Semantic Integration in Xyleme: a Uniform Tree-based Approach

Claude Delobel[a], Chantal Reynaud[a], Marie-Christine Rousset[a], Jean-Pierre Sirot[b], Dan Vodislav[c]

[a] *LRI, Université Paris XI, Orsay*

[b] *Xyleme S.A., 6 rue Emile Verhaeren, Saint-Cloud*

[c] *CNAM/CEDRIC, 292 rue Saint-Martin, Paris*

Abstract

Xyleme is a huge warehouse integrating XML data of the Web. Xyleme considers a simple data model with data trees and tree types for describing the data sources, and a simple query language based on tree queries with boolean conditions. The main components of the data model are a mediated schema modeled by an abstract tree type, as a view of a set of tree types associated with actual data trees, called concrete tree types, and a mapping expressing the connection between the mediated schema and the concrete tree types. The first contribution of this paper is formal: we provide a declarative model-theoretic semantics for Xyleme tree queries, a way of checking tree query containment, and a characterization of tree queries as a composition of branch queries. The other contributions are algorithmic and handle the potentially huge size of the mapping relation which is a crucial issue for semantic integration and query evaluation in Xyleme. First, we 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. Then, 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. Finally, we present a method for semi-automatically generating the mapping relation. [1]

1 Introduction

The rapid growth of structured documents (e.g., XML pages) available on-line raises the need for building integration systems that provide a uniform

[1] This paper is an extension of two previous conference publications [22,12].

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 schema. The *mediated schema* is intended to reconcile the schemata of the 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 [21,20,14,11,3], but also for semistructured data sources [8,13].

In particular, the Xyleme project [25,26,2] is building a warehouse for storing all the XML documents of the web, and for providing querying facilities based on tree-structured mediated schemata [10]. This very wide-area integration of XML sources raises *challenging scalability problems* concerning source heterogeneity, the size of the mediator mapping relation and querying performances. Xyleme provides original solutions to these issues. The system architecture and the design choices are motivated by "web search engine"-like performance requirements, i.e. supporting many simultaneous queries over a Web-scale XML repository. 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. The simplicity of the mapping relation (correspondences between tree paths) eases automatic mapping generation and distributed storage. The query language is intended to enable end-users to express simple Query-by-Example tree queries over the mediated schema. Xyleme optimizes such tree queries to get very efficient response time.

In this paper, we elaborate on two main aspects of Xyleme: the semantic integration and the resulting querying process. Semantic integration is based on a *mapping relation* between paths of the *abstract tree type* modeling the mediated schema and paths of *concrete tree types* serving as schemata for *data trees*. Querying process is centered on a tree query language: users queries are *abstract tree queries* that must be conform to the abstract tree type. Query evaluation is a two step process: using the mapping relation, a *translation* process transforms the abstract tree query into a set of *concrete tree queries*, which are then evaluated against the data trees stored in the repository.

The first contribution of this paper is formal: we provide a declarative model-theoretic semantics for Xyleme tree queries, a way of checking tree query containment, and a characterization of tree queries as a composition of branch queries.

The other contributions are algorithmic and handle the potentially huge size of the mapping relation which is a crucial issue for semantic integration and query evaluation.

First, we 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 algorithm for getting a direct translation of a query into queries against the data sources. Then, 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.

Finally, we present a method for semi-automatically generating the mapping relation. For setting up semantic integration in Xyleme, the main difficulty comes from the very large number of data sources handled by Xyleme. As a consequence, the number of concrete data trees that have to be mapped to the abstract tree type makes impossible that the mapping relation is manually defined by some database administrator. We have therefore studied how to generate the mapping relation as automatically as possible, by combining syntactic, semantic and structural criteria. A prototype has been implemented and evaluated in the cultural domain.

The paper is organized as follows. Section 2 presents an overview of the Xyleme project. The tree-based model and its formal semantics are described in section 3. In section 4, we present 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. The method for semi-automatically generating the mapping relation is presented in section 6. Finally, section 7 is a short conclusion.

2 An Overview of the Xyleme Project

Xyleme is an initiative of INRIA which involves the following participants: the VERSO group of INRIA, the IASI research group of the LRI at the Paris-South university, the Vertigo research group of CNAM in Paris and a research group of the university of Mannheim. This project has been initiated in September 1999. The goal was to look for innovative solutions for Web information retrieval, assuming that the XML language will become a standard for data representation and exchange on the Web.

2.1 Xyleme Architecture

The Xyleme project deals with data integration when data sources are XML documents. All XML documents are stored in a repository. In this way, the system is efficient even when several data issued from distributed sources must be combined to answer queries. A semantic module plays the role of interface between end-users and XML documents which, by definition, come from several sources and are semantically heterogeneous. Figure 1 presents the overall architecture of the system.

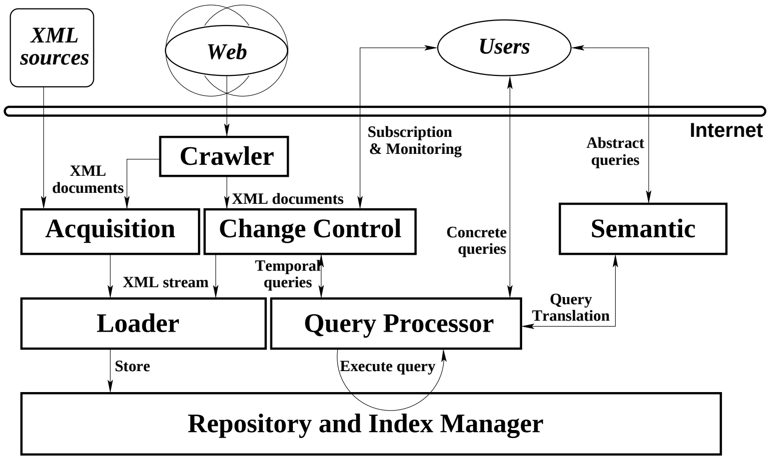

Fig. 1. Xyleme architecture

- The Repository and Index Manager module is the lowest level in Xyleme. It enables to store and index XML documents.
- The Acquisition and Crawler module inspects the Web and collects all the XML documents, which are loaded in the repository by the Loader module.
- The Change Control module is responsible of specific functionalities, such as monitoring of document changes, version management and subscription of temporal queries.
- The Semantic module provides a homogeneous integrated and mediated schema on the heterogeneous XML documents stored in the repository.
- The Query Processor module enables to query the XML repository as a database. In particular, it translates a query in terms of the semantic layer into another one computable on the stored documents.

A general presentation of Xyleme is given in [26]. In this paper we focus on the semantic module and the translation of abstract queries. A presentation of the other modules can be found in [15,17,19].

The aim of the semantic module is twofold: (i) to provide an homogeneous mediated schema over the repository, in order to allow a natural expression of queries for the user without having to be aware of the structure and the syntax of the actual DTDs describing the documents; (ii) to define the connection between this semantic module and the actual data described by the DTDs of the XML documents in the repository, in order to make possible query processing.

The semantic module provides a simple description of domains (e.g. culture, tourism) in the form of an *abstract tree type*. An abstract tree type can be considered as an abstract merger of the DTDs associated with the set of XML documents of some domain. Figure 2 shows a fragment of an abstract tree type in the cultural domain. The label of the root is the domain name (`Culture`). The internal nodes are labelled with terms that represent either concepts of the domain (`Art, Architecture, Cinema`) or properties of those concepts (`Name, Title`).

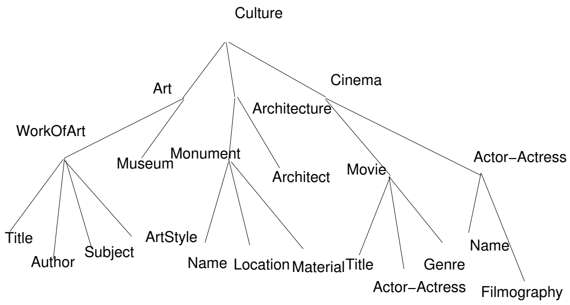

Fig. 2. Fragment of an abstract tree type on culture

There are two main reasons for the choice of representing a mediated schema as a tree. First, it must be compatible with a real XML DTD, because it is intended to be the schema of all the XML documents of the domain. Next, the mediated schema must be simple enough, because it is the support of the visual query interface tool, intended to be used by non-programmer users. The links between a node and its sons in the abstract tree type have no strict semantics. They may correspond to a relation of specialisation between concepts, or to a relation of composition, or simply express different viewpoints on a concept. A link also exists between a concept and a property of it. We do not distinguish between those different types of links. In other words, a node labelled with a term E_2 being a son of a node labelled with E_1 just means that the term E_2 has to be interpreted in the scope of meaning of E_1. Thus, the different occurrences of a same term do not have the same meaning. This is obvious for terms that represent properties (e.g. `Name` can appear different times, and may represent the name of different entities), but it is also the case for terms representing concepts (e.g. `Author` under `Movie` means director, while `Author` under `Painting` means painter).

Such representations are very simple and structured. They enable end-users to build semantic queries from different viewpoints in an easy and natural way, using both the vocabulary and the structure of the abstract tree type. This provides a homogeneous and simple user interface for querying a very large amount of heterogeneous XML documents stored in the repository. Using this interface, end-users must be able to obtain relevant answers for queries.

For this, Xyleme has to identify the XML documents concerning each query issued against the semantic module. This requires the creation and maintenance of a correspondence between the labels of the abstract tree type and the elements of the concrete tree type that describe the structure of the XML documents. This way, each abstract query can be translated into a set of concrete queries over real documents, by simply replacing the abstract labels of the query with the corresponding concrete labels. In Xyleme, the correspondence is stated by a mapping relation between paths of the abstract tree type and paths of the concrete tree types modeling the DTDs stored in the repository. Because of the number of concrete data trees that have to be mapped to the abstract tree type, the mapping relation cannot be manually defined by some database administrator. Some automatic help must be provided. In section 6, we describe a semi-automatic method for generating the mapping relation that has been implemented in Xyleme and experimented.

2.3 The Query Processor

By accessing the abstract tree type through a user-friendly interface, end-users query a single tree structure summarizing many DTDs. The query language is also tree-based.

The query tree on the left in figure 3 models a query in the cultural domain asking the titles of all the films in which the actor Sean Connery is acting. The query tree on the right in figure 3 asks the description of all impressionist works of art. These examples illustrate the basic method for building abstract queries: the user marks in the abstract tree type the nodes to be included in the result (selected nodes are marked with a `S`, e.g. `Title, WorkOfArt`) and the nodes constrained with conditions (conditional nodes are marked with a `C`, e.g. `Actor-Actress, ArtStyle`). The query itself is given by a tree query pattern built from the abstract tree type.

Such queries are defined relatively to the mediated schema and not relatively to the actual DTDs corresponding to the XML data stored in the repository. Therefore they cannot be directly evaluated against that data. The evaluation

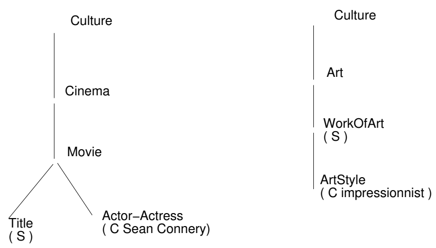

Fig. 3. Example of user queries

of an abstract query is then a two step process. First, the abstract tree query is translated into concrete tree queries by using the mapping relation. Second, each concrete tree query has to be evaluated on the database.

The Xyleme query language is a subset of *XQuery*, the W3C query language for XML data [24]. Unlike XQuery, the Xyleme query language does not allow joins or document transformation. Moreover, the language ignores ordering and random access through an ordinal number to the descendants of a tree node, features that exist in *XPath* [9], which is a part of XQuery. On the other hand, the Xyleme query language is more powerful than XPath, because it enables to extract any part of a subtree, while XPath can only extract a full subtree, identified by its access path. Notice that even if the abstract tree queries do not contain joins, the translation into concrete tree queries may produce joins based on links between concrete documents (see [10]). Notice also that the last version of the Xyleme query language, not presented here, was enriched with additional features such as joins (on values and on links) and simple document transformation.

Due to the large size of the mapping relation in Xyleme, the main querying issue concerns the potential cost of the translation step when executed at query time, since the mapping relation has to be processed globally in order to find all the translations of an abstract query. A naive version of the translation algorithm implemented in Xyleme (described in [10]) is given in Section 3.4. In section 4, we propose a translation algorithm that is made more efficient at query time by adding a pre-evaluation step at compile time.

Another problem which may happen frequently when querying a mediated schema and not the actual DTDs corresponding to the XML documents in the repository, is relative to queries that have no answer. In such cases, it may be interesting to transform a query Q into another one Q' such that Q' has potentially answers relevant to the original query Q. In section 5, we propose different strategies for relaxing queries and obtaining semantically relevant transformed ones.

3 A Uniform Tree Model for Xyleme

3.1 Terminology and Basic Notations

The XML documents we are interested in are valid documents, instances of a DTD that defines their structure. Such real documents are stored in the repository. For example, the DTD `MyDTD` can be defined as follows:

```
<!ELEMENT MyDTD (Film*, Painting*)>
<!ELEMENT Film (Casting, Character*,Title)>
<!ELEMENT Casting (Actor*)>
<!ELEMENT Painting (Title, Author, Museum?)>
<!ELEMENT Actor (#PCDATA)>
............
```

Our formal model abstracts XML documents as labelled 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 where the multiplicity of an element is not considered.

We next present our core framework for a Xyleme database. 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 (Labelled trees) *A* **labelled 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 labelled 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 [2]*. The* **path support** $\mathcal{L}_T$ *of* T *is the set of all the access paths of its nodes.*

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

For manipulating paths, the notation / is used to denote concatenation.
Let p be a path $E_1/\dots/E_n$, and p', a path $E'_1/\dots/E'_k$:
p/p' is the path $E_1/\dots/E_n/E'_1/\dots/E'_k$.
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 *prefix* of p_2.

Definition 3 (Tree homomorphisms) *Let $T = \langle t, \nu \rangle$ and $T' = \langle t', \nu' \rangle$ be two labelled 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: $\nu(u) = \nu'(h(u))$.*

3.2 Data Tree and Tree Type

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

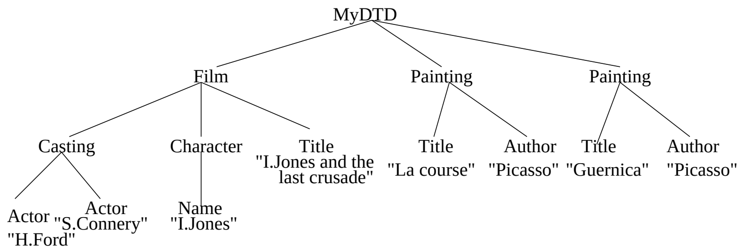

Fig. 4. A data tree

Figure 4 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 :
`MyDTD`, `MyDTD/Film`, `MyDTD/Film/Casting`, `MyDTD/Film/Casting/Actor`...

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

In Figure 4, the value closure of the node labelled by `Film` is: {`H.Ford, S. Connery, I.Jones, I.Jones and the last crusade`}.

Schemas of data trees are defined by tree types.

Definition 6 (Tree types and instances of a tree type) *A* **tree type** *is a labelled tree such that no node has two children labelled the same. Let $d = \langle t, \nu, \upsilon \rangle$ 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.*

A tree type is shown in Figure 5. The data tree of Figure 4 is an instance of the tree type of Figure 5.

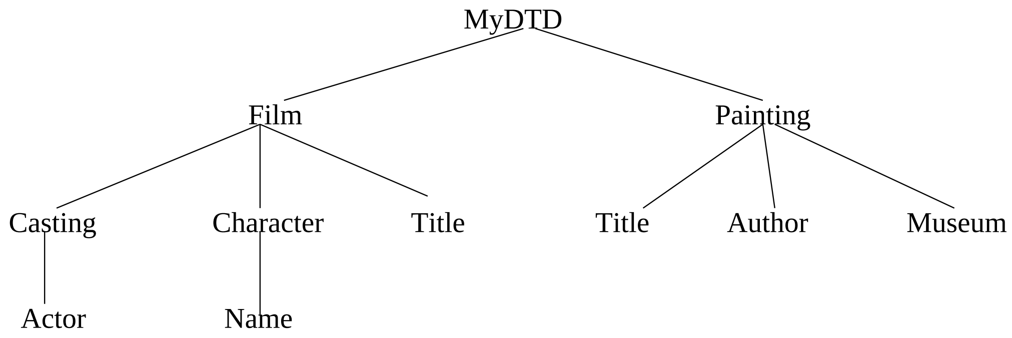

Fig. 5. Example of a concrete tree type

3.3 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 schemas, 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 schemas. In general the abstract vocabulary is distinct from the concrete vocabulary, but it may happen in some cases that we can use the same vocabulary of labels. For instance, the tree type in Figure 2 is an example of an abstract tree type, defined over the vocabulary composed of the terms: `Culture, Art, Cinema, Architecture`, etc. The abstract vocabulary has been chosen to unify the concrete vocabulary and represent a specific domain of interest.

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 schemas. 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) *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 and the concrete tree type given in Figure 5:

```
Culture/Cinema/Movie <-> MyDTD/Film
Culture/Cinema/Movie/ActorActress <-> MyDTD/Film/Casting/Actor
......................
```

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 We will use this example all over the paper. The database schema is composed of an abstract tree type and a concrete tree type as given in Figure 6 with the following mapping relation:

```
Movie <-> Film          Movie/ActorActress <-> Film/Casting
Movie/ActorActress/Name <-> Film/Character
Movie/ActorActress/Name <-> Film/Casting/Actor
Movie/ActorActress/Role <-> Film/Character/Name
```

An instance of the database is also given in Figure 6 with only one data tree.

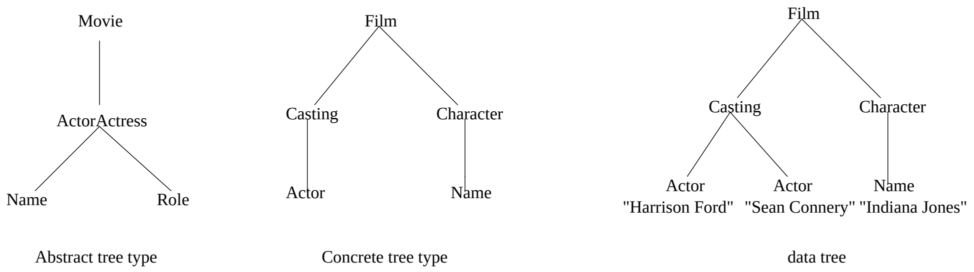

Fig. 6. An abstract tree type, a concrete tree type and a data tree

3.4 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.

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 labelled tree T such that there exists a strict type homomorphism from T to* **A** *(resp. C).*

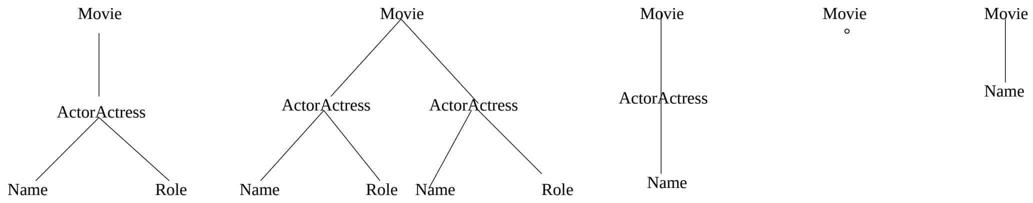

Fig. 7. Tree pattern examples

Figure 7 gives some examples of tree patterns based on the abstract tree type given in Figure 6. The first pattern is isomorphic to the abstract tree type ; in the second pattern, the same tree is duplicated ; the third one is a branch of the abstract tree type: such patterns are called *branch patterns* ; the fourth pattern is reduced to the root, and the last one is not a pattern conform to the abstract tree type where the node labelled by `Name` is not a direct child of the node labelled by `Movie`.

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 operators are: **and, or, not**. We can now define tree queries.

Definition 10 (Tree query) *An* **abstract** *(resp.* **concrete** *)* **tree query** *Q conform to an abstract tree type $\mathbf{A}$ (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 $\mathbf{A}$ (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* [3] .

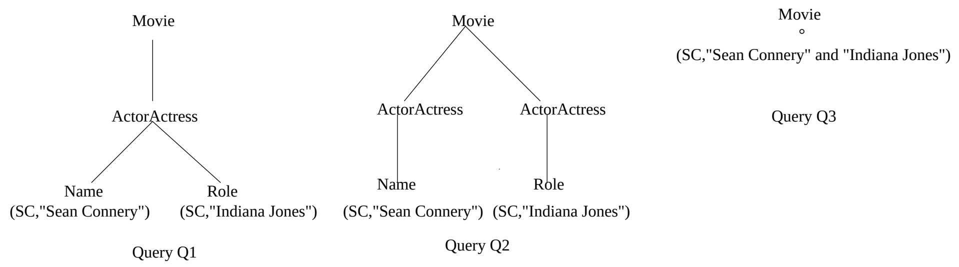

Fig. 8. Example of tree queries

Example 3 Figure 8 gives three examples of abstract tree queries. The first one (Query Q_1) is associated with the first pattern in Figure 7. The interpretation of the query is: find all the data trees rooted at a node labelled by `Movie`, such that there exists a subtree rooted in a node labelled by `ActorActress` which contains two branches, the one rooted in a node labelled by `Name` must contain the string "Sean Connery" and the one rooted in a node labelled by `Role` must contain the string "Indiana Jones"; for those data trees returns the identity of subtrees rooted in the nodes labelled by `Name` and `Role`.

The second one (Query Q_2) has a slightly different interpretation: find all the data trees rooted at a node labelled by `Movie` such that there exists one subtree rooted in a node labelled by `ActorActress` having a branch rooted in a node labelled by `Name` which contains the string "Sean Connery" , and there exists

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

another (possibly distinct) subtree rooted in a node labelled by `ActorActress` having a branch rooted in a node labelled by `Role` which contains the string "Indiana Jones".

The last tree query (Query Q_3) is an example where we have a non atomic boolean condition `"Sean Connery" and "Indiana Jones"`. This boolean condition expresses that the corresponding data subtree must contain the two strings `"Sean Connery"` and `"Indiana Jones"`. When talking about the interpretation of 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 tree types.

Figure 9 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. 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.

Q1: Movie (ActorActress (Name? "Sean Connery", Role? "Indiana Jones"))
Q2: Movie (ActorActress (Name? "Sean Connery" , ActorActress (Role? "Indiana Jones))
Q3: Movie? "Sean Connery" and "Indiana Jones"

Q1: select x , y
from d in Movie and z in d/ActorActress and x in z/Name and y in z/Role
where x contains "Sean Connery" and y contains "Indiana Jones"

Fig. 9. Different syntaxes for tree queries

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 node (called a join node) which has, at least, two distinct descendants that 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: $\gamma^\star(u) = \bigwedge_{v \in \{u\} \cup children(u)} \gamma^\star(v)$.*

3.5 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 $d = \langle t', \nu', \upsilon \rangle$ 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, $u \in C(Q)$, $\gamma^\star(u)$ is evaluated to true on $\upsilon^\star(\sigma_d(u))$.*

The boolean condition $c = \gamma^\star(u)$ is evaluated against a set of strings $D = \upsilon^\star(\sigma_d(u))$ 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: $\{v_1 \to u_1, \ldots, v_k \to u_k\}$. We use the shortcut $\bar{v} \to \bar{u}$, 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:*

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

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 t 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 for necessary nodes: $\forall u \in N(Q)$ $path(u) \leftrightarrow path(h(u)) \in M_C$, 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. 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 tree types 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 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 8 and the database composed of a unique data tree instance d as presented in Figure 6. 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, u_1, u_2, u_3, and u_4, respectively labelled by `Movie`, `ActorActress`, `Name` and `Role` , are such that: $u_3 \prec u_2$ and $u_4 \prec u_2$. The corresponding abstract paths leading to the necessary nodes are: `Movie`, `Movie/ActorActress`, `Movie/ActorActress/Name`, and `Movie/ActorActress/Role`. By the mapping relation given in Example 2, they must be mapped to the concrete paths `Film`, `Film/Casting`, `Film/Casting/Actor` (or `Film/Character`) and `Film/Character/Name` respectively. This suggests a candidate translation based on an homomorphism h such that: $h(u_1)$ is labelled by `Film`, $h(u_2)$ is labelled by `Casting`, $h(u_3)$

is labelled by `Actor` or by `Character`, and $h(u_4)$ is labelled by `Name`. However, in order that candidate translation to be a real translation, since $u_4 \prec u_2$, $path(h(u_2))$, i.e., `Film/Casting`, should be a prefix of $path(h(u_4))$, which must be `Film/Character/Name`. Since `Film/Casting` is not a prefix of `Film/Character/Name`, 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 v_1, v_2 and v_3, respectively labelled by `Movie`, `Name` and `Role`. The abstract paths leading to those necessary nodes are: `Movie`, `Movie/ActorActress/Name`, and `Movie/ActorActress/Role`. Based on the mapping relation, they can be converted respectively into `Film`, `Film/Casting/Actor` and `Film/Character/Name`. This leads to a translation Q'_2 corresponding to an homomorphism h' such that $h'(v_1)$ is labelled by `Film`, $h'(v_2)$ is labelled by `Actor` and $h'(v_3)$ is labelled by `Name`. In this case, h' satisfies the prefix property since: $v_2 \prec v_1$ and $path(h(v_1))$ (i.e., `Film`) is a prefix of $path(h(v_2))$ (i.e., `Film/Casting/Actor`), and $v_3 \prec v_1$ and $path(h(v_1))$ (i.e., `Film`) is a prefix of $path(h(v_3))$ (i.e., `Film/Character/Name`). The resulting concrete query Q'_2 is given in Figure 10.

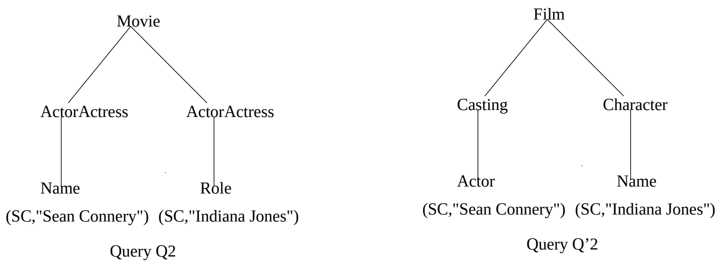

Fig. 10. The abstract query Q_2 and its translation Q'_2

It can then be evaluated against the data tree d: there exists a valuation σ_d, where the node labelled with `Actor` can be associated with the leaf of d labelled by `Actor` and valued by `"Sean Connery"` and the other node labelled by `Name` can be associated with the leaf of d labelled by `Name` and valued by `"Indiana Jones"`.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 labelled with `Film`. The whole data tree d matches also the query Q'_3 because it is rooted at a node labelled by `Film` which contains the strings `"Sean Connery"` and `"Indiana Jones"` by application of the closure valuation principle.

3.6 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.

3.7 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: $\forall u \in N(Q), \gamma'^{\star}(h(u)) \models \gamma^{\star}(u)$ [4] then $Q' \subseteq Q$.*

Example 5 The two first queries given in Figure 8 satisfy $Q_1 \subseteq Q_2$.

3.8 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 [16,6].

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 query Q_1 of Figure 8. Recall that the query Q_1 expressed with the linear syntax is `Movie(ActorActress(Name? "Sean Connery", Role? "Indiana Jones"))`. The node labelled by `ActorActress` is a necessary node and we must turn it into a selected node. We obtain the query:

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

`Movie(ActorActress?(Name? "Sean Connery", Role? "Indiana Jones"))`. It has two branch queries: `Movie(ActorActress?(Name? "Sean Connery"))` and `Movie(ActorActress?(Role? "Indiana Jones"))`.

4 Pre-Evaluation of Abstract Queries

As we have seen in the previous section, 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 concrete tree queries on data trees (e.g., XML documents) has been previously studied (e.g., [25,2]) 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 [10], 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. We show how these map translations can be encoded and stored in *map translation tables* (Section 4.1). As a result, it simplifies the translation process of branch queries that must be done at query time, which is reduced to selection operations on the map translation tables. We show (Proposition 2) how the translation of general queries can exploit the decomposition of queries in their branch queries and can be done at query time 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 $path(v) = E_1/.../E_n$ [5] 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

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

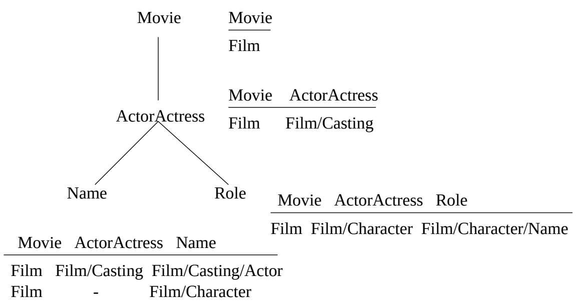

Fig. 11. Map translation tables for an abstract tree

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 11 illustrates the map translation tables corresponding to the abstract tree given in Figure 6 and the related mapping relation. We can remark that the table `R(Movie,ActorActress,Name)` associated with the node labelled by `Name` has two tuples: the first one corresponds to a map translation for a branch query where the necessary nodes can be `Movie, ActorActress` and `Name`, the paths `Movie, Movie/ActorActress, Movie/ActorActress/Name` are respectively translated into `Film, Film/Casting, Film/Casting/Actor`. 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 `Movie` and `Name` then there is a translation where the path `Movie` can be translated into `Film` and the path `Movie/ActorActress/Name` into `Film/Character`. For a branch query with the necessary nodes `Movie` and `Name` then we have two possible translations, one given by the first tuple and the other by the second tuple.

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 optimzation of the naive 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 the 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. As a counterpart, we obtain an immediate translation of branch queries 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 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 `Movie, ActorActress, Name` and `Movie, ActorActress, Role` where all the nodes are necessary. The join of the mapping translation tables `R(Movie, ActorActress, Name)` and `R(Movie, ActorActress, Role)` is empty. For the query Q_2 the decomposition is the same but the necessary nodes are not the same: the node labelled with `ActorActress` is not a necessary node. We use the same map translation tables that we project on `Movie, Name` and `Movie, Role` before applying the join, which now is is not empty: it contains `Film`, `Film/Casting/Actor` and `Film/Character/Name`.

Applying this algorithm provides a 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:

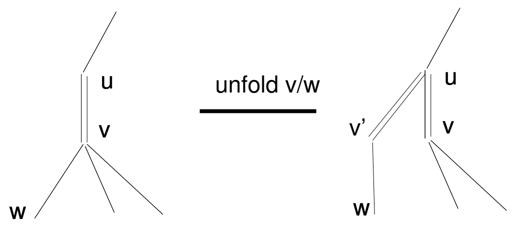

Fig. 12. Unfolding of a tree pattern

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 12 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.

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:
`Movie(ActorActress (Name? "Sean Connery", Role? "Indiana Jones")).`
The unfolding of the node labelled by `ActorActress` relatively to the node labelled with `Name` gives the new following tree query:
`Movie(ActorActress(Name? "Sean Connery"),ActorActress(Role? "Indiana Jones").`

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 anymore. 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: `Movie(ActorActess(Name? "Sean Connery", Role? "Indiana Jones"))`, the deletion of the condition on the node labelled by `Name` gives: `Movie(ActorActress (Name?, Role? "Indiana Jones"))` while the deletion of the condition on the node labelled by `Role` gives `Movie(ActorActress (Name? "Sean Connery", Role? ))`.

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.

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 Automatic Generation of Mappings

We have seen in section 3 and 4 that the mapping relationship is a crucial element of the data model for translating abstract queries into concrete ones. In this section, we address the problem of automatically finding the elements of the mapping relation which states the correspondence between paths of concrete tree types and paths of the abstract tree type. We have to handle heterogeneous names and structures because concrete tree types come from various sources designed by different persons who made personal choices on the names of the labels and on the tree structures. Therefore, automatic mapping generation have to deal with semantic and structure heterogeneity.

Below, we present the method for automatic mapping generation in Xyleme, based on two kinds of criteria: syntactic/semantic and structural. We report the first results of an experiment with a prototype, SAMAG, used to generate mappings for XML documents in the cultural domain.

6.1 Syntactic and Semantic Matching

Automatic mappings generation is based on term matching. The idea is that a mapping is generated only if the abstract and the concrete terms identified by the mapped paths are semantically related. We check two kinds of relations between terms: syntactic and semantic matching.

Semantic relationship	Linked terms
Synonymy	Movie - Film, Picture - Image Each couple of nouns belongs to the same synset
Hyponymy / Hypernymy	Painting - Portrait, because portrait is a kind of painting. Portrait is a hyponym of Painting and Painting is a hypernym of Portrait.
Meronymy / Holonymy	Movie - Scene, because a movie is composed of several scenes. Scene is a meronym of Movie and Movie is a holonym of Scene.
Hyponymy composed with hypernymy	Painting - Dance, because Art is a hypernym of both Painting and Dance

Fig. 13. Semantic relationships used in the SAMAG prototype

Syntactic matching concerns the syntactic inclusion of a term, or of a part of a term, into another one. For example, the concrete term `Actor` is syntactically similar to the abstract term `Actor-Actress`. This method also includes techniques for detecting abbreviations, such as `nb` for `Number`, etc. For more details, see [23].

Semantic matching is based on the use of extra knowledge available through existing ontologies or thesauri. In our experimentation, we chose the WordNet thesaurus [18] .

WordNet groups English nouns, verbs, adjectives and adverbs into sets of synonyms that are linked through semantic relations. Each set of synonyms (synset) represents a concept. A word may belong to several synsets, each one representing a particular sense. Our approach only exploits semantic links between nouns because we chose to use only abstract terms that are nouns. The relationships used in our prototype are listed and illustrated in Figure 13.

Syntactic / semantic matching between words is not enough to obtain precise mappings. The meaning of a term depends on the place it occupies in the concrete tree type, which defines the interpretation context of the word. More precisely, a term may occur several times in the same concrete tree type with different senses. Its meaning may be influenced by the meaning of its predecessors in the tree (e.g. `Name` may be the name of an artist, of a museum, etc.).

Our approach is based on a very simple hypothesis: the terms of a concrete tree type are either object names (e.g. `Painting`), or property names (e.g. `Name`). We consider that property names have a meaning only in the context of the object that they characterise. So, an object, which is characterised by a set of properties, defines the interpretation context of its properties.

In order to decide if a term is an object or a property node, we use heuristics based on the translation of a concrete tree type into a conceptual database schema. We use conceptual schemata built according to the Entity-Relationship formalism, that naturally helps to determine the classes, called entities, in a domain. An entity is a class of instances that share the same properties called attributes. In our approach, we consider that object nodes are similar to entities and that property nodes are similar to attributes. Given a link A - B in the concrete tree type, one heuristics used in the translation process is that B may refer to an entity (according to the above definition) only if B has son nodes, these ones being interpreted as B's attributes. Such a heuristics leads to consider that leaves in a concrete tree type always are property nodes (they can't refer to entities, so they can't be object nodes). Other heuristics are used to qualify internal nodes in a concrete tree type, in order to decide if they are object or property nodes. The exhaustive translation mechanism is described in [23]. However, very often, internal nodes are object nodes.

The basic idea underlying our structural constraints is that a mapping of a property node must be "compatible" with the mapping of the object node it characterizes. In other words, because the property and the object nodes in the concrete tree type are related (the object defines the context of the property), the corresponding abstract terms must be related in the same context-based manner.

Thus, if a concrete path identifying a property name is mapped to an abstract path X, this mapping will be valid only if the object characterized by that property is mapped to a predecessor of X in the abstract tree type. This introduces a context constraint into the mapping generation process, because a property name P of an object O may only be mapped to the subset of abstract terms placed below the abstract terms mapped to O, i.e. the context of the object is transmitted to the property. This also means that mappings for object names must be computed first.

For instance, in the concrete tree type MyDTD, the terms `Title`, `Author` and `Museum` are properties of the object `Painting`. The mappings:

```
Culture/Art/WorkOfArt/Title <-> MyDTD/Painting/Title
Culture/Art/WorkOfArt/Author <-> MyDTD/Painting/Author
```

are valid, because there exists a mapping between the object `MyDTD/Painting` and the abstract term `Culture/Art/WorkOfArt`.

Notice that this context-based constraint may seem too strong in some cases. For instance, the automatic generation algorithm will not find the mapping `Culture/Art/Museum <-> MyDTD/Painting/Museum` if there is no mapping from `MyDTD/Painting` to a prefix of `Culture/Art/Museum`. Imagine that the system does not semantically match `Painting` with `Art`, then the "obvious" mapping `Culture/Art/Museum <-> MyDTD/Painting/Museum` is missed. However, a more careful analysis of this case shows that the missed mapping is not so obvious, because it is not clear if `Culture/Art/Museum` is a museum with paintings.

The structural, context-based constraints for automatic mapping generation significantly improve precision if mappings are correctly found for object nodes. On the other hand, an error at an object node will propagate to all its property nodes.

6.3 Experiment

A prototype, SAMAG (Semi-Automatic MApping Generator), has been implemented in Java. We report here the first results of this work.

The corpus of concrete tree types used in our experiment is related to the cultural domain. An abstract tree type on this domain has been written and mappings have been established manually between this abstract tree type and all the concrete tree types associated with real DTDs of the corpus. We used these manual mappings as a reference set for evaluating the results of the SAMAG system.

SAMAG is a semi-automatic tool. User-validation is necessary each time a syntactic relationship between an abstract term and a concrete term is detected. Indeed, two syntactically similar terms may refer to different concepts and may not have the same meaning. Only a human is actually able to guarantee the semantic consistency of such a mapping.

SAMAG has been implemented as a modular system that can be parameterised. This makes the evaluation of our method easier because results obtained with various parameters from the same inputs can be compared. SAMAG's parameters are:

(1) The semantic relationships used by the system to detect a semantic link between abstract and concrete terms, such as synonymy, hypernymy, holonymy, etc.
(2) The structural constraints to apply on terms of concrete tree types. One can decide to apply or not these constraints. Furthermore, they can be applied only on property nodes in order to consider that property names have a meaning only in the context of the object that they characterise. But they can also be applied on all the nodes of a concrete tree type considering that the meaning of object names is also influenced by the meaning of their predecessors in the tree.

Notice that in all the cases, the system tests the existence of a syntactic relationship between an abstract term and a concrete term.

The results of the automatic mappings generation process are analysed according to (1) the number of relevant mappings among those automatically generated, and (2) the number of mappings manually created that are retrieved by the system. In particular, we study the following categories of mappings:

(1) Mappings that are both manually and automatically generated, such as `Culture/Art/Artist <-> School/Painters_list/Painter`[6]. Such a mapping is generated thanks to a synonymy relationship between `Painter` and `Artist`,
(2) Mappings manually created that are not retrieved by the SAMAG system, such as
`Culture/Art/Artist/ArtMovement <-> School/Name`. Here, there is no semantic relationship in WordNet between `ArtMovement` and `Name`, and the detection of a similarity relationship between an abstract term and a concrete term is a precondition to generate a mapping,
(3) Irrelevant mappings, that are automatically but not manually generated, such as
`Culture/Cinema/Movie/Picture <-> Painting/Image`. Here, there is a synonymy relationship between `Picture` and `Image`, but `Picture` is used in a cinema context and `Image` is used in a painting context. SAMAG is unable to detect such context differences, because `Image` is not a property node. Other irrelevant mappings may be generated because of the polysemy of terms (a word may have different meanings), not handled by

[6] The abstract tree type in Figure 2 is just a part of the one used in this experiment.

Semantic relationships	All	Synonymy	All
Structural constraints	None	All	Property nodes
Generated mappings	683	32	267
Relevant mappings	145	28	115
Percentage of relevant mappings	21%	87.5%	43%
Manual mappings retrieved	74 / 111	18 / 111	52 / 111

Fig. 14. Behavior of the SAMAG prototype

SAMAG.

(4) Mappings automatically generated, that may be considered relevant, even if they have not been found manually, such as `Culture/Art <-> Painting`. The above discussion about the mapping `Culture/Art/Museum <-> MyDTD/Painting/Museum`, demonstrates that several meanings are possible for a term in this context, and that sometimes the semantics is a matter of choice. We also considered as relevant mappings that "have a sense", such as `Culture/Art/Artist <-> School/Painters_list`, but that were considered as useless at manual mappings creation because there is a tag `Painter` below `School/Painters_list`.

The table in Figure 14 gathers some significant statistical results about the mappings generated by the SAMAG system.

6.4 Results analysis

The first column in the table describes results that are obtained when all the semantic relationships in WordNet are used and no structural constraints are applied on terms of the concrete tree types. With this configuration, one obtains the maximum number of mappings that SAMAG can generate. The only condition to generate a mapping is the existence of a similarity relationship (semantic or syntactic) between abstract and concrete terms. In this configuration a lot of mappings are not relevant because the system does not consider the interpretation context of terms.

The second column in the table describes the results when SAMAG is highly constrained. We consider that the meaning of all terms in the tree is influenced by the meaning of its predecessor. Moreover, only one semantic relationship, synonymy, is checked between terms. It is the relation that introduces the lowest distance between terms. In this configuration, almost all the generated mappings are relevant. However, few mappings are generated, so few manual mappings are retrieved.

The last column in the table seems a good trade-off between the number of relevant mappings and the number of mappings automatically retrieved by SAMAG among manual mappings. In this configuration, all the semantic relationships in WordNet are used and only property names are considered to have a meaning in the context of their predecessor in the tree.

The relatively high number of mappings not found by SAMAG (33%) is explained by the fact that many manually created mappings are based on corpus-specific knowledge, e.g. `Culture/Art/WorkOfArt/Artist <-> XML_DOC/Document/Title`, because a document `XML_DOC` is about a painter whose name is contained in the document's title `XML_DOC/Document/Title`. Manually created mappings take advantage of such knowledge, while SAMAG cannot. Other mappings are missed because the concrete terms are non-standard abbreviations instead of significant nouns. The last category of manual mappings missed by the system are mappings that connect an object to some identifying property, e.g. a director and his name, such as in `Culture/Cinema/Director/Name <-> List/Movie/Directed_by/Director`.

The major issue explored in this experiment is the way the interpretation context of terms involved in mappings should be considered in an automated process. We proposed a solution based on structural constraints and the analysis of our results must be viewed as a contribution to better understand how to take context into account.

First, we conclude that structural constraints are necessary to limit the number of generated mappings and to produce relevant ones. The number of irrelevant mappings is too high when no structural constraint is applied (145 relevant mappings among 683 when the configuration in column three gives 115 relevant mappings among 267).

The results also show that applying contextual or structural constraints only on property nodes is not enough restrictive. In the last column of the table in Figure 14, more than 50% of automatically generated mappings are not relevant. An analysis of the irrelevant mappings shows that most of terms involved in bad mappings are objects nodes. Thus, structural constraints must also be applied on object nodes. However, if we apply structural constraints on all the nodes (see results in the second column), the system becomes too constrained and does not generate enough mappings.

The question is then how to define the interpretation context for object nodes. In many cases, the context of an object node is not only given by its predecessor node within the associated path. For an object node, the number of nodes within the path that are significant to define its context of interpretation depends on it, so this number is variable from an object node to another. For instance, the context of interpretation of the object node `Actor` in

`Culture/Cinema/Movie/Actor` is `Movie` whereas the context of interpretation of the object node `Museum` in `Culture/Painting/Oil/Museum` is `Painting, Oil`.

The results also depend on the extra knowledge used to establish semantic relationships between terms. WordNet was used as a resource to look for semantic relationships among terms. This choice has been principally motivated by the fact that data in WordNet are easily accessible for automatic applications. It offers a large coverage of general lexicon in English. However, because a term in natural language has often many meanings, WordNet returns synsets for all the senses of a given term. However, we don't know how to select automatically the right sense of a term, because it directly depends of its interpretation context. So when the system, for instance, asks to WordNet the set of synonyms of a given term, this one, being not domain-specific, takes synonyms of all the senses of the term.

An important remark concerns the influence of mapping generation on query translation. Of course, if a relevant mapping is missed by SAMAG, the query processor will miss the results concerning that mapping. However, additional irrelevant mappings do not necessarily produce irrelevant answers because query translation need all the mappings in the translation of the query tree to be compatible (i.e. to form a tree that respects the descendant relationship among terms). Yet, reducing the number of irrelevant mappings is important for storage saving reasons, which are critical for web-scale domains.

7 Conclusion and Future Work

The work presented in this paper is strongly related to the the Xyleme project [25], whose objective is to develop a huge and highly heterogeneous XML repository. We focus on Xyleme's Semantic and Query Processor modules by proposing a data model and query processing techniques for such heterogeneous XML data.

We defined here a uniform tree structure model for data sources and schemata, for the mediated schema (the abstract tree type and the mappings with the data source) and for Xyleme queries. Based on this model, we proposed two query processing techniques for Xyleme: pre-evaluation of queries at compile time and query relaxation.

The query pre-evaluation step improves the efficiency of query translation at run-time. Query translation is a costly process, because of the large number of data sources and mappings. Our method is based on pre-processing the mappings between the mediated schema and the data schemata, on computing and encoding in relational tables all the branch query translations. At run-time, the query translator will simply combine these pre-computed branches.

Query relaxation is necessary when queries have no answer. We have studied several strategies for relaxing queries, guided by a characterization of query containment 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 [14], which is also based on query containment.

At last, we presented a method for semi-automatic mapping generation, based on syntactic and semantic similarity and on structural constraints. Automatic mapping generation is mandatory in Xyleme, because of the very large number of heterogeneous data schemata. The first experimental results confirmed the need of structural constraints to avoid ambigous interpretation of terms and gave us very usefull hints for improving mapping generation.

Currently, Xyleme implements the tree model presented in this paper for data sources and schemata, the mediated schema and the mappings. The mediated schema model and implementation, seen as a view over the data sources, are described in [10]. As mentioned before, the last version of the query language is richer than the tree queries presented in this paper because it allows joins, node constructors and some additional features such as descendant edges (where the lower node is some descendant of the higher one and not necessarly a direct child), optional nodes, etc. However, the query plan obtained when parsing Xyleme queries is based on a specific algebraic operator called *PatternScan*, which implements tree queries very similar to those presented by our model.

Pre-evaluation of queries is not currently implemented in Xyleme. Query translation uses a very similar algorithm, but concrete branches are computed for each query at run-time. Query pre-evaluation will be implemented and tested in a future version of Xyleme. Moreover, a more general translation algorithm is needed, which allows inversions in the descendant relationship between nodes. This is necessary when the descendant order is not semantically important (e.g. Movie/Actor and Actor/Movie keep the same meaning for terms). Otherwise, a mediated schema containing Movie/Actor will never match a document where the order is Actor/Movie even if semanticaly it should. The pre-evaluation strategy will be reformulated in this new translation context.

Query relaxation is not yet implemented in Xyleme neither. Among the large number of possible relaxation levels for a query, Xyleme will first include a fixed number of levels. The use of descendant edges will allow new relaxation techniques, such as internal node deletion. Among the relaxation levels intended to be implemented first we mention: (i) deletion of all internal nodes that are not root or leaves (flat structure), and (ii) deletion of all the nodes that are not root or selected, with propagation of the condition of a deleted node to the root (keyword search).

Automatic generation of mappings has been implemented. However, to improve the results obtained with SAMAG, another direction is currently experimented and evaluated. Here are the main improvements:

(1) Semantic matching does not dynamically use WordNet anymore. Instead, each abstract tree type node is annotated with a set of semantically related words, used then at mapping generation time. Besides performance improvement (WordNet is not queried at run-time, but at annotation time), this technique has several advantages. The most important is that the user will choose only the relevant words (in the domain-specific context) among those proposed by WordNet (which is domain-independent), and may add other words. Typically, after a mapping generation, the user will realize that some mappings were not found because of some missing terms in the annotation and will add them in order to improve the results.
(2) Context description is also done at the abstract tree type level. The user annotates each node with the list of predecessors that define its interpretation context. The advantage is that the user has a very good knowledge of the domain and can be very precise when describing context. The problems of distinguishing between object and property nodes and of finding the context of an object node disappear.

The first test on the same corpus as the one used in this paper produced 101 mappings, with 99% relevant mappings. The system found 70% of the manual mappings, which is better than the best results of SAMAG (67% when no context is considered) and much better than its results when normally configured (47%). By adding new domain- and corpus-dependent terms to the initial annotation we found 80% of the manual mappings, while the relevance percentage slightly decreased to 97%. We also evaluated that by matching a term with its identifying property (e.g. a movie director with his name), the number of manual mappings found increases to 88%.

The results obtained with the abstract tree type annotation technique are promising. The ratio of relevant mappings is very high and the precision can be iteratively improved. The additional annotation work is reasonable, because it is necessary only on the abstract tree type. Future work will focus on improving precision.

Acknowledgments:

We would like to thank all the Xyleme team for many fruitful discussions and particularly Serge Abiteboul, Bernd Aman, Sophie Cluet, Irini Fundulaki, Tova Milo and Pierangelo Veltri.

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 Intelligence*, 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] J. Clark and S. DeRose (eds). *XML Path Language (XPath).* W3C Recommandation, 1999.
`http://www.w3c.org/TR/xpath`.

[10] Sophie Cluet, Pierangelo Veltri, and Dan Vodislav. Views in Large Scale XML Repository. In *Proc. Int. Conf. on Very Large Data Bases*, 2001.

[11] C-web project.
`http://cweb.inria.fr`.

[12] C. Delobel and M-C. Rousset. A uniform approach for querying large tree-structured data through a mediated schema. In *Proc. of the 2001 Int. Workshop on Foundations of Models for Information Integration*, September 2001.

[13] 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.

[14] 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.

[15] Carl-Christian Kanne and Guido Moerkotte. Efficient storage of xml data. Technical report, University of Mannheim, http://pi3.informatik.uni-mannheim.de/, 1999.

[16] J. McHugh and J. Widom. Query Optmization for XML. In *Proc. Int. Conf. on Very Large Data Bases*, pages 315–326, 1999.

[17] Laurent Mignet, Mihai Preda, Serge Abiteboul, Sebastien Ailleret, Bernd Amann, and Amelie Marian. Acquiring xml pages for a webhouse. In *BDA'00*, 2000.

[18] G. Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38(11), 1995.

[19] Benjamin Nguyen, Serge Abiteboul, Gregory Cobena, and Mihai Preda. Monitoring xml data on the web. In *VLDB'01*, 2001.

[20] 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.

[21] Y. Papakonstantinou, H. Garcia-molina, and J. Widom. Object Exchange Across Heteregeneous Information Sources. In *ICDE Conf. on Management of Data*, 1995.

[22] 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.

[23] Jean-Pierre Sirot. Documents xml et serveurs d'information: approche, techniques et outil d'indexation bases sur l'utilisation d'une ontologie. Technical report, Rapport de stage de DEA I3, University of Paris-Sud, 2000.

[24] XQuery 1.0: An XML Query Language.
`http://www.w3.org/TR/xquery/`.

[25] Xyleme.
`http://www.xyleme.com`.

[26] Lucie Xyleme. A dynamic warehouse for xml data of the web. *IEEE Data Engineering Bulletin*, 2001.