Sigma_s, \delta_s, s, F, \lambda)$ is a variation of NFA, referred to as the *selecting* NFA *of* $\mathcal{M}$, where $K_s, \Sigma_s, \delta_s, s, F$ are the states, alphabet, transition function, start state and final states as in the standard NFA definition; and λ is a partial mapping from K_s to names X_i, i.e., a state in N_s may be annotated with a single X_i.

We employ a variation of AFA's [24] to represent $\mathcal{X}_{\text{reg}}$ filters. We define an AFA A^{FA} to be $(K, \Sigma, \delta, s, F)$, where (a) K is a set of states partitioned into K_{op}, K_l and F, where K_{op} is a set of *operator states* marked with AND, OR or NOT, K_l is a set of *transition* states, and F is a set of final states optionally annotated with predicates of the form *text()='c'* or *position()=k*; (b) Σ is a set of labels; (c) s is the start state in K; and (d) δ is the transition function defined as follows. (1) For a state s_1 in K_{op}, δ is only defined for empty string ϵ and $\delta(s_1, \epsilon) = K'$, where K' is a subset of K. In particular, if s_1 is marked with NOT, K' has a single state in it. (2) For each state s_2 in K_l, δ is only defined for a single label $A \in \Sigma$ and $\delta(s_2, A)$ contains a single state in K. (3) δ is not defined for any state in F. Observe that except for operator states marked with AND or OR, from each state at most one state can be reached via δ. These operator states capture Boolean operators $\wedge, \vee$ and $\neg$ in $\mathcal{X}_{\text{reg}}$ filters.

Example 4.1: Consider an $\mathcal{X}_{\text{reg}}$ query Q_0 posed on an XML tree conforming to the DTD of Fig. 1(b), which is to find all patients who have an ancestor diagnosed with heart disease:

Q_0 = (*patient*/*parent*)*/*patient*[q_0],
q_0 = (*parent*/*patient*)*/*record*/*diagnosis*[*text()* ="heart disease"].

Consider MFA $\mathcal{M}_0$ in Fig. 3. It consists of a *selecting* NFA N_s (shown at the top of the figure), and an AFA A_0^{FA}, corresponding to the filter q_0 (shown at the bottom). The MFA $\mathcal{M}_0$ is *equivalent* to Q_0, in the sense that when *evaluating* $\mathcal{M}_0$ at a node n in an XML tree T (described below), it returns the same set $n[\![\mathcal{M}_0]\!]$ of nodes as $n[\![Q_0]\!]$.

The (conceptual) evaluation of $\mathcal{M}_0$ is illustrated, by example, in Fig. 4. At the root node 1 of the tree, $\mathcal{M}_0$ associates a set $\{s_1, s_3\}$ of N_s states, where s_1 is the start state of N_s and s_3 is reached from s_1 via an ϵ-transition. It then inspects the children of node 1: for all its children labeled *patient* (nodes 2 and 9), it associates them with states s_2, s_4, moves down to these children and processes them inductively, in parallel. At a node associated with state s_2, for

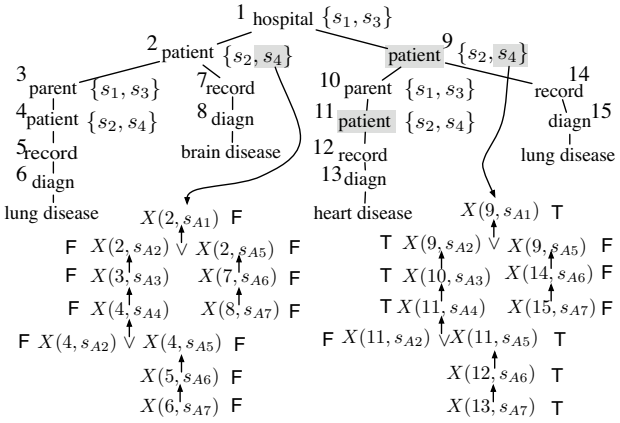


Figure 4. Conceptual evaluation of $\mathcal{M}_0$

all its children labeled *parent* (nodes 3 and 10) it associates them with states s_1, s_3 and processes them in the same way as at the parent node of the tree. In the case of state s_4, since this state is annotated with A_0^{FA}, any node associated with state s_4 must also evaluate A_0^{FA} (the evaluation of A_0^{FA} is described below). This is the case for both nodes 2 and 9. Since s_4 is a final state, if A_0^{FA} evaluates to true, the corresponding node is added to $n[\![\mathcal{M}_0]\!]$ (the *answer* of $\mathcal{M}_0$).

When the AFA A_0^{FA} is invoked, e.g., at node 2, a Boolean value $2[\![A_0^{\text{FA}}]\!]$ is computed as follows: A_0^{FA} associates a Boolean variable $X(2, s_{A1})$ with node 2, whose value is to be computed and treated as $2[\![A_0^{\text{FA}}]\!]$, where s_{A1} is the start state of A_0^{FA}. It then traverses the subtree rooted at node 2 top-down. From s_{A1} there are two ϵ-transitions to s_{A2} and s_{A5}, and thus node 2 is also associated with variables $X(2, s_{A2})$ and $X(2, s_{A5})$ for these AFA states. Since s_{A1} is an OR state, $X(2, s_{A1})$ is computed via $X(2, s_{A2}) \vee X(2, s_{A5})$. To compute $X(2, s_{A5})$, it inspects the children of node 2: if no child is labeled *record*, no A_0^{FA} transition can be made from s_{A5} and $X(2, s_{A5})$ is assigned false; otherwise, for *all children* labeled *record*, in this case node 7, it associates a variable $X(7, s_{A6})$, moves down to these children and process them in parallel. Inductively, $X(7, s_{A6})$ is true if node 7 has a child labeled *diagnosis* and carrying text "heart disease", and if so, $X(2, s_{A5})$ is assigned true as well. Similarly, $X(2, s_{A2})$ is computed and becomes true if it has a descendant that is reachable via *(parent/patient)***/record/diagnosis* and carries text "heart disease". If either $X(2, s_{A2})$ or $X(2, s_{A5})$ is true, then $X(2, s_{A1})$ is true and so is the output $2[\![A_0^{\text{FA}}]\!]$. This is not the case here, however, and A_0^{FA} returns false. □

Observe the following. (a) Although A_0^{FA} traverses the subtree top-down, the Boolean variables are computed bottom-up. (b) In A_0^{FA} the only operator states are OR states (s_{A_1}, s_{A4}); but AND and NOT states can be processed similarly. (c) The conceptual evaluation requires multiple passes over a subtree, one pass for each filter. In contrast, our evaluation algorithm in Section 6 requires only one pass of the input tree, regardless of the number of filters.

Equivalence of MFA and $\mathcal{X}_{\text{reg}}$ queries. An MFA $\mathcal{M}$ and an $\mathcal{X}_{\text{reg}}$ query Q are *equivalent* if for each XML tree T and any node n in T, $n[\![\mathcal{M}]\!] = n[\![Q]\!]$, where $n[\![\mathcal{M}]\!]$ (resp. $n[\![Q]\!]$) denotes the result of evaluating an MFA $\mathcal{M}$ (resp. Q) at n.

The result below tells us that we can identify a class of MFA's, namely, MFA's with a syntactic restriction on AFA's called the *split property*, to precisely capture the fragment $\mathcal{X}_{\text{reg}}$ of regular XPath queries; as a result, MFA's can be used to represent $\mathcal{X}_{\text{reg}}$ queries.

Theorem 4.1: *For any $\mathcal{X}_{reg}$ query Q, there exists an equivalent* MFA *$\mathcal{M}$ with the split property, and vice versa.* □

5. Rewriting Algorithm

We now present an efficient algorithm, called rewrite (not shown due to space constraints), for rewriting (regular) XPath queries on arbitrary views into equivalent MFA's on the underlying documents.

Algorithm rewrite takes as input an $\mathcal{X}_{\text{reg}}$ query Q and a view definition $\sigma : D \rightarrow D_V$; it returns an MFA $\mathcal{M} = (N_s, \vec{\mathcal{A}})$ as output, such that for any XML tree T of D, $\mathcal{M}$ on T yields the same result as Q on $\sigma(T)$. It is based on dynamic programming: for each sub-query Q' of Q and each element type A in D_V, it computes a local translation $\mathsf{rewr}(Q', A)$, i.e., an MFA on D that is equivalent to Q' when Q' is evaluated at any A elements of D_V. The MFA $\mathsf{rewr}(Q', A)$ is constructed inductively, based on structure of Q'. It assembles local translations to obtain $\mathcal{M} = \mathsf{rewr}(Q, r)$, where r is the root type of D_V.

Example 5.1: Given query Q_0 of Example 4.1 on the view σ_0 of Example 2.2, assume that we want to compute $\mathsf{rewr}(Q_0, \mathit{hospital})$. Fig. 5(a) shows a simplified parse tree of Q_0. Algorithm rewrite uses this parse tree to inductively build the MFA for Q_0. In more detail, Fig. 5(b) shows three MFAs and two AFAs that are the basis of the induction of the rewriting of Q_0. Specifically, $\mathcal{M}_0^0$ corresponds to $\mathsf{rewr}(parent, patient)$, $\mathcal{M}_0^1$ to $\mathsf{rewr}(patient, parent)$ and $\mathcal{M}_0^2$ to $\mathsf{rewr}(patient, hospital)$. Notice that the construction of $\mathcal{M}_0^2$ also requires the construction of A_0^{FA}.

Figure 5(c) shows how Algorithm rewrite uses these basic blocks to build inductively the MFA $\mathsf{rewr}(Q_0, hospital)$. Specifically, it constructs $\mathcal{M}_0^3 = \mathsf{rewr}(Q_0^0/Q_0^1, hospital)$ by concatenating MFA $\mathcal{M}_0^2$ and $\mathcal{M}_0^0$. Then, it constructs $\mathcal{M}_0^5 = \mathsf{rewr}((Q_0^0/Q_0^1)^*, hospital)$ by concatenating $\mathcal{M}_0^3$ with $\mathcal{M}_0^4 = \mathsf{rewr}(Q_0^0/Q_0^1, parent)$ and adding appropriate ϵ-transitions for the recursion. Finally, the algorithm considers the rewriting of $Q_0^2[q_0]$ and concatenates this to MFA $\mathcal{M}_0^5$ to compute the final result. □

Similarly rewrite constructs AFA's for filters q, with the following features. (a) For a "path sub-queries" Q' (i.e., of the form p given in Section 2) of q, rewrite defines its AFA in same way as MFA for Q'. (b) For logical connectives $\wedge$,$\vee$, or $\neg$, rewrite connects inductively obtained AFA's by introducing a new logical state, i.e., an AND, OR, or NOT state. (c) For nested filters, i.e., $q = p[q_1]$ where $q_1 = p'[q_1']$, rewrite constructs a *single* AFA, rather than nested AFA's, for q, by "concatenating" the AFA's for p and q_1.

Example 5.2: Consider the filter q_0 in the query Q_0 of Example 4.1. Figure 5(b) shows how its AFA A_1^{FA} is constructed step-wise, by reusing the MFA's $\mathcal{M}_0^0, \mathcal{M}_0^1, \mathcal{M}_0^2$ for path sub-queries, and by concatenating these and "local" AFA's to build A_0^{FA} and A_1^{FA}. Note that although q_0 contains a nested filter *text()*='heart disease', the two filters are combined into a single AFA and *no "nested" AFA's* are required. □

Concluding, we have the following result, which, in contrast to Corollary 3.3, justifies the use of MFA's.

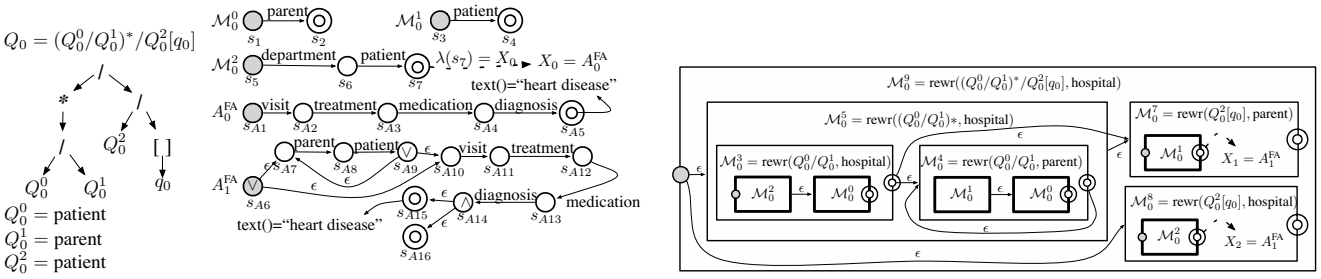
Theorem 5.1: *Given a view definition $\sigma : D \rightarrow D_V$ and an $\mathcal{X}_{reg}$ query Q over D_V, Algorithm* rewrite *computes an equivalent* MFA *of size at most $O(|Q||\sigma||D_V|)$ over the original document in at most $O(|Q|^2|\sigma||D_V|^2)$ time.* □

6 Evaluation Algorithm

To make query rewriting a practical approach it is necessary to be able to efficiently evaluate MFA's. We next present an evaluation algorithm for MFA's, referred to as HyPE (Hybrid Pass Evaluation, Fig. 6). Algorithm HyPE takes as input a document tree T, a context node n in T and an MFA $\mathcal{M} = (N_s, \vec{\mathcal{A}})$; it outputs $n[\![\mathcal{M}]\!]$. The desired result $r[\![\mathcal{M}]\!]$ is obtained by invoking HyPE with the root r of T.

A salient feature of HyPE is that it requires only a *single top-down* pass over the document tree, and a *single* pass over an auxiliary structure, which in most cases is much smaller than the document tree. It employs several pruning strategies in its top-down pass to avoid visiting irrelevant parts of the tree and the computation of irrelevant AFA's.

Since any regular XPath query can be transformed into an MFA, HyPE serve as a stand-alone evaluation algorithm for regular XPath, beyond the rewriting context. To the best of our knowledge, HyPE is the first practical algorithm for evaluating regular XPath. Indeed, no practical algorithm has been provided thus far that can be done within a bounded number of tree traversals. For XPath only, a two-pass algorithm was presented in [16]: a bottom-up phase for evaluating filters followed by a top-down phase for selecting nodes. However, it requires a pre-processing step (another scan of the tree) during which the document tree is converted to a special data format (a binary representation of the tree), and the construction of a tree automata which are more complex than MFA's and are possibly large. Algorithm HyPE requires neither pre-processing of the data nor the construction of tree automaton. Moreover, in contrast to HyPE, the two-pass XPath evaluation algorithm may have to evaluate filters at nodes in its first phase, although these nodes will not be accessed in its second phase. As will be verified in Section 7, the pruning technique of HyPE speeds up the



(a) Parse tree of Q_0 (b) Basic rewriting components (c) Resulted MFA rewriting

Figure 5. Rewriting query Q_0 to the corresponding MFA

Algorithm HyPE$(n, T, \mathcal{M})$.

Input: Context node n, tree T, MFA $\mathcal{M}$.
Output: Answer set $n[\![\mathcal{M}]\!]$.

1. Initialize mstates(n), fstates$\downarrow(n)$, and $\mathcal{P} = \{n\}$;
2. cans(n):=PCans$(n,$ mstates$(n),$ fstates$\downarrow(n))$;
3. Traverse cans(n) starting from set I of cans(n), add
4. visited nodes $\nu(v)$ for vertices in cans(n) to $n[\![\mathcal{M}]\!]$;
5. return $n[\![\mathcal{M}]\!]$;

Procedure PCans$(n, T,$ mstates$(n),$ fstates$\downarrow(n))$

Input: Context node n, tree T, states mstates(n), vector fstates$\downarrow(n)$.
Output: Candidate answers cans(n).

1. if mstates$(n) \neq \emptyset$ or fstates$\downarrow(n) \neq \vec{\emptyset}$ then
2. for each child v of n then
3. $push(v, \mathcal{P})$;
4. mstates(v):=NextNFAStates(mstates$(n), v, N_s)$;
5. fstates$\downarrow(v)$:=NextAFAStates(fstates$\downarrow(n), v, \vec{\mathcal{A}})$;
6. for each $s \in$ mstates(v), s.t. $\lambda(s) = X_i, i \in [1..\kappa]$, do
7. add initial state of A_i^{FA} to fstates$\downarrow(v)[i]$;
8. cans(v):=PCans$(v,$ mstates$(v),$ fstates$\downarrow(v))$;
9. cans(n):=connect mstates(n) to I of cans(v);
10. Set the set I of initial vertices in cans(n) to mstates(n);
11. for each i such that fstates$\downarrow(n)[i] \neq \emptyset$ do
12. fstates$\uparrow(n)[i]$:=PrevAFAStates(fstates$\uparrow(n)[i]$);
13. fstates$\uparrow(n)[i]$:=fstates$\uparrow(n)[i] \cup \{f \in F \mid f$ is true at $n\}$;
14. for each $s \in$ mstates(n) s.t. associated AFA is false do
15. Delete s and all its in- and outgoing edges from cans(n);
16. for each final state f of mstates(n) in cans(n) do
17. assign n to f, i.e., $\nu(f) := n$;
18. $pop(n, \mathcal{P})$;
19. if $head(\mathcal{P}) \neq \emptyset$ do
20. u:=$head(\mathcal{P})$;
21. fstates$\uparrow(u)$:=fstates$\uparrow(u) \cup$ fstates$\uparrow(n)$;
22. return cans(n);

Figure 6. Evaluation algorithm for MFA's.

evaluation of *both* regular XPath and XPath queries.

In a nutshell, HyPE consists of two phases (not to be confused with two passes of the tree T). In the first phase, the tree T is traversed (top-down) depth-first, during which the MFA $\mathcal{M}$ prunes away irrelevant subtrees and identifies which AFA's in $\vec{\mathcal{A}}$ need to be evaluated at nodes in the tree. Visited nodes are pushed into a stack $\mathcal{P}$. This stack is used to evaluate the AFA's in a synthesized (bottom-up) way. A node is popped from $\mathcal{P}$ once all its related AFA's have been evaluated. The size of $\mathcal{P}$ is at most *the depth* of T. During this traversal, HyPE also constructs an auxiliary DAG structure, called cans (for candidate answers), representing the history of the run of the selecting NFA N_s. Vertices in cans will correspond to states in this run for which the associated AFA evaluated to true. Moreover, vertices in cans are possible annotated with a node in T which is potentially in the answer set $n[\![\mathcal{M}]\!]$. A node in T associated with a vertex in cans will be in $n[\![\mathcal{M}]\!]$ if this node is reachable from a node in cans corresponding to an initial state of N_s at context node n. This allows for distinguishing between potential and real answer nodes in cans. In the second phase, cans is traversed top-down to identify the real answer nodes. The size of cans is typically much smaller than T.

Example 6.1: Consider the MFA $\mathcal{M}_0$ in Fig. 3 and the tree T shown in Fig. 4. We illustrate how HyPE evaluates $\mathcal{M}_0$ on T through the table in Fig. 7. In the figure, we assume that HyPE already traversed, top-down, the left-most patient (node 2) in the tree and we *join* the execution of HyPE at the point where node 9 is considered (the first row in the table). Each row in the table corresponds to a step in the execution of HyPE during which the node n at the head of the stack $\mathcal{P}$ is considered. In the table, we also show (a) mstates(n), i.e., the ϵ-closure of states in N_s (i.e., the set of states reached by following one or more ϵ moves), reached by descending to n in T; (b) fstates$\downarrow(n)$, i.e., a set of states in A_0^{FA}. If this set is non-empty then n will be involved in the bottom-up evaluation of A_0^{FA}; and (c) fstates$\uparrow(n)$, i.e., a set of states (and their truth values) of A_0^{FA} used in the bottom-up evaluation of A_0^{FA}. At the bottom of Fig. 7, we show the auxiliary structure cans. It is constructed during the traversal of T. We indicate, through boxes, which rows in the table are responsible for the corresponding updates to cans (note that cans is constructed from left to right in Fig. 7).

Going back to the figure, the first row of the table indicates two things. First, since s_4 is a final state of N_s, we know that node 9 is a candidate answer. Second, state s_4 is annotated with A_0^{FA} and therefore we need to evaluate A_0^{FA} to determine whether node 9 is an actual answer. We *remember* that A_0^{FA} needs to be evaluated on node 9 by initializing fstates$\downarrow(9)$ with the initial states of A_0^{FA}. Consider now the second row in the table. Node 10 is in the

stack $\mathcal{P}$	mstates	fstates↓	fstates↑ true	fstates↑ false
⋮	⋮	⋮	⋮	⋮
(1|9)	s_2, s_4	s_{A1}, s_{A2}, s_{A5}	$\emptyset$	$\emptyset$
(1|9|10)	s_1, s_3	s_{A3}	$\emptyset$	$\emptyset$
(1|9|10|11)	s_2, s_4	s_{A2}, s_{A4}, s_{A5}	$\emptyset$	$\emptyset$
(1|9|10|11|12)	$\emptyset$	s_{A6}	$\emptyset$	$\emptyset$
(1|9|10|11|12|13)	$\emptyset$	s_{A7}	$\emptyset$	$\emptyset$
(1|9|10|11|12)	$\emptyset$	s_{A6}	s_{A6}	$\emptyset$
(1|9|10|11)	s_2, s_4	s_{A2}, s_{A4}, s_{A5}	s_{A1}, s_{A4}, s_{A11}	$\emptyset$
(1|9|10)	s_1, s_3	s_{A3}	s_{A3}	$\emptyset$
(1|9)	s_2, s_4	s_{A1}, s_{A2}, s_{A5}	s_{A1}, s_{A2}, s_{A4}	$\emptyset$
(1|9|14)	$\emptyset$	s_{A6}	$\emptyset$	$\emptyset$
(1|9|14|15)	$\emptyset$	s_{A7}	$\emptyset$	$\emptyset$
(1|9|14)	$\emptyset$	s_{A6}	$\emptyset$	s_{A6}
(1|9)	s_2, s_4	s_{A1}, s_{A2}, s_{A5}	s_{A1}, s_{A2}, s_{A4}	s_{A5}
(1)	s_1, s_3	$\emptyset$	$\emptyset$	$\emptyset$

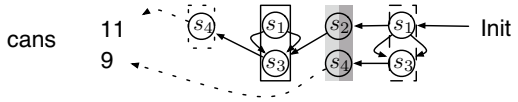


Figure 7. HyPE **evaluation.**

top of $\mathcal{P}$. Furthermore, mstates(10) is $\{s_1, s_3\}$ and is obtained by calling function NextNFAStates with arguments the mstates$(9) = \{s_2, s_4\}$ (line 4 in algorithm of Fig. 6). Similarly, NextAFAStates computes fstates↓$(10) = \{s_{A3}\}$ from fstates↓(9) (line 5 in Fig. 6). The fact that fstates↓(10) is non-empty tells us that node 10 is relevant for the evaluation of A_0^{FA}. The actual evaluation of A_0^{FA} starts when in the head of $\mathcal{P}$ is node 13. At that point, fstates↓(13) includes the final state of A_0^{FA} and from that point on A_0^{FA} is evaluated bottom-up. This hybrid mixing of a top-down with a bottom-up evaluation is the distinguishing characteristic of HyPE. Essentially, HyPE uses the former evaluation type to determine when to initiate the latter. When HyPE returns to $\mathcal{P} = \{1, 9\}$ (the dark grey row of the table), the fact that fstates↑(9) includes $\{s_{A1} = \text{true}\}$ indicates that the evaluation of A_0^{FA} results in true. Therefore, node 9 is an actual answer. Concerning cans, this is constructed bottom-up. For each node n for which mstates$(n) \neq \emptyset$, mstates(n) is connected to the existing cans, each time the subtree below a child of n has been traversed. For example, when $\mathcal{P} = \{1, 9\}$ (dark gray row), mstates(9) is connected (using the transitions in $\mathcal{M}_0$) to the cans structure to its left. At this point, notice that by following the path s_2, s_3, s_4 we reach node 11 in T. Furthermore, through the new state s_4 node 9 is also reachable. When the construction of cans is completed (row with dashed box), a traversal of cans starting from the Init nodes shows that nodes 9 and 11 are still reachable and hence are in the answer of $\mathcal{M}_0$ on T. □

Complexity. The complexity of HyPE is determined by that of PCans (for constructing cans) and the traversal of cans. PCans needs for each context node n at most $O(|\mathcal{M}|)$ time. Moreover, connecting and updating cans takes at most $O(|\mathcal{M}|)$ time as well. Hence, the overall time complexity of PCans is $O(|T||\mathcal{M}|)$. Moreover, PCans requires a single scan of the input document T and cans. The space requirement of PCans is dominated by the size of cans, which, although in the worst case is $O(|T||\mathcal{M}|)$, is typically much smaller than $|T|$. Traversing cans takes again $O(|T||\mathcal{M}|)$ time in the worst case. As a consequence:

Theorem 6.1: *Given an* MFA $\mathcal{M}$ *and tree* T*,* HyPE *computes* $r[\![\mathcal{M}]\!]$ *in at most* $O(|T||\mathcal{M}|)$ *time and space.* □

Using the evaluation algorithm together with the rewriting algorithm, we obtain a complete practical method for answering queries on (virtual) views. The overall complexity of our method follows from Theorems 5.1 and 6.1.

Theorem 6.2: *Given an* $\mathcal{X}_{reg}$ *query* Q *on a view of an* XML *source* T*, our query answering method returns the answer to* Q *in* $O(|Q|^2|\sigma||D_V|^2 + |Q||\sigma||D_V||T|)$ *time.* □

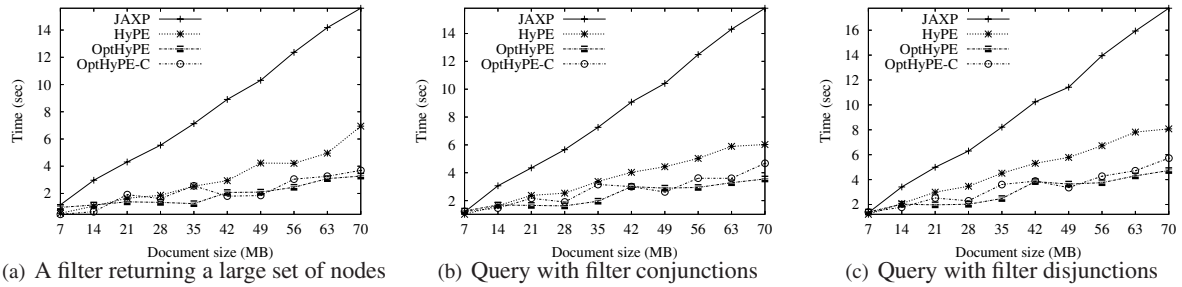
The size $|T|$ of the document is dominant and is typically much larger than the size $|D_V|$ of the view DTD and the size $|\sigma|$ of the view definition σ; when only $|T|$ is concerned (e.g., if D_V and σ are fixed as commonly encountered in practice), our method answers queries in *linear-time* (data complexity), and in quadratic combined complexity.

Variants of HyPE**.** Although HyPE already performs well in practice (see Section 7), we developed a novel index structure which enables HyPE to skip even more subtrees. In the following, we denote by OptHyPE the version of HyPE which is built on top of the index, and by OptHyPE-C the version of HyPE which uses a compressed version of the index.

7. Experimental Study

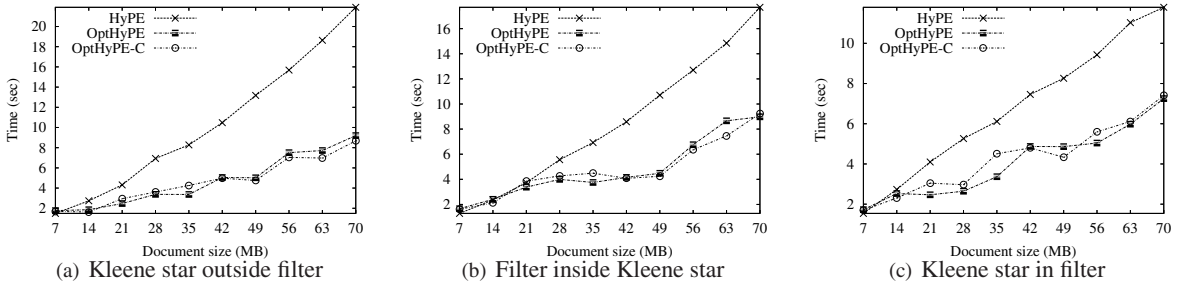
We have developed a prototype system SMOQE [10] supporting MFA's and algorithms rewrite and HyPE (and its variants OptHyPE and OptHyPE-C). In our experiments, we focused on the most time-consuming module of SMOQE, i.e., the query evaluator. The experiments were conducted on a dual 2.3GHz Apple Xserve with 4GB of memory. For the generation of our datasets, we used ToXGene [1]. We generated XML documents that conform to our recursive hospital DTD shown in Fig. 1(a), with sizes ranging from 7MB to 70MB, in 7MB increments. Each increment roughly corresponds to adding the medical history of 10,000 patients to our document tree. Therefore, the largest document stores the medical history of approximately 100,000 patients. The maximal depth of the trees is 13. The generated data consist mainly of element nodes, and to a lesser extent of text nodes. Therefore, the size of the document has a direct impact on query evaluation. For example, our smallest document (7MB) consists of 303,714 element nodes vs 151,187 text nodes. The text nodes are used to increase the selectivity of queries but their size is kept to a minimum (so as not to increase the document size).

Using the generated document trees, we conducted two sets of experiments, one regarding XPath evaluation, the



(a) A filter returning a large set of nodes (b) Query with filter conjunctions (c) Query with filter disjunctions

Figure 8. XPath query evaluation times



(a) Kleene star outside filter (b) Filter inside Kleene star (c) Kleene star in filter

Figure 9. regular XPath query evaluation times

other regarding regular XPath. The reported times are averaged over at least 5 runs of each experiment.

Evaluating XPath Queries. Since regular XPath subsumes XPath, we investigate the performance of HyPE and its variants for the evaluation of XPath queries.

We compared our performance with that of the Java API for XML Processing Reference Implementation (JAXP RI 1.3), which relies on XERCES and XALAN [23]. We also compared with JAXP-COMPILE, a version of JAXP that precompiles the input query and converts it into a set of Java classes. The two JAXP versions had similar performance and thus we only report one of them.

We ran various types of XPath queries with simple filters on data values, unions of queries, and Boolean combinations of filters. Figure 8 shows the evaluation time for three different types of XPath queries. We show the evaluation time both for queries with result sizes of a few hundreds of nodes (Figures 8(b) and (c)) and queries that return a few thousands of nodes (Fig. 8(a)). For each query type, we report the evaluation time for JAXP, HyPE, OptHyPE and OptHyPE-C. The figures show clearly that our algorithm consistently outperform JAXP by a factor of *three* for HyPE, and *four* for OptHyPE and OptHyPE-C. We also observe that in most cases, both optimized versions of HyPE run almost twice as fast as HyPE. Note as well that the performance of OptHyPE-C is almost identical to that of OptHyPE (while OptHyPE-C uses a compressed index).

Evaluating Regular XPath Queries. The second set of experiments investigated the performance of evaluating regular XPath queries with the different versions of HyPE. Existing alternatives rely on a translation of regular XPath into a more powerful query language like XQuery. We conducted a series of experiments following this approach. Specifically, we translated several regular XPath queries into XQuery and evaluated them in GALAX (*http://db.bell-labs.com/galax*). These experiments consistently showed that the queries in XQuery required considerably more time than their regular XPath counterparts. As a result we omit GALAX from our discussion because even for a simple regular XPath query on the smallest used document tree, GALAX needed more time than HyPE for the same query on the largest tree. Hence, we only focus on the relative performance of our algorithm.

We ran different types of regular XPath queries that involve Kleene star outside a filter, inside a filter, filters inside Kleene stars and combinations thereof. Figure 9 reports the evaluation time for three of these queries. The overall conclusion is consistent with our observations regarding XPath queries. Indeed, OptHyPE and OptHyPE-C show considerable improvement over HyPE.

An interesting observation is that HyPE prunes a substantial number of element nodes. Specifically, HyPE (resp. OptHyPE) prunes, on average, 78.2% (resp. 88%) of the element nodes for our example queries.

8. Related Work

There has been a host of work on rewriting queries posed on XML views to relational queries on top of RDBMS (e.g., [22, 12]). Even in this setting, recursion in the view DTD makes the translation challenging. As observed by [18], most of the existing approaches cannot translate recursive queries over recursive views (two exceptions are [22, 11]).

There has been little work on query rewriting for XML views in the native XML setting where one does not rely on any RDBMS, i.e., the setting considered in this paper. To

our knowledge, the only work in this context is [9], which showed that $\mathcal{X}$ is closed under query rewriting for *non-recursive* XML views. Our rewriting algorithm given here is the first practical solution to rewriting queries in XPath and its extension regular XPath over recursive XML views.

In [19], regular XPath was introduced and it was shown that a regular XPath query Q can be evaluated over an XML tree T in $O(|Q||T|)$ time, requiring multiple passes over the document tree. A two-pass algorithm for XPath was developed in [16], but it cannot be easily extended to deal with the Kleene star. As already observed in Section 6, even when only XPath is concerned, our evaluation algorithm, HyPE, does not need a pre-processing step (another scan of T) that is required by the algorithm of [16], and is more effective in pruning irrelevant nodes when traversing T, among other things. To our knowledge, HyPE is the first practical algorithm to evaluate regular XPath queries.

As remarked in Section 1, several automaton formalisms were proposed for processing multiple XPath queries on streaming XML (e.g. [6, 13]). The idea of AFA was explored by [13] for evaluating XPath filters. However, no previous work has attempted to characterize regular XPath in terms of both NFA and AFA in an integrated automaton.

Another line of research concerns view-based *query rewriting* and *answering* (see [14] for a survey). Here, given a set of (materialized) views and a query Q on the underlying database, the goal is to answer Q solely on the basis of the views. The problem we consider here is the *opposite scenario* where the query Q is posed on the view, and it is to find a rewriting Q' of Q on the underlying document.

9. Conclusion

We have provided a solution for efficiently answering regular XPath queries posed on possibly recursively defined XML views. On the theoretical side, we have established results for the closure property and complexity of rewriting (regular) XPath queries on views into (regular) XPath queries on the source. On the practical side, we proposed a practical approach for query rewriting, by using MFA as an intermediate representation of rewritten regular XPath queries. The novelty of the approach consists in (a) an algorithm for rewriting regular XPath queries on XML views to equivalent MFA on the source, and (b) an efficient evaluation algorithm for MFA. These yield an effective method for answering queries posed on XML views of XML data, and are useful in enforcing XML security, among other things. Furthermore, our evaluation algorithm is among the first for efficiently processing regular XPath queries. We have fully implemented a prototype system supporting all these algorithms, and our experimental results verified the efficiency of our techniques. We are currently studying extension of our rewriting algorithms to handle XML queries and views defined in XQuery and XSLT.

Acknowledgment. Wenfei Fan is supported in part by EPSRC GR/S63205/01, GR/T27433/01 and BBSRC BB/D006473/1. Floris Geerts is a postdoctoral researcher of the FWO Vlaanderen.

References

[1] D. Barbosa, A. O. Mendelzon, J. Keenleyside, and K. A. Lyons. Toxgene: An extensible template-based data generator for XML. In *WebDB*, 2002.
[2] S. Cho, S. Amer-Yahia, L. Lakshmanan, and D. Srivastava. Optimizing the secure evaluation of twig queries. In *VLDB*, 2002.
[3] B. Choi. What are real DTDs like. In *WebDB*, 2002.
[4] J. Clark and S. DeRose. XML Path Language (XPath). W3C Working Draft, Nov. 1999.
[5] E. Damiani, S. di Vimercati, S. Paraboschi, and P. Samarati. Securing XML documents. In *EDBT*, 2000.
[6] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and scalable filtering of XML documents. In *ICDE*, 2002.
[7] EBI. Gene Ontology. *http://www.geneontology.org/*.
[8] A. Ehrenfeucht and H. P. Zeiger. Complexity measures for regular expressions. *JCSS*, 12(2):134–146, 1976.
[9] W. Fan, C. Y. Chan, and M. Garofalakis. Secure XML querying with security views. In *SIGMOD*, 2004.
[10] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. SMOQE: A system for providing secure access to XML data. In *VLDB*, 2006. Demo.
[11] W. Fan, J. X. Yu, H. Lu, J. Lu, and R. Rastogi. Query translation from XPath to SQL in the presence of recursive DTDs. In *VLDB*, 2005.
[12] M. F. Fernandez, Y. Kadiyska, D. S. A. Morishima, and W. Tan. SilkRoute: A framework for publishing relational data in XML. *TODS*, 27(4):438–493, 2002.
[13] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *TODS*, 29(4):752–788, 2004.
[14] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.
[15] IBM. DB2 XML Extender. *http://www-3.ibm.com/software/data/db2/extended/xmlext/*.
[16] C. Koch. Efficient processing of expressive node-selecting queries on xml data in secondary storage: A tree automata-based approach. In *VLDB*, 2003.
[17] R. Krishnamurthy, V. Chakaravarthy, R. Kaushik, and J. Naughton. Recursive XML schemas, recursive XML queries, and relational storage: XML-to-SQL query translation. In *ICDE*, 2004.
[18] R. Krishnamurthy, R. Kaushik, and J. Naughton. Efficient XML-to-SQL query translation: Where to add the intelligence. In *VLDB*, 2004.
[19] M. Marx. XPath with conditional axis relations. In *EDBT*, 2004.
[20] Microsoft. XML support in microsoft SQL server 2005, 2005. *http://msdn.microsoft.com/library/en-us/dnsql90/html/sql2k5xml.asp/*.
[21] Oracle. Oracle Database 10g Release 2 XML DB Technical Whitepaper. *http://www.oracle.com/technology/tech/xml/xmldb/index.html*.
[22] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *VLDB*, 2001.
[23] Xerces and Xalan. http://xml.apache.org.
[24] S. Yu. Regular languages. In *Handbook of Formal Languages*, volume 1. Springer, 1996.