A Uniform Approach for Querying Large Tree-structured Data through a Mediated Schema

Claude Delobel[1] and Marie-Christine Rousset[2]

[1] INRIA, Rocquencourt,BP 105, 78153 Le Chesnay, France
[2] University of Paris-Sud, L.R.I, Building 490,
91405, Orsay Cedex, France
mcr@lri.fr , home page : http://www.lri.fr/people/mcr.html

Abstract. We are interested in defining and querying large and highly heterogeneous database repository for XML data and their DTDs. We consider a very simple data model with data trees and tree types for describing the data sources and a very simple query language based on tree queries with boolean conditions. The main components of the data model are a mediated schema as a view of a set of concrete tree types and a mapping expressing the connection between the mediated schema and the concrete tree types. We provide a general tree query semantics and a characterization of tree queries as composition of branch queries. We show also that we can pre-evaluate queries by storing some specific meta-information related to mapping information. We consider also the problem of query containment for tree queries and we propose some policies of relaxation for queries without answer. The research presented here was motivated by the Xyleme project at INRIA, whose objective is to develop a data warehouse for Web XML documents.

1 Introduction

The rapid growth of structured documents (e.g., XML pages) available online raises the need for building integration systems that provide a uniform access to multiple and heterogeneous tree-structured data. A data-integration system enables users to pose queries through a *mediated schema*, thus freeing them from having to interrogate each source separately, and to deal with the heterogeneity of their schematas. The *mediated schema* is intended to reconcile the schemas sources into a uniform virtual view. To answer queries, the data-integration system uses *mappings* between the mediated schema and the schemata of the data sources in order to translate a user query into a set of queries on the data sources. In the recent years, considerable research has been done about data integration for structured (relational or object-oriented) data sources [15, 14, 12, 10, 3], but also for semistructured data sources [8, 11]. In particular, the Xyleme project [17, 2] is building a warehouse for storing all the XML documents of the web, and for providing querying facilities based on tree-structured mediated schemata [9]. In Xyleme, the choice of a simple tree structure for mediated

schemata results in a uniform modeling setting that makes possible the web scaling up of the approach. In particular, the mappings between a mediated schema and the data schematas map paths of the (labeled) tree modeling the mediated schema to paths of (labeled) trees summarizing DTDs (i.e., the data schemata). Those mappings may be numerous and they are semi-automatically generated [16].

In this paper, we elaborate on the Xyleme tree query language with boolean conditions for which we provide a declarative model-theoretic semantics, a way of checking query containment, and a characterization of tree queries as a composition of branch queries. We then propose a method for pre-evaluating queries at compile time by storing some specific meta-information about the mapping into map translation tables. These map translation tables summarize the set of all the branch queries that can be generated from the mediated schema and the set of all the mappings. Based on these map translation tables, we provide an efficient algorithm for translating a query into queries against the data sources. Lastly, we propose different operators and strategies for relaxing queries which, having an empty map translation table, will have no answer if they are evaluated against the data. In a data integration setting, since the schemata of the data sources are hidden to end users, queries with no answer may happen quite frequently. Relaxing such queries is interesting because the answers to relaxed queries have good chance of bringing information that is relevant to the original query.

The paper is organized as follows. The tree data model and the tree queries are described in, respectively, Section 2 and Section 3. In Section 4, we propose a method for pre-evaluating the queries, based on the construction of map translation tables. In Section 5, we address the problem of query relaxation. Finally, Section 6 is a short conclusion.

2 Tree Model

2.1 Terminology and Basic Notations

Our formal model abstracts XML document as labeled trees. Our abstraction simplifies real XML documents in several ways. For example, the model does not distinguish between attributes and elements. Also our trees are unordered. We use also a simplified version of DTDs that we call tree type.

We consider: an infinite set $\mathcal{N}$ of nodes, a finite set of labels, and a set $\mathcal{D}$ of data values. We denote labels by $E_1, E_2, ...$ or by `A, B, C,...`, nodes by $u, v, u_1, ...$ For simplicity we assume that the values are strings of characters.

Definition 1. *Labeled trees A* **labeled tree** *is a pair $\langle t, \nu \rangle$ where (i) t is a finite* **tree** *whose nodes are in $\mathcal{N}$, (ii) ν is a* **labeling function** *that assigns a label to each node in t.*

The labels play the role of the begin-end tag in XML data models. We use the current terminology about trees: root, children, descendant, leaf, subtree,

etc. We also use some functional notations such as: $root(t)$ returns the root node of the tree t, and $children(u)$ returns the set of nodes that are children of node u.

Definition 2. *Access path and path support Let $T = \langle t, \nu \rangle$ be a labeled tree. If u is a node in t, its* **access path**, *denoted $path(u)$, is the sequence of labels associated with the path from the root of t to u* [1]. *The* **path support** $\mathcal{L}_T$ *of T is the set of all the access paths of its nodes.*

For manipulating paths, the notation / is used to denote concatenation. Let p be a path $E_1/\ldots/E_n$, and p', a path $E'_1/\ldots/E'_k$, p/p' is the path $E_1/\ldots/E_n/E'_1/\ldots/E'_k$. The empty path is denoted ϵ. Given two paths p_1 and p_2, with the same root, we denote by $p_1 \lhd p_2$ the fact that p_1 is a *sub-path* of p_2 (or p_1 is a *prefix* of p_2).

Definition 3. *Tree homomorphisms Let $T = \langle t, \nu \rangle$ and $T' = \langle t', \nu' \rangle$ be two labeled trees, a mapping h from nodes in t to nodes in t' is a* **strict** *(resp.* **weak***)* **structural homomorphism** *iff: (i) h preserves the roots: $root(t') = h(root(t))$, and (ii) h preserves the structure: whenever v is a child of u (resp. descendant), $h(v)$ is a child of $h(u)$ (resp. descendant). h is a* **type homomorphism** *(strict or weak) iff h is a structural homomorphism which preserves the labels: $\forall u$ in t: $\nu(u) = \nu'(h(u))$.*

2.2 Data Tree and Tree Type

Definition 4. *Data trees A* **data tree** *is a triple $\langle t, \nu, v \rangle$ where $\langle t, \nu \rangle$ is a labeled tree and v a partial* **value function** *assigning a data value from $\mathcal{D}$ to some nodes. Its* **path support** *is the path support of $\langle t, \nu \rangle$.*

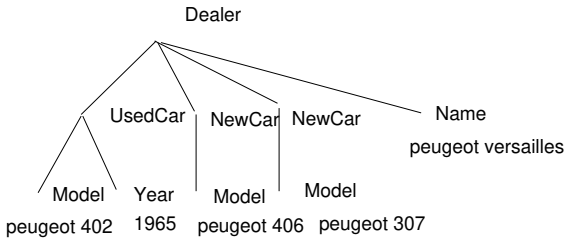

Fig. 1. A data tree

Figure 1 gives an example of a data tree. To make the distinction between the labels and the values we use upper case characters for the labels and lower case for the values. The path support is composed of :

[1] if r is the root of t, $path(r) = \nu(r)$.

`Dealer`, `Dealer/UsedCar`, `Dealer/UsedCar/Model`, `Dealer/UsedCar/Year`, `Dealer/NewCar`, `Dealer/NewCar/Model`, `Dealer/Name`.

For each node u of a data tree, we need to define its value closure, which is the union of the values appearing in the subtree rooted in u.

Definition 5. *Value closure Let $\langle t, \nu, v \rangle$ be a data tree. For each node u in t, its* **value closure**, *denoted $v^\star(u)$, is inductively defined as follows: if u is a leaf, then $v^\star(u) = \{v(u)\}$ if $v(u)$ is defined, and $v^\star(u) = \emptyset$ otherwise, else $v^\star(u) = \bigcup_{v \in \{u\} \cup children(u)} v^\star(v)$.*

In Figure 1, the value closure of the node labeled by `UsedCar` is: {`peugeot402`, `1965`}.

Schematas of data trees are defined by tree types.

Definition 6. *Tree types and instances of a tree type A* **tree type** *is a labeled tree such that no node has two children labeled the same. Let $d = \langle t, \nu, v \rangle$ be a data tree and T be a tree type, d is an* **instance** *for T iff there exists a strict type homomorphism from $\langle t, \nu \rangle$ to T.*

Three tree types are shown in Figure 2. The data tree of Figure 1 is an instance of the tree type of Figure 2(b).

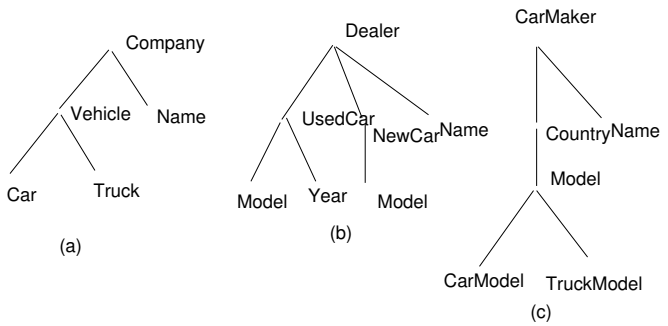

Fig. 2. Example of tree types

2.3 Database Model

A *tree database* consists of a large set of data trees related to a given domain and associated with a large variety of tree types. Let $\mathbf{C}$ be the set of tree types that are models of the data trees stored in the database. These tree types are called **concrete tree types**. The **path support** of $\mathbf{C}$ is defined as follows: $\mathcal{L}_{\mathbf{C}} = \bigcup_{C \in \mathbf{C}} \mathcal{L}_C$.

An *integrated tree database schema* is composed of three parts: a set of *concrete tree types*, with their path supports, representing the data schematas, an *abstract tree type* with its abstract path support, representing the mediated schema,

and a *mapping* between abstract and concrete path supports. An *abstract tree type* is a tree type whose labels are terms of an abstract vocabulary, distinct from the concrete vocabulary used in the labeling functions of the tree types defining the data schematas.

We will use the tree type in Figure 2(a) as an example of an abstract tree type, defined on the abstract vocabulary composed of the terms: `Company, Vehicle, Name, Car, Truck`. The abstract vocabulary has been chosen to unify the concrete vocabulary and represent a specific domain of interest. Its internal nodes are labeled with terms that represent either concepts (`Vehicle`) of the domain, or properties of those concepts (`Name`). This abstract tree will be used to present an integrated view of the concrete types presented in Figure 2(b) and (c).

Users queries are issued with the vocabulary defined in the abstract tree type. To be executable, queries must be translated into queries expressed in terms of the labels used in the data trees, which are the labels appearing in the tree types defining their schematas. Query translation relies on a *mapping relation* which connects paths of an abstract tree type to paths of concrete tree types.

Definition 7. *Mapping relation between abstract and concrete path support Let* $\mathbf{A}$ *be an abstract tree type related to a set* $\mathbf{C}$ *of concrete tree types. Let* $\mathcal{L}_{\mathbf{A}}$ *and* $\mathcal{L}_{\mathbf{C}}$ *be the path supports of* $\mathbf{A}$ *and* $\mathbf{C}$ *respectively. A* **mapping relation** M *between* $\mathbf{A}$ *and* $\mathbf{C}$ *is a binary relationship between* $\mathcal{L}_{\mathbf{A}}$ *and* $\mathcal{L}_{\mathbf{C}}$ *:* $M \subset \mathcal{L}_{\mathbf{A}} \times \mathcal{L}_{\mathbf{C}}$*. We will denote a pair* (l, l') *in* M *by:* $l \leftrightarrow l'$*.*

Example 1. Here are some examples of meaningful mapping elements between the abstract tree type given in Figure 2 (a) and the concrete tree types given in Figure 2 (b) and Figure 2 (c):

```
Company <-> Dealer
Company/Vehicle/Car <-> Dealer/UsedCar/Model
Company/Vehicle/Car <-> Dealer/NewCar/Model
Company/Name <-> Dealer/Name
Company <-> CarMaker
Company/Vehicle/Car <-> CarMaker/Country/Model/CarModel
```

The definition of the mapping relation M given above is arbitrary. This means, for instance, that from one path in the abstract tree type, we could have more than one image in the mapping relation. We can summarize the elements of the data model in the following definition.

Definition 8. *Database instance Let* $\mathbf{A}$ *be an abstract tree type,* $\mathbf{C}$ *be a set of concrete tree types, and* M *be a mapping relation. A* **database instance** I *of the integrated schema* $S = (\mathbf{A}, \mathbf{C}, M)$ *is a set of data trees such that: for all* $d \in I$ *there exists* $C \in \mathbf{C}$ *such that* d *is an instance of* C*.*

Example 2. Here is another example that we will use all over the paper. The database schema is composed of an abstract tree type and a concrete tree type as given in Figure 3 with the following mapping relation:

```
A <-> A'          A/B/C <-> A'/B'/C'
A/B <-> A'/B'     A/B/C <-> A'/F'
A/B <-> A'/F'     A/B/D <-> A'/F'/D'
```

An instance of the database is also given in Figure 3 with only one data tree.

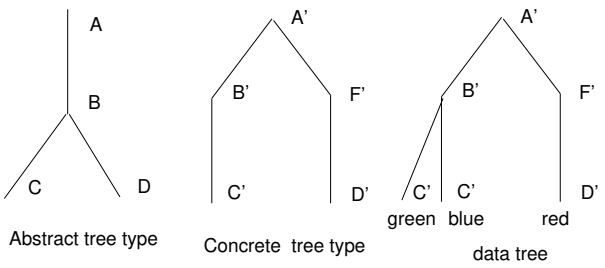

Fig. 3. An abstract tree type, a concrete tree type and a data tree

3 Tree Queries

In the data model presented in this paper, end users do not have a direct access to the instances of the database through the query processor. Instead, they pose their queries against the mediated schema which has the form of an abstract tree type. Such queries are called *abstract* queries as opposed to *concrete* queries that will be generated automatically by the system and processed by the database query processor. The translation of an abstract query into a concrete one only depends of the mapping relation. We define a very simple query language, similar to the *prefix-selection query* language presented in [1]. Although very limited, we claim that it is often sufficient in practice. It is adapted to final users who need to get a visual representation of their queries in the abstract tree. Only paths in the abstract tree are used to formulate the query. In some sense, it is similar to the language QBE for relational database. The simplicity of the language is also really needed because final users do not know the structure of the data trees that are stored. They only have an abstract view of it. This approach is different from those developed in the framework of XML proposals for query languages where the objective is more to write applications on XML databases.

3.1 Well Formed Tree Queries

Queries are obtained by putting some conditions on some nodes, and pointing out some selected nodes, in *tree patterns*. The definition of an abstract or a concrete query is the same because a tree pattern expresses the conformity with an abstract tree type or with a concrete tree type.

Definition 9. *Tree pattern Let* **A** *(resp. C) be an abstract tree type (resp. concrete tree type), a* **tree pattern** *conform to* **A** *(resp. C) is a labeled tree T such that there exists a strict type homomorphism from T to* **A** *(resp. C).*

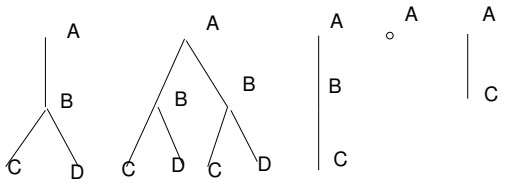

Fig. 4. Query pattern examples

Figure 4 gives some examples of tree patterns based on the abstract tree given in Figure 3. The first pattern is isomorphic to the abstract tree, in the second the same tree is duplicated, the third one is a branch of the abstract tree, such patterns are called *branch patterns*, the fourth is a pattern reduced to the root, and the last one is not a pattern conform to the abstract tree type because the node `C` is not a direct child of `A`.

Giving the tree pattern is not enough to define completely a query, we must add conditions to some nodes and we must point out the selected nodes. We only allow boolean conditions where the atoms are strings and the operator are: **and, or, not**. We can now define the notion of tree queries.

Definition 10. *Tree query An* **abstract** *(resp.* **concrete** *)* **tree query** Q *conform to an abstract tree type* **A** *(resp. a concrete tree type C) is a 4-tuple* $\langle t, \nu, \omega, \gamma \rangle$*, where* $\langle t, \nu \rangle$ *is a tree pattern conform to* **A** *(resp. C), and* ω *is a partial function from the nodes to* $\{S, C, SC\}$*, where* S, C, SC *stands for respectively "select", "conditional" or "select and conditional" node, and* γ *is a partial function assigning a boolean condition to the conditional nodes. For a tree query* Q*, we will denote* $S(Q)$ *and* $C(Q)$ *respectively, the sets of its selected nodes and the set of its conditional nodes* [2]*.*

Example 3. Figure 5 gives three examples of abstract tree queries, the first one is associated with the first pattern in Figure 4. The interpretation of the query is: find all the data trees rooted at `A`, such there exists a subtree rooted in `B` which contains two branches `C` and `D`, the subtree rooted in `C` must contain the string "blue" and the subtree rooted in `D` must contain the string "red"; for those data trees returns the identity of subtrees rooted in `C` and `D`. The last tree query is an example where we have a non atomic boolean condition `blue and red`. This boolean condition expresses that the corresponding data subtree must contain the two strings `blue` and `red`. When talking about the interpretation of

[2] $S(Q)$ and $C(Q)$ are not necessarily disjoint

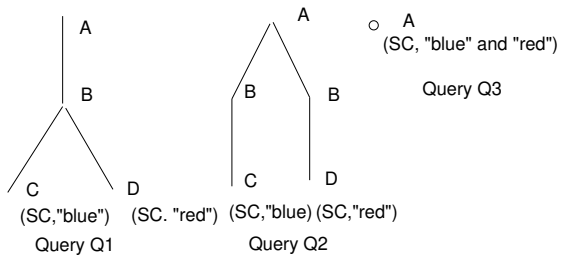

Fig. 5. Example of tree queries

an abstract query, we must understand that these data trees exist only virtually. As explained in the database model, the real data trees are associated with concrete types.

```
Q1: A ( B ( C? blue, D? red ) )          select  x, y
Q2: A ( B ( C? blue, B ( D? red ) ) )    from d in A and z in d/B and x in z/C and y in z/D
Q3: A? blue and red                      where x contains blue and y contains red
```

Fig. 6. Different syntaxes for tree queries

Figure 6 presents other syntaxes for writing tree queries in an equivalent way and which may be used. A linear syntax is based on the parenthesis notation for trees. If a node label `X` is used we can adorn the symbol `X` with ? for a selected node and write the condition after. Such tree queries can also be expressed with SQL-like expressions, the figure illustrates only the expression for query Q_1. One can remark that in the SQL expression we are using variables ranging over path expressions, while in a tree query these variables do not appear explicitly.

In fact, some nodes in the tree query are crucial. They are nodes with conditions, nodes that are selected, or "join" nodes.

Definition 11. *Necessary nodes in a tree query A node in a tree query is* **necessary** *iff it is the root, a selected node, a conditional node or a join node, i.e. a node that has, at least, two distinct descendants which are necessary nodes. For a tree query Q, we denote $N(Q)$ the set of its necessary nodes.*

The necessary nodes can be partially ordered by the relation $\prec$. Let u and v be two necessary nodes: $u \prec v$, iff u is a descendant of v and there is no necessary node w such that $u \prec w \prec v$. The *path support of a query* is the path support of the tree pattern of the query restricted to the necessary nodes.

Finally, we define the *conditional closure* of a node u in a tree query, which summarizes the conditions associated with all the conditional nodes of the subtree rooted at u.

Definition 12. *Conditional closure of a node in a tree query Let $Q = \langle t, \nu, \omega, \gamma \rangle$ be a tree query. For each node u of Q, we inductively define its* **conditional closure** *$\gamma^\star(u)$ as follows: (i) if u is a leaf: if u is a conditional node then $\gamma^\star(u) = \gamma(u)$ and $\gamma^\star(u) = true$ otherwise, (ii) else: $\gamma^\star(u) = \bigwedge_{v \in \{u\} \cup children(u)} \gamma^\star(v)$.*

3.2 Tree Query Semantics

We first define the semantics of concrete queries. The semantics of an abstract query is then defined through its translation into concrete queries. A concrete query is interpreted, through *valuations*, against the data trees that are stored into the database.

Definition 13. *Valuation of a concrete query Let I be an instance of an integrated schema $(\mathbf{A}, \mathbf{C}, M)$ and let $Q = \langle t, \nu, \omega, \gamma \rangle$ be a concrete tree query conform to $C \in \mathbf{C}$, and $d = \langle t', \nu', v \rangle$ be a data tree in I instance of C. A* **valuation** *of Q w.r.t d is a mapping σ_d from necessary nodes in t to nodes in t' and such that: (i) σ_d is a weak type homomorphism from $\langle t, \nu \rangle$ into $\langle t', \nu' \rangle$, and (ii) σ_d satisfies the conditions of Q: for every conditional node, $v \in C(Q)$, $\gamma^\star(v)$ is evaluated to true on $v^\star(\sigma_d(v))$.*

The boolean condition $c = \gamma^\star(v)$ is evaluated against a set of strings $D = v^\star(\sigma_d(v))$ as usual: an atomic condition c is evaluated to `true` iff $c \in D$, c **and** c' is evaluated to `true` iff c and c' are evaluated to `true`, etc.

Given $\bar{v} =< v_1, \ldots, v_k >$ the necessary nodes of Q, we denote the valuation σ_d which maps each node v_i to a node u_i of the data tree d by the set: $\{v_1 \rightarrow u_1, \ldots, v_k \rightarrow u_k\}$. We use the shortcut $\bar{v} \rightarrow \bar{u}$, where $\bar{u} =< u_1, \ldots, u_k >$, and we denote $\sigma_d(\bar{v})$ the image of $\bar{v}$ under σ_d.

Definition 14. *Answers set of a concrete query Let Q be a concrete tree query, and let $\bar{x}$ be the set of its selected nodes. Let I be an instance of an integrated schema $(\mathbf{A}, \mathbf{C}, M)$. The set of answers of Q against I is a set of tuples of data trees defined as follows:*

$$Q(I) = \{\sigma_d(\bar{x}) \mid d \in I \wedge \sigma_d \, is \, a \, valuation \, of \, Q\}$$

.

To be evaluated, an abstract query has to be translated into concrete queries.

Definition 15. *Tree query translation Let $Q = \langle t, \nu, \omega, \gamma \rangle$ be an abstract tree query conform to $\mathbf{A}$, and $Q' = \langle t', \nu', \omega', \gamma' \rangle$ be a concrete tree query conform to C, Q' is a* **translation** *of Q iff there exists a weak structural isomorphism h from necessary nodes of Q onto t' such that: (i) h preserves the selection and conditional nodes, $\forall u \in N(Q) : \omega'(h(u)) = \omega(u)$ and $\gamma'(h(u)) = \gamma(u)$, (ii) h preserves the path support, $\forall u \in N(Q) : path(u) \leftrightarrow path(h(u)) \in M_C$, where M_C is the restriction of the mapping relation to the path support of C.*

The pattern of the translated query has exactly the same tree structure ; the only difference is that the labels are related to concrete tree types. Furthermore, only the necessary nodes of Q have an image in Q'. An important property which results from the definition is that for two necessary nodes u and v in $N(Q)$ such that $u \prec v$, their images by h must satisfy that $path(h(v))$ is a prefix of $path(h(u))$, i.e.,: $path(h(v)) \lhd path(h(u))$.

An abstract query can have more than one translation, depending on the mapping relation M_C. It may also be possible that an abstract query has no translation. The translation process must be repeated for all the concrete type C in the database schema.

Finally, the *answer set of an abstract tree query* Q is the union of all the answers of the translated concrete queries Q':

$$Q(I) = \bigcup_{Q' \in \mathcal{T}(Q)} Q'(I)$$

where $\mathcal{T}(Q)$ is the set of all the translations of Q. One can remark that the union of the answer set for the translated queries can be considered as a relational union. For each Q', the type of an element in the answer set is a a tuple of type $[path(v'_1), \ldots, path(v'_n)]$ where the v'_i's are the the necessary nodes of Q'. As Q' is a translation of the abstract query Q the inverse image by the homomorphism h is defined and the inverse type image is $[path(h^-1(v'_1)), \ldots, path(h^-1(v'_n))] = [path(v_1), \ldots, path(v_n)]$ where the v_i's are the necessary nodes of Q. Therefore, all the answer sets have a uniform type.

Example 4. We are considering the abstract queries given in Figure 5 and the database composed of a unique data tree instance d as presented in Figure 3. In general, each abstract path to any necessary node of the query must be converted into a concrete path using the mapping relation. For example, if we consider the abstract tree query Q_1, its necessary nodes, `A`, `B`, `C` and `D` , are such that: `C` $\prec$ `B` and `D` $\prec$ `B`. The abstract paths to the necessary nodes are: `A`, `A/B`, `A/B/C`, and `A/B/D`. They can be mapped to the concrete paths `A'`, `A'/B'`, `A'/B'/C'` and `A'/F'/D'` respectively . However, this does not lead to a translation of the query because `D` $\prec$ `B` and $path$(`B'`) = `A'/B'` is not a prefix of of $path$(`D'`) = `A'/F'/D'`. This means that there is no translation, and thus no answer, for the query Q_1.
If we consider now the abstract query Q_2, its necessary nodes are: `A`, `C` and `D`. The abstract paths to the necessary nodes are: `A`, `A/B/C`, and `A/B/D`. They can be converted respectively into `A'`, `A'/B'/C'` and `A'/F'/D'`. This provides a translation Q'_2 which can then be evaluated against d: there exists a valuation σ_d, where the node labeled with `C'` can be associated with the subtree labeled `C'` with the value `blue` and the other node labeled with `D'` can be associated with `D'` and the value `red`. Therefore, the query Q'_2 has at least this answer.
For the abstract query Q_3 the pattern of the translation Q'_3 is reduced to one node labeled with `A'`. The whole data tree d matches also the query Q'_3 because the data tree rooted at `A'` contains the strings `blue` and `red` by application of the closure valuation principle.

3.3 A Naive Algorithm for Query Translation

Example 4 shows that when we translate an abstract query Q into a concrete one Q', the crucial elements are the paths to necessary nodes in the query Q and their image by the mapping relation. The other elements such as the select and conditional nodes are not modified: they are just rewritten in Q' from Q. For a query Q, a translation relative to a concrete tree type C can be characterized by a **map translation**: $\{v_1 \rightarrow l'_1, ..., v_n \rightarrow l'_n\}$, where the v_i's are the necessary nodes in Q, the l'_i's are concrete paths in the path support of C. It can be shown that a query Q has a translation relative to a concrete tree type C iff there exists a map translation: $\{v_1 \rightarrow l'_1, ..., v_n \rightarrow l'_n\}$ such that: if $v_i \prec v_j$ then $l'_j \lhd l'_i$ and $path(v_i) \leftrightarrow l'_i \in M_C$.

Building map translations can be done by verifying some properties on the *images* of the necessary nodes in the mapping relation M. The image $L_C(v)$ of a necessary node v in the mapping relation M_C restricted to a tree type C is defined as follows:

$$L_C(v) = \{l' \mid path(v) \leftrightarrow l' \in M_C\}$$

The following naive algorithm computes the set of map translations of a given query relatively to a tree type C by building the Cartesian product of the images of the necessary nodes and by selecting those elements that satisfy the property of path extension: if a necessary node u is a descendant of another necessary node v, then the image of v must be a prefix of the image of u.

Algorithm 1: Generating all the map translations of an abstract tree query Q for a concrete tree type C.
input: a n-tuple of necessary nodes $< v_1, ..., v_n >$ for the abstract query, the partial order relation $\prec$ between necessary nodes, and the mapping relation M_C restricted to a concrete tree type C.

1. For each node v_i, compute its image in the mapping relation: $L_C(v_i)$,
2. Select all the elements $(l'_1, ..., l'_n)$ in the Cartesian product $L_C(v_1) \times ... \times L_C(v_n)$ such that: for all pairs (v_i, v_j) if $v_i \prec v_j$ then $l'_j \lhd l'_i$.

output: The set of map translations relative to C.

3.4 Tree Query Containment

We define the notion of *query containment* in a classical way.

Definition 16. *Query containment Let Q_1 and Q_2 be two tree queries defined relatively to the same integrated schema S. Q_1 is* **contained** *in Q_2, denoted $Q_1 \subseteq Q_2$, iff for every instance I of S, the set of answers of Q_1 against I is included in the set of answers of Q_2 against I, i.e: $Q_1(I) \subseteq Q_2(I)$.*

The following proposition provides a constructive characterization of tree query containment. It is similar to the homomorphism property for conjunctive relational query [7].

Proposition 1. *Let* $Q =< t, \nu, \omega, \gamma >$ *and* $Q' =< t', \nu', \omega', \gamma' >$ *be two tree queries.*
If there exists a strict type homomorphism h *from nodes in* t *to nodes in* t' *such that: (i)* h *preserves the necessary nodes:* $h(N(Q)) \subseteq N(Q')$*, (ii)* h *strictly preserves the selected nodes:* $h(S(Q)) = S(Q')$*, and (iii)* h *logically preserves the conditions:* $\forall u \in N(Q), \gamma'^{\star}(h(u)) \models \gamma^{\star}(u)$ [3] *then* $Q' \subseteq Q$*.*

Example 5. The two first queries given in Figure 5 satisfy $Q_1 \subseteq Q_2$.

3.5 Tree Query Decomposition

Since the set of answers has been defined in a relational way, as a set of tuples (Definition 14), we can obtain it by applying project and join operations to the answer sets of some elementary queries. This is the idea underlying the decomposition of a tree query into *branch queries.*

Let $Q = \langle t, \nu, \omega, \gamma \rangle$ be a tree query. We transform it into a new tree query by making all the necessary nodes selected nodes. Let $\langle t, \nu, \omega', \gamma \rangle$ be the new tree query. Let $t_1, \ldots, t_n$ be the branches of t. For each branch t_i in t, we consider the branch query $Q_i = \langle t_i, \nu/t_i, \omega'/t_i, \gamma/t_i \rangle$ where the notation ν/t_i (respectively ω'/t_i, γ/t_i) denotes the function ν (respectively ω', γ) restricted to the nodes on the branch t_i. We say that the tuple of branch queries $< Q_1, Q_2, ..., Q_n >$ is a **branch decomposition** of the query Q. The following proposition establishes the relation that exists between the set of answers of a tree query and the sets of answers of its branch queries. It states in our framework results that have been previously used in query plan optimization for XML [13, 6].

Proposition 2. *Let* Q *be a tree query and* $< Q_1, Q_2, ..., Q_n >$ *its decomposition into branch queries. Let* I *be an instance of a schema* S*. We have:*

$$Q(I) = \pi_{\bar{x}}(\bowtie_{i=1,n} Q_i(I))$$

where $\pi_{\bar{x}}$ *denoted the project operator on the selected nodes* $\bar{x}$ *and* $\bowtie$ *the join operator.*

Example 6. To illustrate the decomposition process, we give some examples based on the abstract tree of Figure 3 and Q_1. Recall that the query Q_1 expressed with the linear syntax is `A(B(C? blue, D? red)).B` is a necessary node and we must turn it into a selected node. We obtain the query `A(B?(C? blue, D? red))` which has two branch queries: `A(B?(C? blue))` and `A(B?(D? red))`.

4 Pre-Evaluation of Abstract Queries

The evaluation of an abstract query is a two step process: first, the abstract tree query has to be translated into concrete tree queries; second, each concrete tree query has to be evaluated on the database. The problem of efficiently evaluating

[3] $\gamma^{\star}$ is defined for all the necessary nodes, not only for conditional nodes

concrete tree queries on data trees (e.g., XML documents) has been previously studied (e.g., [17, 2]) and is out the scope of this paper. We focus here on the translation process, which may be very costly due to the large size of the mapping relation. In [9], the translation algorithm implemented in Xyleme is presented. This algorithm is executed at query time and processes globally the mapping relation to find all the translations of an abstract query. In this section, we propose a pre-evaluation step which makes it possible to pre-compute *at compile time* all the map translations of all the possible branch queries that can be issued from a given abstract tree type. These map translations are encoded and efficiently stored in *map translation tables* (Section 4.1). At query time, the translation process of branch queries is straightforward and simply consists in selection operations on the map translation tables. The translation of general queries exploits the decomposition of queries in their branch queries (Proposition 2) and can be done by join and select operations on the map translation tables (Section 4.2) .

4.1 Map Translation Tables

To each node v of an abstract tree, we associate a relational table $R(E_1, ..., E_n)$, in which we will encode all the branch queries having $path(v) = E_1/.../E_n$[4] as tree pattern and v as necessary node. There are 2^{n-2} such branch queries, varying the necessary nodes on the path (the root and the last node of the path being necessarily necessary nodes). Each row in $R(E_1, ..., E_n)$ will encode map translations $\{v_{i_1} \rightarrow l'_{i_1}, ..., v_{i_k} \rightarrow l'_{i_k}\}$ of branch queries whose necessary nodes are among $v_{i_1}, \ldots, v_{i_k}$, by assigning the value l'_{i_j} to each attribute E_{i_j}, and by assigning the null value "$-$" to the attributes E_i that do not correspond to a necessary node v_{i_j}. For example, a 5-tuple over $E_1, E_2, ..., E_5$ with the following value $< A', -, A'/B'/C', -, A'/B'/C'/D'/E' >$ encodes the translations of 2 branch queries where the necessary nodes are either E_1, E_3 and E_5 or E_1 and E_5. Furthermore, if the relational table contains only this tuple, there is no possible translation for any branch query having E_2 or E_4 as necessary nodes.

Example 7. Figure 7 illustrates the map translation tables corresponding to the abstract tree given in Figure 3 and the related mapping relation. We can remark that for the abstract node `C` the table $R(A, B, C)$ has two tuples: the first one corresponds to a map translation for a branch query where the necessary nodes can be `A, B` and `C`, the paths `A, A/B, A/B/C` are respectively translated into `A', A'/B', A'/B'/C'`. The second tuple has a null value; the interpretation of this tuple is the following: if we consider a branch query where the necessary nodes are only `A` and `C` then there is a translation where the path `A` can be translated into `A'` and the path `A/B/C` into `A'/F'`. For a branch query with the necessary nodes `A` and `C` then we have two possible translations, one given by the first tuple and the other by the second tuple.

[4] We assume without loss of generality that all the labels of an abstract path are distinct.

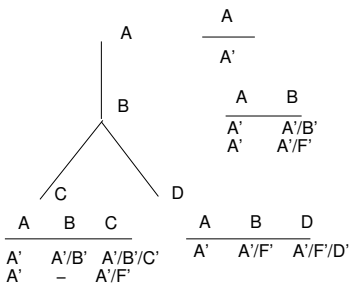

Fig. 7. Map translation tables for an abstract tree

Algorithm 2 describes the way a map translation table R_C can be generated, from the restriction M_C to a concrete tree type C of the mapping relation. It is an efficient implementation of Algorithm 1 when one only consider branch queries. The algorithm provides the map translations for a given tree type C. If we want all the map translations, we have to repeat the process for all C's in the database schema. The map translation table $R(E)$ is the union of R_C's for all C's. The algorithm uses temporary tables U_p of arity p. They progressively store the partial map translations.

Algorithm 2: Generation of a map translation table $R_C(E_1, ..., E_n)$.
input: The mapping relation M_C.

1. Initialize $U_1 \leftarrow L_C(v_n)$
2. for $p = 1$ to $n - 1$ do
   $U_{p+1} \leftarrow \emptyset$
   for each $r \in U_p$ do
   (* r is a row of the table U_p *)
   $null \leftarrow true$
   for each $e \in L_C(v_{n-p})$ do
   if $e \lhd first(r)$ then
   (* $first(r)$ returns the first not null value of row r *)
   insert tuple $< e :: r >$ into U_{p+1}
   $null \leftarrow false$
   endif
   if *null* then insert tuple $< - :: r >$ into U_{p+1} endif
   endfor
   endfor
   endfor
3. $R_C \leftarrow U_n$

output: R_C.

The basic idea of the algorithm is to avoid building the whole Cartesian product of the $L_C(v_i)$'s as in Algorithm 1. For doing so, we start considering in the mapping the images of the longest path ($path(v_n)$), and at each step, we only

consider concrete paths in the image of the mapping relation which are prefixes of those paths previously accepted as valid partial translations.

The space to store all the map translation tables is larger than the space to store only the mapping relation. However, according to some preliminary estimations, the increasing factor should not be greater than 3. It worths it to lose a little in storage to gain a lot at query time. In addition, it is possible to reduce the storage space by compacting more the information in the tables. For example, if we consider a line such as: `<A' A'/B' A'/B'/C'>`, it is not necessary to store the whole paths because each path is a prefix of the next one, we can compact the line to: `<A' B' C'>`.

4.2 Generation of Query Translation

Using map translation tables, it is possible to obtain map translations of branch queries directly at query time. The following algorithm provides a simple and efficient way to obtain all the map translations for general queries. It is based on the decomposition property of a tree query into branch queries.

Algorithm 3: Generation of map translations for a query.
input: a tree query Q.

1. Decompose the tree query into branch queries $Q_1, ..., Q_i$,
2. For each branch query Q_k, $k \in [1..i]$, let $< v_1, ..., v_n >$ be the necessary nodes of Q_i ordered from top to bottom, do:
   (a) Let $E_1/.../E_p$ be the path to v_n, access the relational table $R(E_1, ..., E_p)$,
   (b) Define the table: $R(Q_i) = \sigma_{notnull}(\pi_{\nu(v_1),...,\nu(v_n)}(R(E_1, ..., E_p)))$
3. Join all the tables: $R(Q) = \bowtie_{k \in [1..i]} R(Q_k)$.

output: R(Q) contains all the map translations for Q.

In the algorithm, we compute $R(Q_i)$ which encodes all the map translations of the branch sub-query Q_i. The operation $\sigma_{notnull}$ means that we only select in a table $R(E_1, ..., E_n)$ those tuples that do not contain a null value because a tuple containing a null value for a necessary node cannot lead to a valid translation. If at the end of step 2 we obtain an empty table, this means that there is no possible translation for Q_i and therefore no answer for Q, and Step 3, which consists of taking the join, does not have to be processed.

Example 8. If we consider the abstract query Q_1 we can decompose it into two branch queries over `A, B, C` and `A, B, D` where all the nodes are necessary. The join of the mapping translation tables $R(A, B, C)$ and $R(A, B, D)$ is empty while the tables are not empty. For the query Q_2 the decomposition is the same but the necessary nodes are not the same, the node labeled with `B` is not a necessary node, we use the same map translation tables but we project on `A, C` and `A, D` and the join is not empty.

Applying this algorithm provides an efficient static pre-evaluation of the abstract query. In addition, it provides useful information for relaxing the query that has no translation and thus no answer. In particular, two distinct cases are possible: either there exists some query branches without translation, or the join produces an empty table. Distinguishing those situations is important because it leads to different relaxation strategies, as it will be shown in the next section.

5 Query Relaxation

In this section, we consider the case where our pre-evaluation algorithm has detected that the query Q posed by the user has no answer because it has no translation. The goal is to minimally transform Q into Q' such that Q' has potentially answers, while being semantically close to the initial query Q. Query containment (Definition 16) is a good formal basis for defining weaker queries that are semantically related to the initial query. Proposition 1 provides a formal tool for guaranteeing the generation of such weaker queries. We investigate different strategies for building weaker queries of the original query, for which the translation set is not empty. We explore two kinds of strategies: one which preserves the tree pattern but decreases the number of conditions on the necessary nodes (see Sections 5.2 and 5.3), another one which modifies the tree pattern up to an homomorphism while keeping the conditions (see Section 5.1).

5.1 Unfolding of a Node in the Tree Pattern

The basic idea of unfolding is to modify the tree pattern of the query by duplicating some nodes. Figure 8 illustrates the transformation. The double edge between u and v expresses the fact that u is an ancestor of v; as we are not enumerating all the nodes between u and v we will write $< u..v >$. The unfolding operation is only applied on internal nodes, except the root node.

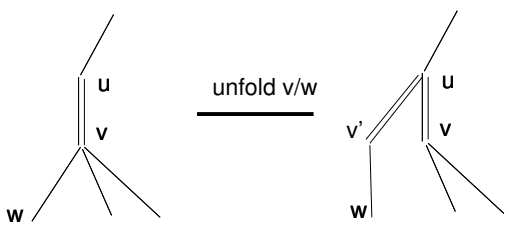

Fig. 8. Unfolding of a tree pattern

More precisely, let $Q = \langle t, \nu, \omega, \gamma \rangle$ be a tree query, and let u, v, w be three nodes in t such that: v is a necessary node ($v \in N(Q)$), w is a direct child of v, and u is the first ancestor of v belonging to $N(Q)$. The *unfolding* of v

relatively to w, denoted $unfold_{v/w}(Q)$ consists of duplicating the node v by adding a new node v', whose child is w. The resulting query Q' contains a new branch $< u..v', w >$, all the labels on this new branch are identical to the ones on $< u..v, w >$, the existing conditions on w are preserved, and the branch $< u..v' >$ is free of condition.

It is easy to remark that $N(Q') \subseteq N(Q)$ and $S(Q') = S(Q)$. Therefore we have a tree homomorphism from Q' to Q, which respects the conditions for weakening the query Q into Q'.

Example 9. Let Q be the tree query: `A(B green(C? blue, D? red))`. The unfolding of `B` relatively to `C` gives: `A(B(C? blue),B green(D? red))`. If we unfold again `B` relatively to `D` we obtain: `A(B(C? blue),B(D? red), B green)`.

One can remark that if we apply successively the unfolding process on a query, in any order, at the end we obtain a tree query where the only joining node is the root of the tree.

5.2 Deletion of a Condition

Let $Q = \langle t, \nu, \omega, \gamma \rangle$ be a tree query, and a node $u \in C(Q)$ which is a conditional node. The deletion of the condition for u results in a new tree query corresponding to the same tree pattern and such that there is no condition associated with u. The only change is therefore on the set of necessary nodes. The new set is $N(Q')$ and we have: $N(Q') \subseteq N(Q)$. If the node u is a leaf of the tree, the deletion of this condition can imply the deletion of the node because it is not needed any more. But if it is also a selected node, we must keep it without the condition. In any case this transformation preserves the tree pattern and the only modification results in decreasing the number of necessary nodes. Therefore, the conditions for query containment of Proposition 1 are satisfied, and we obtain a weaker query.

Example 10. If we consider the query: `A(B green(C? blue, D? red))`, the deletion of the condition on the leaf node `C` gives: `A(B? green (C?, D? red))` while the deletion of the condition on node `B` gives `A(B (C? blue, D?red))`.

5.3 Propagation of a Condition

In the previous transformation the deletion of the condition implies a complete loss of the condition. It is possible to preserve the condition by propagating it to upper nodes. Nevertheless, we must be careful not to create a new necessary node. To avoid that situation we propagate the condition to the first ancestor belonging to the set of necessary nodes.

Let $Q = \langle t, \nu, \omega, \gamma \rangle$ be a tree query, and a node $v \in C(Q)$, i.e. v is a conditional node, the propagation of the condition for v to u (the first ancestor of $v \in N(Q)$) leads to a new tree query with the same tree pattern and with a smaller set of necessary nodes. The new condition on u is the conjunction of the conditions. Therefore, according to Proposition 1, we obtain a weaker query.

Example 11. If we consider the query: `(A rose (B green (C? blue, D? red)))` the propagation of the condition on `B` gives: `(A rose and green (B (C? blue), (D? red)))`. The condition on `B` has been propagated to `A` and a boolean expression has been constructed. The set of necessary nodes is unchanged: `B` is always a necessary node. This new query is weaker than the first one because the condition is not evaluated on data trees rooted at `B` but at `A`.

5.4 Strategies for Relaxation

According to the pre-evaluation algorithm of a query Q, the absence of answers may have two causes: (i) there is at least one branch where the map translation table $R(Q_i)$ is empty, (ii) the join of all tables $R(Q_i)$ is empty. We propose and analyze different policies for relaxation.

- **Empty table**. If we have an empty map translation table for a branch query this means that we have too many necessary nodes on this branch. The only solution is to remove them progressively until we obtain a non-empty table. If we succeed on a node of this branch, we can apply one of the relaxation operations: unfold, delete or propagate, depending of the nature of the node. For example, for a join node we can only apply the unfolding operation while for a conditional node we can apply a propagating or a deleting operation.
- **Empty join**. The join of relation can be done in any order. Nevertheless, the structure of the tree pattern provides a natural ordering for the joining operations. Join first two map translation tables if they have the longest branch in common. If the result is empty we can only apply an unfolding operation on the lower joining node. Nothing can be done if the joining node is the root .

6 Conclusion

The work presented in this paper has been strongly influenced by the Xyleme project [17], whose objective is to develop a huge and highly heterogeneous XML repository. The simple and uniform tree structure for the data, the data sources schemata, and the mediated schema has been a choice for scaling up the approach. In this paper, our first contribution has been to propose a query pre-evaluation step that can be done at compile time, which improves the efficiency of query translation at query time. Our method is based on pre-processing the mappings between the mediated schema and the data schematas, to compute and encode in relational tables all the branch query translations. Our second contribution concerns query relaxation. We have studied several strategies for relaxing queries which have no answer. They are guided by a characterization of query containment that we have obtained for our setting. They exploit the information that has been stored and used for the pre-evaluation phase. This aspect of our work has some connection with a recent work about minimization of tree queries [4]. Though the techniques are different, it is also similar in spirit to the work presented in [5] about query relaxation in PICSEL [12], which is also based on query containment.

Acknowledgments:

We thank all the Xyleme team and particularly Serge Abiteboul, Bernd Aman, Sophie Cluet, Irini Fundulaki, Tova Milo, Pierangelo Veltri and Dan Vodislav.

References

1. S. Abiteboul, L. Segoufin, and V. Vianu. Representing and Querying XML with Incomplete Information. In *Proceedings of ACM PODS*, 2001.
2. V. Aguiléra, S. Cluet, P. Veltri, D. Vodislav, and F. Wattez. Querying the XML Documents on the Web. In *Proc. of the ACMSIGIR Workshop on XML and I. R., Athens*, July 28, 2000.
3. B. Amann, I. Fundulaki, and M. Scholl. Integrating Ontologies and Thesauri for RDF Schema Creation and Medata Querying. *International Journal of Digital Libraries*, 2000.
4. Sihem Amer-Yahia, SungRan Cho, Laks Lakshmannan, and Divesh Srivastava. Minimization of tree pattern queries. In *Proc. of the ACM SIGMOD*, 2001.
5. A. Bidault, C. Froidevaux, and B. Safar. Repairing queries in a mediator approach. In *Proc. of the 14th European Conference on Artificial Conference*, pages 406–410, 2000.
6. P. Buneman, M. Fernandez, and D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *Very Large Data Bases (VLDB) journal*, 9, 2000.
7. A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational databases. In *STOC*, 1977.
8. Vassilis Christophides, Sophie Cluet, and Jérôme Siméon. On Wrapping Query Languages and Efficient XML Integration. In *Proc. of the ACM SIGMOD*, Dallas, Texas, May 2000.
9. Sophie Cluet, Pierangelo Veltri, and Dan Vodislav. Views in Large Scale XML Repository. In *Proc. Int. Conf. on Very Large Data Bases*, 2001.
10. C-web project. `http://cweb.inria.fr`.
11. Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. Declarative Specification of Web Sites with Strudel. *Very Large Data Bases (VLDB) journal*, 9(1):38–55, 2000.
12. F. Gasdoué, V. Lattès, and M-C Rousset. The use of CARIN language and algorithms for information integration: The Picsel system. *Jour. of Cooperative Information Systems*, 2000.
13. J. McHugh and J. Widom. Query Optmization for XML. In *Proc. Int. Conf. on Very Large Data Bases*, pages 315–326, 1999.
14. J. Ordille, A. Levy, and A. Rajaraman. Querying Heteregeneous Information Sources Using Source Descriptions. In *Proc. Int. Conf. on Very Large Data Bases*, pages 251–262, 1996.
15. Y. Papakonstantinou, H. Garcia-molina, and J. Widom. Object Exchange Across Heteregeneous Information Sources. In *ICDE Conf. on Management of Data*, 1995.
16. C. Reynaud, J.-P. Sirot, and D. Vodislav. Semantic Integration of XML Heterogeneous Data Sources. In *Proc. of the 2001 Int. Database Engineering & Applications Symposium*, July 2001.
17. Xyleme. `http://www.xyleme.com`.