The VLDB Journal manuscript No.
(will be inserted by the editor)

Views in a large scale XML repository

Vincent Aguilera[1], Sophie Cluet[2], Tova Milo[1,3], Pierangelo Veltri[1], Dan Vodislav[4]

[1] I.N.R.I.A., Rocquencourt France; (Vincent.Aguilera@inria.fr, Pierangelo.Veltri@inria.fr)
[2] Xyleme S.A., St Cloud France; (Sophie.Cluet@xyleme.com)
[3] Tel-Aviv University, Israel; (milo@post.tau.ac.il)
[4] Cedric/CNAM Paris France; (vodislav@cnam.fr)

Received: date / Revised version: date

Abstract We are interested in defining and querying views in a huge and highly heterogeneous XML repository (Web scale). In this context, view definitions are very large, involving lots of sources, and there is no apparent limitation to their size. This raises interesting problems that we address in the paper: (i) how to distribute views over several machines without having a negative impact on the query translation process; (ii) how to quickly select the relevant part of a view given a query; (iii) how to minimize the cost of communicating potentially large queries to the machines where they will be evaluated.

The solution that we propose is based on a simple view definition language that allows for automatic generation of views. The language maps paths in the view abstract DTD to paths in the concrete source DTDs. It enables a distributed implementation of the view system that is scalable both in terms of data and load. In particular, the query translation algorithm is shown to have a good (linear) complexity.

Keywords: XML, views, semantic integration, warehouse, query evaluation.

1 Introduction

We believe that XML will soon take an important and increasing share of the data published on the Web. This represents a major opportunity to, at last, provide an intelligent access to this amazing source of information.

With that goal in mind, we proposed to build a dynamic warehouse to store and provide sophisticate database like services over all the XML documents of the Web. This project, named Xyleme [38], was completed at the end of 2000. A prototype of the system has been implemented and actually turned into a product by a company named Xyleme S.A. [37]. Notably, users of Xyleme will be able to ask precise questions (such as "what are the names and addresses of Spanish museums that own a painting by Picasso?") and get accurate answers (i.e., XML documents with just the right information, not a list of often useless URLs).

Queries are precise because, as opposed to keywords searches, they are formulated using the structure of the documents. In some areas, people are defining standard documents types or DTDs (e.g., [8,27]), but most companies publishing in XML often have their own. Thus, a modest question may involve hundreds of different types. Since, we cannot expect users to know them all, Xyleme provides *a view mechanism*, which enables the user to query a single structure, that summarizes a domain of interest, instead of many heterogeneous schemas.

As in classical database systems, views in Xyleme are used to manipulate data coming from different collections through a unique and more convenient structure. However, the differences between the classical approach and Xyleme are as important as the similarities.

Relations vs. Clusters. Xyleme data is not stored into relations by proprietary applications but gathered by crawlers from the Web. To put some order in that massive amount of data, Xyleme relies on an automatic classification tool that partitions the documents into thematic clusters. E.g., the cluster named *art* contains all documents relative to art.

A view in a relational system builds one virtual (abstract) relation by combining information coming from various actual (concrete) relations (see Figure 1). Similarly, a view in Xyleme combines several concrete clusters into an abstract one. However, whereas the tuples in a relation share a common type, a cluster is a collection of highly heterogeneous documents. Thus, a view cannot be defined by one relatively simple join query between two or three collections but rather by the union of many queries over different sub-clusters. In other words, the size of a view definition in Xyleme is orders of magnitude larger than that of a relational view. Obviously, tech-

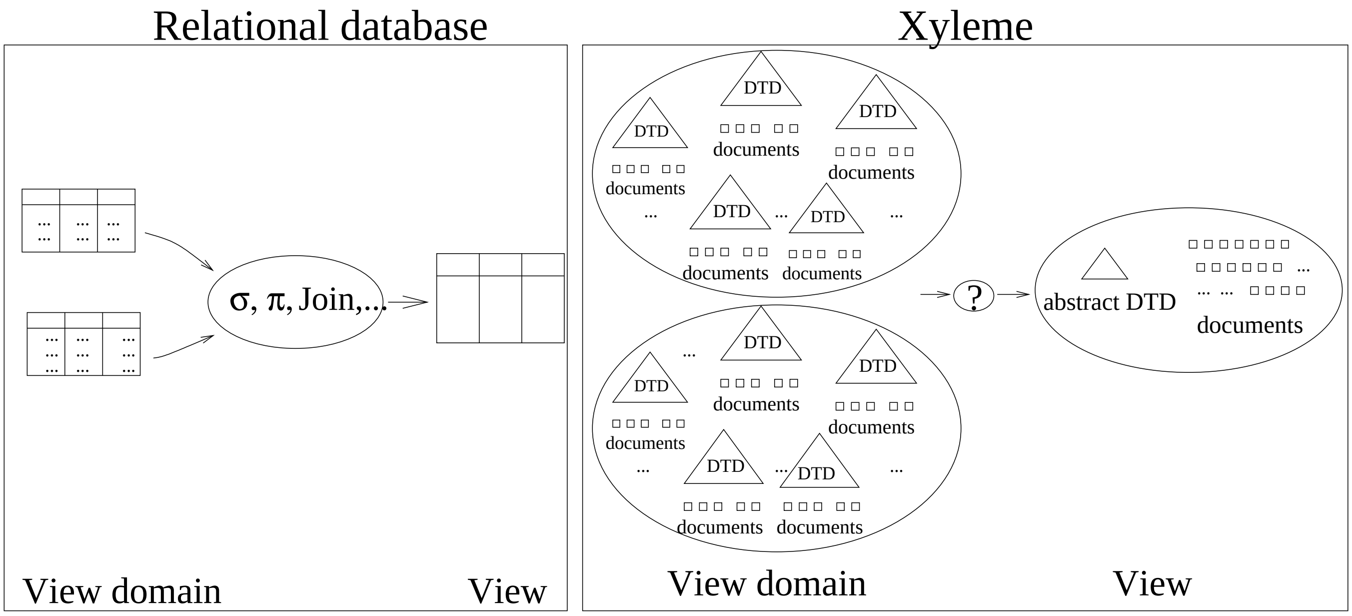


Fig. 1 Relational vs. Xyleme Views

niques that are used to maintain and query relational views have to be seriously re-considered to fit Xyleme's needs.

Human vs. Machine. Views are large because documents are heterogeneous. We are talking here of heterogeneity at a Web scale, one that human cannot cope with. This is why Xyleme supports a (semi-) automatic view generation tool [29]. A user creates a view by first (i) designing a view schema/DTD and (ii) selecting a view domain (i.e., some clusters). Then, the view generation tool extracts the various DTDs on this particular domain and, with the help of a thesaurus, finds possible mappings between the view schema and the documents structures. From these mappings, a view definition is generated. At any time during the process, the user can interact to refine the definition. Whereas a human cannot cope with large scales, on the other hand software does not have the intelligence of humans. This seriously constrains the expressive power of the view definition language.

This paper presents our solution to this new problem. We are not concerned here by view materialization but focus on four main issues: (i) the view definition model, (ii) the view implementation and distribution, (iii) the translation of queries against views, and (iv) the maintenance of view definitions. We chose a simple definition language to allow automatic generation of views. Still, this language is adapted to most practical cases and allows a distributed implementation of the view system that is scalable both in terms of data and load. The query translation algorithm has a good (linear) complexity. Future work will aim at a more powerful view language while preserving the good performance and scalability properties of the current solution.

1.1 Related work

To our knowledge, there is no previous work on views in a web-scale heterogeneous information system. Previous works provided view mechanisms with more powerful definition languages for relational [25,33], object [22], semistructured [1] databases or mediators [16,30], but they are not adapted to heterogeneity on a large scale.

There have been many research projects focusing on data integration for structured and semistructured data sources [19,21,18,30,7,5]. The common idea is to provide a global data description as an interface between end users and data sources. A query is formulated on the global description and then translated in a union of queries against the different sources.

Data integration system can be distinguished according to the way global description and sources are connected [32,20]. In a first approach called *global-as-view*, (GAV) the global description is a collection of views defined over the schema of the local sources [18,30]. Query translation algorithm can be very efficient, but the global description has to be changed whenever a local source is updated. In a second approach called *local-as-view* (LAV), local sources are defined as a view over the global description [21,5,6]. Source updating is quite straightforward, but query translation is an expensive computational process.

To obtain a scalable system and an efficient query translation, none of the two previous approaches is perfectly suitable. In our approach a global view is used to identify the clusters of documents (i.e. local sources) relevant for the query. Local views on each local source are defined by a system of mappings that connect the user query to data, and are used in query translation.

We are not concerned in defining XML views over heterogeneous data sources(e.g. as in [26], and [12]), neither in XML views on relational data (e.g. as in [31]). Our contribution can be considered as the definition of an XML system of views over a large scale XML repository.

1.2 Organization

The paper is organized as follows. In Section 2 we address the problem of: (i) defining and (ii) querying a view. In Section 3 we present the view implementation and in Section 4 the translation of a query against views into queries against data. The query evaluation process is presented in Section 5. Section 6 describes possible extensions to the view mechanism. Section 7 considers the view maintaining process and, finally, Section 8 concludes the paper.

2 View Definition

We first briefly introduce the Xyleme query language before presenting the view definition model. We then present the automated view definition process.

2.1 Queries in Xyleme

The Xyleme query language is an extension of OQL [11] with path expressions and the full text *contains* predicate. It is consistent with the requirements published by the W3C XML Query Working Group [35] and similar to many languages proposed by the database community for text or semistructured databases (e.g., [2,9,14]). It can also be considered as a subset of XQuery, the W3C proposal of query language for XML [36].

The most basic query filters a collection of XML documents according to some predicates, using the documents structure. For instance, consider the following query that retrieves the titles of van Gogh's paintings exhibited at the Orsay museum. The query is issued against documents of the "art" domain, contained in the *art* cluster.

Example 1
select *painting*/title
from *doc* **in** art,
painting **in** *doc*/WorkOfArt,
where *painting*/artist **contains** "van Gogh"
and *painting*/gallery **contains** "Orsay"

Obviously, the *art* cluster will contain documents of different types. The query considers only those with the specific structural pattern shown in Figure 2 (tags are in bold font, **title** is framed because it is part of the query answer). Note that, as opposed to standard database queries, structure is used here as an added means for filtering the documents.

Xyleme query language supports both child (/) and descendant (//) relations. In this paper, we ignore this distinction. Indeed, as we will see in Section 2.3, our view definition language considers only child relation between nodes.

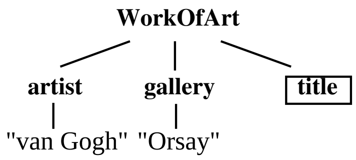


Fig. 2 Query Pattern Tree

This simple kind of query that filters a collection according to some tree pattern is called a tree query. It constitutes the basic brick of Xyleme query language. A particular algebraic operator called *PatternScan* has been designed especially to capture tree queries. It is implemented using an adaptation of the full text indexing technology [3] [4], and will be briefly presented in Section 5. Naturally, tree queries are also the basis of the view mechanism.

2.2 Views in Xyleme

Let us consider the query in Example 1. To formulate this query users have to know the structure of the queried documents. Moreover, documents in the "art" cluster with different structures than that of Figure 2 are filtered out by the query processor, even if they contain interesting information. For these two reasons, we provide a view mechanism.

In classical relational systems a view is created by selecting a domain (i.e. a set of relations containing data), designing a virtual schema (i.e. the view schema) and defining the correspondence between view schema and data in the relations by means of a query (see Figure 1). Similarly in Xyleme a view has a domain, a schema and a definition. In what follows, everything related to the view domain is called **concrete** (i.e. actual data) and everything related to the view itself is called **abstract**.

1. *The domain* of a view is a set of *clusters*, i.e., a subset of Xyleme repository. A cluster is a set of semantically related XML documents. E.g., the cluster named *art* contains all documents relative to art. The structure of the documents within a cluster is described by DTDs (also called *concrete DTDs*). Figure 3 shows three concrete DTDs (called *dtd1, dtd2 , dtd3*) belonging to the *art* cluster. Notice that they are represented as trees. They may actually be graphs. However, it is always possible to replace a graph DTD structure by a forest of tree-like DTDs, that we call

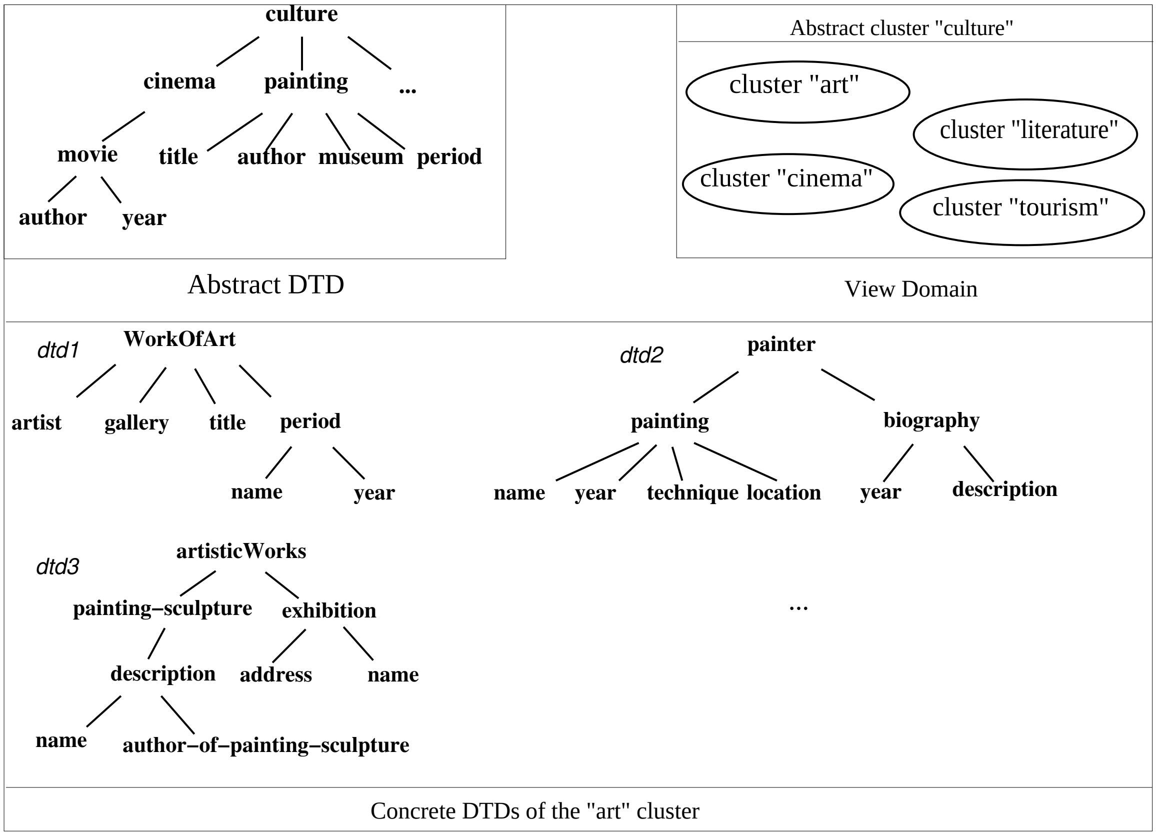


Fig. 3 An Example of View over the *culture* Domain

DTD summaries. We do this in Xyleme by considering all the roots of the documents conforming to a DTD and by extracting from the graph the trees rooted into these elements. In the sequel, we assume without loss of generality that all concrete DTDs are trees with a unique root. Moreover, we do not distinguish attributes from elements.

2. *The view schema* is an *abstract DTD*. It is a tree of concepts (rather than attributes or elements). Abstract DTDs describe abstract documents, i.e., those that are within the view. For instance, consider the abstract DTD shown in Figure 3. All edges must be interpreted as 0-n (i.e., "*") and provide context: e.g., *author* under *painting* may be interpreted as painter, while *author* under *movie* as director.
3. *The view definition* is traditionally a query. In Xyleme, we chose a simple view definition language. A view is defined by a set of pairs $< p, p' >$, called mappings, where p is a path in the abstract DTD and p' a path in some concrete DTD. Naturally, we call these paths respectively abstract and concrete. An abstract path p is associated with a non empty set of concrete paths p' in some DTDs.

The abstract DTD of Figure 3 represents the concrete DTDs of the clusters "art", "literature", "cinema" and "tourism". Thus Query 1 can be formulated on the abstract DTD as illustrated by the tree query of Figure 4. Using the view definition the system will translate this query into a union of queries on concrete DTDs (concrete queries), among which we will find the query of Figure 2.

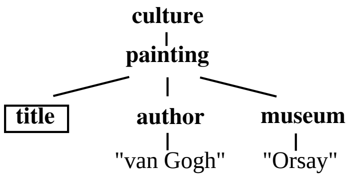


Fig. 4 The Query 1 Defined on the Abstract DTD

We now define the semantic of the view definition, presenting our choice and comparing it with other alternatives.

2.3 View Definition Language

The intuition that is underlying our views is that of "path-to-path" mapping, i.e. a view specifies mappings between path in the abstract and concrete DTDs. Before presenting path-to-path mapping, we discuss two alternative ways of mapping abstract to concrete DTDs: tag-to-tag and DTD-to-DTD.

We justify our choice by comparing these solutions according to four features: (i) size of the view definition,

(ii) automatic generation, (iii) precision in query translation, and (iv) query translation processing time.

Tag-to-tag mappings. The view is defined here by a set of mappings from *abstract tags* to *concrete tags*. An *abstract tag* is the label of a node in the abstract DTD tree (e.g., "painting" in the abstract DTD of Figure 3) and a *concrete tag* is the label of a node in some concrete DTD (e.g., "WorkOfArt" or "painting" in the concrete DTDs of Figure 3). Mappings can thus be generated using, for instance, synonym properties between labels and represented using tables. An example of tag-to-tag mappings relative to the DTDs of Figure 3 are represented in Table 1. A concrete tag is identified also by its DTD.

abstract tag	**concrete tag**
painting	WorkOfArt, dtd1
painting	painting, dtd2
painting	painting-sculpture, dtd3
title	title, dtd1
title	name, dtd1
title	name, dtd2
...	

Table 1 Tag to tag Mappings

We notice that an abstract tag, and likewise a concrete tag, may each be involved in several mappings.

One possible way to translate a query against a view, is to replace abstract tags by their corresponding concrete tags, while respecting the tree structure of the abstract query. All possible combinations are considered, each resulting in a tree query. Then in each tree query we verify that all concrete tags belong to the same DTD. The various tree queries are unioned to form the final result.

From a storage point of view, this solution is less costly, because tags occurring several times can be factorized. This is for instance the case of the abstract tag *title* in Table 1. It is also the easiest to generate automatically, just a matter of finding synonym properties between words.

This method is however the less precise since it completely ignores the structure of the documents. This is bad because it leads either to wrong or missing answers to user queries. Indeed, consider a query involving the abstract path: **culture/painting/title**. Consider the concrete DTD of Figure 3 rooted in **artisticWorks**. It contains the path: **artisticWorks/painting-sculpture /description/name**. With tag to tag mappings, one cannot translate one path to the other because they do not have the same length. The only possibility would be to systematically use descendant rather then child relationship. In this way **culture/painting/title** would be translated to **artisticWorks//name**. But then the name below **artisticWorks/exhibition** in some concrete document will be interpreted as the title of a painting.

The lack of precision is also bad from a processing point of view. Indeed, translating queries with tag-to-tag mappings leads us to consider all tags combinations, (i.e. k^n if k is the average number of mappings per abstract tag and n is the size of the query) many of which are useless because they do not appear in any concrete DTDs. Thus, this is very inefficient.

DTD-to-DTD (tree-to-tree) mappings. Another possible way to define views is to map abstract DTD to concrete DTD, that is a mapping between tree structures. An example is illustrated in Figure 5 where the selected abstract subtree on the left is mapped into a subtree of the concrete DTD on the right. The context of interpretation of each node is defined by the position of the node in the DTD. The dotted lines in Figure 5 represent the correspondence between abstract nodes (on the left) and concrete ones.

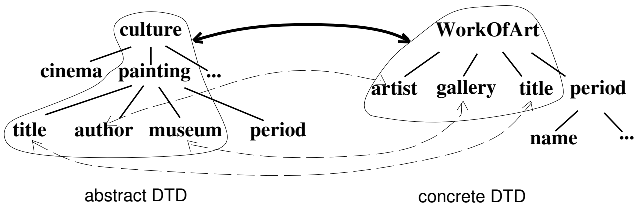


Fig. 5 A Mapping from DTD to DTD

Actually, this method corresponds to the classical definition of a database view, where the connection between two structures is defined by a query. For instance, the mapping clustered by Figure 5 corresponds to the following query:

Example 2

select culture [painting [title : t,
author : a,
museum : m]]
from p **in** WorkOfArt,
a **in** p/artist,
t **in** p/title,
m **in** p/gallery

The **select** clause builds the abstract DTD subtree that is bounded in Figure 5 and indicates the connection between abstract and concrete nodes (e.g., title and t, which stands for WorkOfArt/title).

Query translation with this definition can be performed in two steps. First we select the mappings whose abstract subtree includes that of the query. Then, from each of them, we generate a concrete query in a straightforward way. The result is the union of all these concrete queries.

The drawbacks of DTD-to-DTD mappings concern storage and view generation. DTDs often share similarities, because familiar concepts (e.g. address) often take the same form, or because new DTDs are often created by modifying existing ones (customization). DTD-to-DTD mappings cannot factorize such similarities, while the other mappings do.

Concerning view definition, it is doubtful that software will ever achieve the level of precision that is required here.

Still tree-to-tree mappings offer several advantages. Theoretically it is the most precise method. Consequently, query translation produces no useless and no irrelevant query.

Also, the query translation algorithm is efficient, provided that the selection of the right mappings is fast. This can be achieved with appropriate data structures.

Path-to-path mappings. This is the method we chose, because, as explained below, and illustrated by Figure 7, it combines the advantages of the two previous approaches. The idea here is to preserve the context of interpretation of abstract and concrete nodes, using their path from the root. This is illustrated by Figure 6 that shows a set of path-to-path mappings. Note that mapping is context sensitive, and so the fact that a/b/c is mapped to a'/b'/c' does not mean that a/b is mapped to a'/b'. Indeed, the example contains counter-examples of that. For instance consider **culture/painting/museum−> artisticWorks/exhibition/address**. This mapping simply states that the abstract concept describing the location of paintings is closely related to the one describing where exhibitions take place. It does imply that paintings and exhibitions are the same thing.

Query translation is performed in two steps. First, we select the abstract paths that compose the query. For instance, in the query of Figure 4 these are the paths
culture/painting,
culture/painting/title,
culture/painting/author, and
culture/painting/museum.
Then, among all the possible combinations of their associated concrete paths, we select only those that (i) form a subtree of some concrete DTD and (ii) preserve the parent/child relationship of the abstract query. The result is the union of concrete queries built from the valid tree combinations. For instance, given the mappings of Figure 6, the abstract query of Figure 4 will be translated by the query tree of Figure 2.

Path-to-path mappings save space by factorizing DTD similarities and allow semi-automatic mapping generation. In the next section we will sketch the approach used in Xyleme to automatically find mappings on the basis of semantic and structural criteria.

The main problem of the path-to-path mappings is precision. The descendant relationship constraint in query translation might lead one to miss relevant concrete queries.

culture/painting --> WorkOfArt
culture/painting --> painter/painting

culture/painting/title --> WorkOfArt/title
culture/painting/title --> painter/painting/name

culture/painting/author --> WorkOfArt/artist
culture/painting/author --> painter

culture/painting/museum --> WorkOfArt/gallery
culture/painting/museum --> painter/painting/location
culture/painting/museum --> artisticWorks/exhibition/address
...

Fig. 6 Path-to-path Mappings

Still remember that, automatic generation of mappings cannot be completely precise. As a matter of fact, the precision we miss here is typically that which cannot be inferred by software since links in XML do not have a semantics.

Query translation is as fast as for DTD-to-DTD mappings (see Section 4), because mappings are grouped by concrete DTD and the translation algorithm can process them together.

Method \ Feature	Storage	Precision	Query processing	Automatic view generation
tag–to–tag	●	○	○	●
DTD–to–DTD	○	●	●	○
path–to–path	◐	◐	●	◐

legend: ● very good ◐ good ○ bad

Fig. 7 Comparison between Mapping Methods

Figure 7 illustrates the features of the analyzed view definition methods. The method we implemented is the path-to-path. The measure unit is intuitive. Full circle means very good property, where empty circle means a weak property, and a partially full circle means good quality (i.e. an acceptable compromise). Note that the query processing parameter is strongly influenced by the mapping precision: a high number of irrelevant concrete queries negatively influences the query processing.

As a concluding remark, notice that views as presented here map structure to structure un-conditionally. In fact, the implementation of the view mechanism features simple conditional statements of the form "*path* **contains** *constant*". This extension is rather trivial and will not be discussed here.

2.4 Mapping Generation

In Xyleme, the difficulty of connecting abstract DTD and concrete DTDs comes from the very large number of data sources. Therefore, it is necessary to generate mappings automatically. For completeness, we present the basic principles of automatic mapping generation in Xyleme. A presentation of the problem and details about a first version of the implemented tool can be found in [29].

The Xyleme mapping generation tool uses a combination of two criteria: syntactic/semantic and structural. Similar methods are proposed by some recent mapping generation systems, based on various data models. SEMINT [23] does not use structural matching, but only syntactic/semantic criteria and information about data types, cardinality, value ranges, etc. DIKE [28], ARTEMIS [10] and Cupid [24,15] use both syntactic/semantic and structural matching. DIKE is based on ER schema modeling, where matching is influenced by the closest neighbors in the ER graph, ARTEMIS uses a class model, while Cupid uses XML, like Xyleme, with the difference that Cupid uses a bottom-up structural matching different from our top-down approach.

Syntactic and semantic matching. Let us consider an abstract and a concrete path. A mapping between them is generated only if the leaves of these paths are semantically similar. Two kinds of relations between tags are checked: syntactic and semantic matching.

Syntactic matching concerns the syntactic inclusion of a tag, or of a part of a tag, into another one. For instance, the concrete tag **author-of-painting** is syntactically similar to the abstract tag **author**. This method also includes techniques for detecting common abbreviations (e.g. **nb** for **number**).

Semantic matching is based on the use of extra knowledge available through existing ontologies or thesauri, such as the WordNet thesaurus [17], [34]. WordNet groups nouns, verbs, adjectives and adverbs into sets of synonyms that are linked through semantic relations. Each set of synonyms represents a concept. E.g. terms such as **picture, painting, image** are grouped to represent the concept of painting. The main semantic relations used for mapping generation in Xyleme are synonymy, hypernymy (generalization) and meronymy (part of).

This method can be used to generate tag-to-tag mappings as those of Table 1.

Structural matching. Syntactic/semantic matching between tags is not enough to obtain precise mappings. A tag may occur several times in the same DTD tree with different meanings. We consider that the interpretation context for the tag associated to a node is given by the tags on the path from the tree root to the node. For instance, the meaning of the tag **location** in the concrete DTD *dtd2* (see Figure 3) is given by its ancestors in the DTD tree, **painter** and **painting**, i.e. the tag represents the location of some painting.

The goal of structural matching is to map tags that respect syntactic/semantic matching *and that have the same interpretation context.* However, considering that the interpretation context is defined by *all* the tag ancestors is a too strong requirement. Only rarely are the paths of two equivalent tags in different DTDs completely similar (i.e. same length and the corresponding tags are similar).

In a first approach of defining the interpretation context (described in [29]), we decided to classify tags of a concrete DTD either as *object* names (e.g. **painting**), or *property* names (e.g. **name**, **technique**). Then objects characterized by a set of properties define the interpretation context of their properties. Some heuristics enable us to automatically decide which tags are objects and which are properties. Nevertheless, experimental results have shown the limitations of this method: no interpretation context for object names and not enough precision for the automatic heuristic-based method.

In the current approach [13], we forgot about this distinction. The abstract DTD designer (e.g. the user defining the view) has to manually define the interpretation context of all the *abstract tags* associated to nodes in the abstract DTD. He associates to each node in the abstract DTD tree, the list of ancestors defining its interpretation context. We called this list *the vector of context of the abstract tag.* The advantage of this method is that the abstract DTD designer has a very good knowledge of the abstract DTD domain and can define very precisely the interpretation contexts. Since the size of the abstract DTD is small, this manual operation is not costly.

The automatic generation algorithm will then produce the mappings between abstract paths and concrete paths of concrete DTDs in the view domain. Starting from the root node of the abstract DTD tree, the algorithm checks if mappings verify the following structural matching rule. Let us suppose that we must decide if some mapping $a \to c$ is valid, where a is an abstract path and c a concrete path, and the tags identified by these paths already match syntactically/semantically. If V is the vector of context of a, it contains only prefixes of a (ancestors of the node identified by a). Then the mapping $a \to c$ is valid only if for all $a' \in V$ there exists a mapping $a' \to c'$, where c' is a prefix of c. In this way, all the elements that define the interpretation context of the abstract tag have a corresponding partner in the interpretation context of the concrete tag. For instance, in the abstract DTD of Figure 3 the vectors of context of **culture** and of **culture/painting** are empty and the vector of context of **culture/painting/title** contains only **culture/painting**. The mapping **culture/painting** → **painter/painting** (connecting the abstract DTD with the concrete DTD *dtd2*) is valid because the abstract and the concrete tags are identical and there is no structural matching to do (the vector of context of the abstract path is empty). Then the mapping **culture/painting/title** → **painter/painting/name** is valid because "title" and "name" match semantically and because the only element of the vector of context of **culture /paint-**

ing/title (i.e. **culture/painting**) is mapped to a prefix of the concrete path (it is mapped to **painter/painting**).

3 View Implementation

In order to explain how views are implemented, we need to describe the distribution of data and processes in Xyleme. Thus, in the next section we briefly describe the Xyleme machine architecture. Then, in Section 3.2 we present the view implementation from two different angles: data structures and processing.

3.1 Distribution in Xyleme

The Xyleme system runs on a cluster of Linux PCs. As illustrated in Figure 8, we distinguish (functionally) three kinds of machines (from bottom to top):

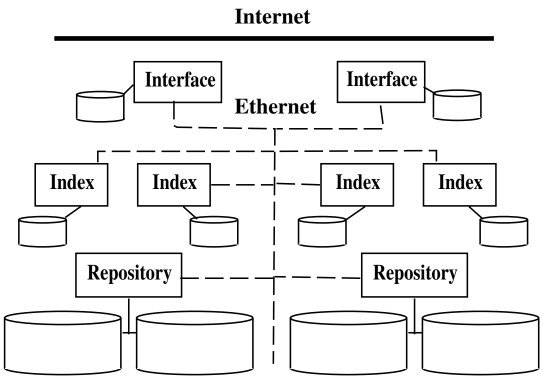


Fig. 8 Xyleme Machine Architecture

Repository machines (RM) are in charge of storing the documents. Data is clustered according to a semantic classification, each RM storing potentially several clusters of semantically related data (e.g., *art*, and *literature*). In this way, we reduce the number of machines that have to be accessed to evaluate a particular query. In order to also reduce IO operations, we index each cluster (see next item).

Index machines (XM) have large memories that are mainly devoted to indexes. We partition clusters on index machines so as to guarantee that (i) all indexes reside in main memory and (ii) each XM is associated to only one RM. We adapted the full text index technology to answer structured as well as keywords queries with one index.

Interface machines (IM) are connected to the Internet. They are in charge of running applications, compiling queries and of dispatching tasks/processes to the other machines. Typically, they all use the same meta information about e.g., data distribution. Whereas the number of RMs and XMs depends on the warehouse size, the number of interface machines grows with the number of users.

3.2 Distributing Views

Traditionally, a query against a view is evaluated by first replacing the view name by its definition. Then, the expanded query is optimized and executed. If we keep this model in Xyleme, the view must reside and be replicated in the memory of all the interface machines.

Still, remember that views in Xyleme are large and grow with the Web. More precisely, their size depends on the number of DTDs of the Web, which is potentially unlimited. Now, consider a machine with 1GB of memory. It takes 500,000 DTDs of 2KB[1] to consume it entirely. To obtain a scalable solution, we, obviously, must distribute views.

A view can be seen as a collection of subviews, one per DTD. A first obvious answer to the distribution problem is to split the collection among the different interface machines. This solution is not appropriate for mainly two reasons. First, to evaluate a query, one will have to broadcast and pre-process it on all interface machines. Second, the number of interface machines grows with the number of users, not with the size of the warehouse and, consequently, of the view.

Another bad answer to the problem is to try and bypass distribution by, e.g., encoding views so as to reduce their size as much as possible. However, no matter how smart you are at encoding paths, this does not scale.

Thus the standard translation pattern must be reconsidered. We process abstract queries in two steps. First, the query is pre-compiled on an interface machine. Using local information, we find which clusters of data are concerned and generate a plan whose specificity is that it contains abstract instead of concrete tree patterns. E.g., considering the abstract query represented in Figure 4: the pattern tree is preserved while the abstract cluster (domain) *culture* is replaced by the clusters containing mappings for all the abstract paths in the query (e.g. *art* and *tourism*). In a second step, the plan is distributed and evaluated. For efficiency reason, evaluation always starts on index machines. There, the abstract patterns are translated into concrete ones before they are used to query the index. Section 4 explains this in greater detail.

Note that this two-step translation has some very nice properties. First, we avoid useless broadcast. Also, the plans that are shipped are small, they do not include the many combinations of concrete patterns matching an abstract one. But more importantly, we have solved the distribution problem. Indeed, the only "global" information that needs to be maintained on the interface machines is a correspondence between abstract DTDs and clusters. The remaining view information is naturally distributed over the concerned index machines.

To illustrate this, let us reconsider the view example on the *abstract* cluster, "culture" introduced in the

[1] This is the average size of the DTDs found on the Web.

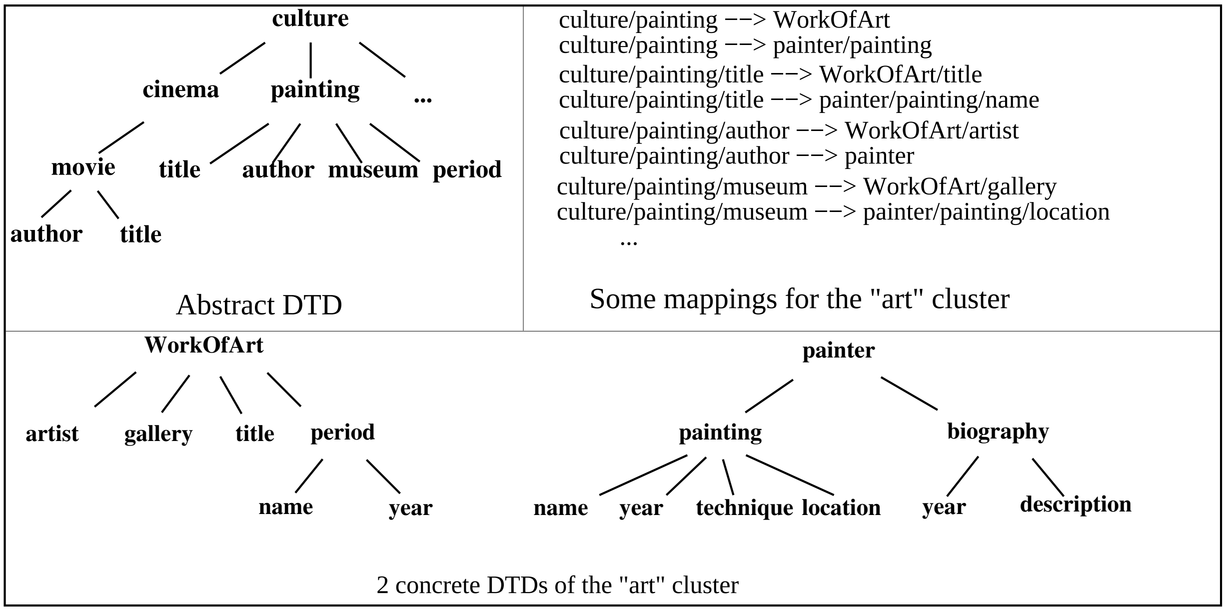


Fig. 9 A View Over the *culture* Domain

previous section and recalled in Figure 9 with the path-to-path mappings. We consider here the representation of a view in main memory. The persistent representation is a straightforward translation into XML documents. As explained above, we distribute the view among interface and index machines.

On each interface machine, we find a tree representing an annotated abstract DTD. More precisely, each node is marked with the clusters in which there exists matching concrete paths (see Figure 10 for an example). This structure is used at pre-processing time to understand how the query should be distributed. It is replicated because each interface machine must be able to translate all queries. Note that it could have been made smaller by keeping only the root of the abstract DTD. However, as it is, it allows to (i) check the abstract "typing" of queries and (ii) reduce the number of subplans (e.g., if the user is interested in titles of paintings, there is no need to generate a plan over the *cinema* cluster). Also we will see in Section 6 that this tree is used to support joins in view definitions.

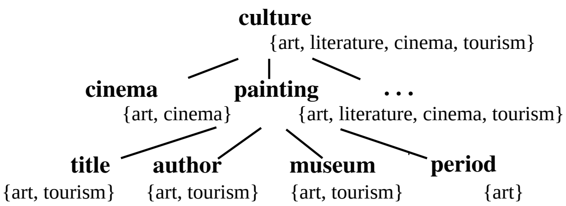


Fig. 10 View on an Interface Machine

Note that interface machines manage only abstract DTDs and their associated clusters, two items whose size is usually rather small and very much controlled.

0	WorkOfArt	-1
1	artist	0
2	gallery	0
3	title	0
4	painter	-1
5	painting	4
6	name	5
7	year	5
8	location	5

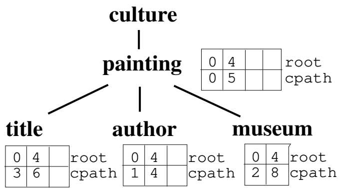


Fig. 11 View on an Index Machine

On each index machine, we find the view information relative to its indexed clusters. Figure 11 corresponds to the representation of the mappings given in Figure 9. It consists of two parts, a table and a tree. (i) The table represents in a simple way the forest of all concrete paths that have been mapped to some abstract paths. Each node is represented by its tag and the identifier of its father (-1 when it is a root). Nodes are identified by their entry in the table. E.g., 6 identifies **painter/painting/name**. (i) The tree maps abstract paths to concrete paths. Concrete paths are represented in the tree by two integers identifying, respectively, the concrete path itself (**cpath**) and the DTD root element from which it stems (**root**). The second information is here to reduce the complexity of the translation process (consider that there are thousands of concrete paths but rarely more than a dozen stemming from one root element; see Section 4).

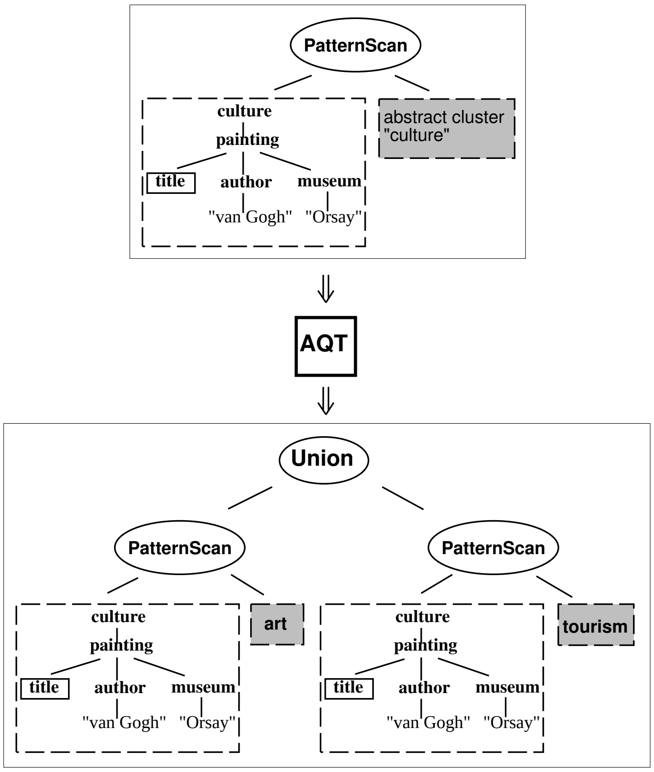


Fig. 12 AQT Transformation of an Abstract PatternScan

Note that the size allocated to a view on an index machine is very small compared to the size of the index itself (usually less than a thousandth). Also, the size of a view depends on the size and heterogeneity of clusters. When a cluster becomes too big, we refine the classification so as to split it. This results in a re-organization of store and indexes that is performed lazily while (re-)loading. Views are reconstructed when the index re-organization is over. In the meantime, views are simply larger than they should.

We are now ready to see how abstract queries are translated into concrete ones.

4 Query Translation

As explained in the previous section, the evaluation of a query against a view is performed in three steps:

1. On some interface machine, the tree-query is parsed, and an algebraic expression is generated. This expression features a *PatternScan* operator that retrieves, from a collection of documents, the elements that match the given pattern. At this stage the PatternScan is defined on an abstract pattern and on an abstract cluster. The *Abstract Query Translator* (AQT) translates the PatternScan into a union of PatternScan operators defined on concrete clusters, but still featuring abstract patterns. Then, the query is partly optimized, an execution plan is generated and distributed. The query execution plan is sent only to local machines indexing the corresponding clusters.
2. Plan evaluation starts on index machines. There, a physical operator called *Abstract to Concrete* (A2C) translates abstract patterns into unions of concrete patterns.
3. Each such pattern is evaluated using a special *PatternScan* algebraic operator, that utilizes the full-text index and returns the selected elements and documents (see Section 5). Eventually, those may be further processed by other operators on other machines.

We now describe the AQT and A2C operators.

4.1 The Abstract Query Translator (AQT)

The AQT introduces concrete clusters in an abstract PatternScan operator. It returns a union of PatternScan operators, each defined on a concrete cluster. This process is illustrated by Figure 12 where the abstract cluster called *culture* has been replaced by the two concrete ones *art* and *tourism*. Note that the pattern tree has not been modified, what changes in the PatternScan is the cluster argument (darkened part in the Figure 12). The algebraic operators, PatternScan and Union are framed in a circle, while the arguments are enclosed in dashed line boxes.

To achieve this, the AQT asks the view manager for the means to navigate within the annotated tree of the view of Figure 10. Then, it visits both the annotated view tree and the query tree, checking that the latter is consistent, and computing the intersection of the concrete clusters found on the way. Finally, it generates a union operation with as many branches as clusters (see Figure 12).

The AQT result is given to the query processor that incorporates it into the global algebraic expression and generates an execution plan. The important role of the AQT is to avoid useless broadcast: query execution plans are sent only to local index machines concerning the queried clusters.

In Section 6.1, we will see that the AQT gains in complexity when we add joins to the view definition language.

4.2 Abstract to Concrete Translation (A2C)

The A2C module transforms abstract pattern trees into concrete ones. Its input is an abstract pattern and a view identifier. Its output is a set of concrete patterns. Figure 13 illustrates the result of the A2C translation of an abstract pattern matched against the view of Figure 9.

Thus after a local query translation, PatternScan operators are now defined on concrete queries. Figure 14

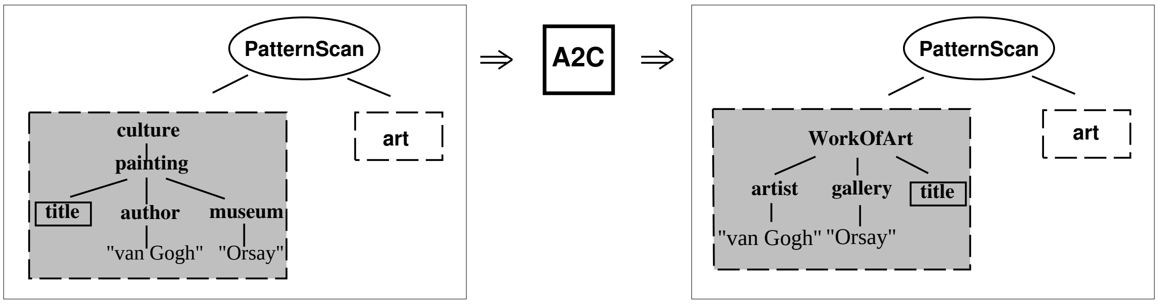


Fig. 14 The PatternScan Operator after an A2C Translation

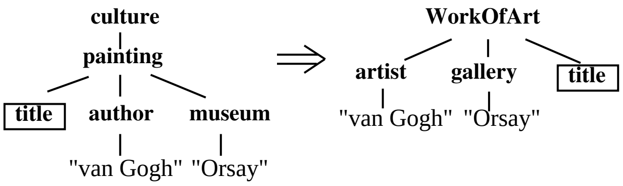


Fig. 13 A Concrete Pattern Generated by A2C

shows the structure of one of the PatternScan of Figure 12 (i.e. the one defined on the "art" cluster) after the A2C local translation. The grey selected part is the argument changed by the A2C.

The main problem of the A2C algorithm is due to the large amount of mappings associated to each path of the abstract DTD. For n nodes in the abstract query pattern, with k mappings for each node, A2C should examine k^n possible configurations! Actually, few of these configurations are valid because, as briefly explained in Section 2.2, the corresponding concrete paths must (i) belong to the same concrete DTD and (ii) preserve the descendant relationships of the query. We explain the second requirement in more detail.

Rule *PreserveAncDesc.* Let a_1, a_2 be nodes of an abstract pattern tree T_a, with a_2 descendant of a_1, and c_1, c_2 their corresponding nodes in a concrete pattern tree T_c. Then T_c is a valid translation of T_a only if c_2 is a descendant of c_1.

This rule states that one cannot swap two nodes when going from abstract to concrete. Somehow, it implies that descendant is a semantically meaningful relationship that cannot be broken. This is sometimes true (e.g., **painting/name**) and sometimes not (e.g., **painter/painting**).

This constraint can reduce the number of concrete queries captured by the query translation. For instance, given the mappings of the view of Figure 9, the abstract query of Figure 4 will not have a translation in the DTD rooted in "painting" of Figure 9 since it reverses the painting/painter relationship.

Still, we chose to impose this rule because it reduces the complexity of the A2C algorithm. We estimate that the number of lost DTDs is very low in practice, where an efficient translation is relevant for users that are generally impatient to obtain results. However, in case of few answers we can relax this constraint to find more solutions. In Section 6, we discuss the relaxation of this rule.

Rule *NoTwoSubpaths.* In order to further reduce the complexity, we also impose a technical rule that may seem somewhat arbitrary but is rarely violated in practice. Let V be a view defined by the set of path-to-path mappings M. Let $(a \rightarrow c)$ be in M and a_p be a prefix of a. Then, V is valid only if there does not exist c_1, c_2 distinct prefixes of c such that:

$$(a_p \rightarrow c_1) \in M \wedge (a_p \rightarrow c_2) \in M$$

This means that a should not have an ancestor that is mapped to two different ancestors of c. In other words, there should be at most one solution to the mapping of nodes along an abstract path to nodes along some concrete path.

Exceptions may exist, for instance the abstract path **culture/painting/author** could be mapped to both **painter** and **painter/influenced_by/artist**. Nevertheless, such cases are rare and the rule is respected in practice

The A2C algorithm

Consider the leftmost branch of the abstract pattern tree of Figure 13. Rule *PreserveAncDesc* implies that the translation of this branch is another branch that can be computed going up and Rule *NoTwoSubpaths* guarantees that, once the leaf mapping has been chosen, there is at most one solution. This solution is constructed as follows: we choose a concrete node for **culture/painting/title** (e.g., **WorkOfArt/title**) then we go up and search the mappings of **culture/painting** among the prefixes of **WorkOfArt/title** (e.g., **WorkOfArt**).

To compute the translation of a whole tree, we decompose it in upward paths starting from each leaf and stopping when we reach a node that has already been visited by a previous upward path. This is illustrated on the right part of Figure 15 (note that we ignore constants in the query tree). The left part of the figure is a reminder of the local view structure presented in Sec-

0	WorkOfArt	-1
1	artist	0
2	gallery	0
3	title	0
4	painter	-1
5	painting	4
6	name	5
7	year	5
8	location	5

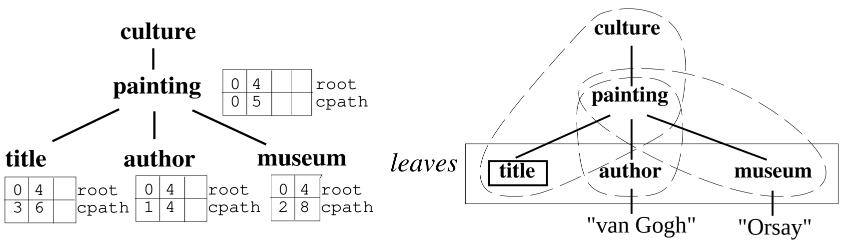


Fig. 15 Upward Paths from the Pattern Leaves

tion 3.2 that we will use to illustrate the translation process. In the example, the two right upward paths stop at node **painting** instead of **culture**. We call this node their *upperbound*. It constrains the upward path interpretation with that already attributed to **painting** by the previous paths. Once the decomposition has been performed, A2C translates each upward path to a concrete branch, then it computes concrete pattern trees by combining branch solutions as follows.

Remember that, as shown in the left part of Figure 15, the view stores for each node of the abstract DTD its mappings as a list of couples (*root*, *cpath*), where *root* identifies the concrete DTD and *cpath* the concrete path of the mapping. This list is sorted by *root* and then by *cpath*.

First suppose that *each leaf has at most one mapping for each root*. Then the A2C algorithm computes the solution by finding compatible branch solutions going from left to right, as follows:

1. The leftmost leaf L is the master leaf. It considers its mappings one by one, the other nodes in the abstract pattern remaining "synchronized", i.e. the mapping that they consider at any time has the same root as L. The reason is that a concrete pattern solution must have the same root for all its nodes. E.g., suppose that we move from one mapping to the next in L and that, in so doing, we go from $root_{i-1}$ to $root_i$. Then all other nodes advance to their next $root_i$ mapping.
2. Concrete branches are computed upward starting from their leaf (it exists at most one such branch, as explained above). For each abstract node on the upward path, A2C looks for a mapping among those with $root_i$ and checks that is a prefix of the *cpath* already found for the node below it. Checking that a *cpath* is a prefix of another one is done in constant time using the concrete path table on the left in Figure 15 (i.e. typically 1 or 2 table accesses, which is the difference of length between the paths). The branches other than the leftmost one must contain the *cpath* that has been computed by some previous branch for their upperbound (if any).
For instance, if the leftmost upward path in Figure 15 found the mapping (0, 0) for **painting**, the upward paths of **author** and **museum** are constrained to find the same mapping when computing their concrete branches.
3. A solution is found when all upward paths have a concrete branch solution. Then L goes to its next mapping to search for a new solution, and so on, until all the mappings of L have been explored.

Now, suppose that there is more than one mapping for a given node and a root. Note that this rarely happens. Then for each distinct $root_i$ of L, we check all possible combinations of the pattern leaves "$root_i$" mappings. This implies some backward steps in leaf mappings (except for the master leaf L).

Complexity

The scalability factor for this algorithm is k, the average number of mappings for an abstract node (the size of the mapping lists in Figure 15). To be more precise, k can be decomposed into two factors: $k = r \times m$ where r represents the average number of distinct *root*s associated to one node and m the average number of mappings of a node for a given root. Now, the scalability factor becomes r since it grows proportionally to the number of stored DTDs.

All other values, i.e., m (usually 1, because an abstract path, is rarely mapped to more than one concrete path in the same DTD), the number of nodes/leaves in the abstract query tree (4 or 5 in average) or the length of the upward paths (2 or 3 in average) are considered as constants relative to r.

The cost of a branch construction is at worst $h \times m$, where h is the height of the abstract pattern, because for each node of the upward path there are m mappings for the current root.

For each mapping of the leftmost leaf (k mappings), A2C must build at worst the branches for all the combinations of mappings with the same root for the other $l-1$ leaves (l is the number of leaves of the abstract pattern). There are m^{l-1} combinations, so for each mapping of the leftmost leaf A2C computes $1 + m^{l-1}$ branches. The overall complexity in the worst case is then :
$k \times h \times m \times (1 + m^{l-1})$.

The algorithm is linear in k and this proves its scalability. The constant may be evaluated by remarking that m has an average value very close to 1, h is often smaller than 3 and l smaller than 5; the result is a fairly reasonable worst case complexity (6 k if $m = 1$, 10 k if $m = 1.2$). In the average case, A2C does not compute all the m^{l-1} combinations, because when we fail to build a branch from a leaf mapping, the leaves on the right of it will not build branches for that combination. This leads to a lower constant in the average case.

5 Query Evaluation: the PatternScan Operator

The PatternScan operator is the heart of the Xyleme query processor. A full description of the system's query processor is beyond the scope of the present paper and can be found in [3]. We only describe here the main principles in order to complete the picture of view queries evaluation.

The PatternScan operator takes as input a cluster name and a (concrete) tree pattern. Its goal is to select the cluster documents that contain the given pattern, and return the marked elements (e.g. *title* in Example 14). This is done using Xyleme's full-text index. This index is essentially a big hash table whose entries are the tag names and words in the indexed documents. To allow for fast processing of tree pattern, XML documents are viewed as trees, (essentially the parse tree of the document), and each document element (node in the tree) is given a unique identifier. Every entry (tag/attribute name or word) in the hash table is associated with the list of ids of the relevant document nodes. A special feature is that the node ids are designed such that given the ids of two nodes one can determine whether one node is an ancestor or a parent of the other. (We will see the exact structure of the ids in what follows). Thus, structural queries can be answered using the index only, without access to the actual document. We first use the index to retrieve the items (in fact, their ids) having the required tags/values (**artist**, **gallery**, **title**, "Orsay",...). Next, we iterate over the retrieved id sets and select those having the appropriate ancestor/parent relationships, and finally, we project out the ids of the marked elements.

Consequently, the PatternScan operator is implemented using the following three lower level operations.

FTI gets a string as input and uses the full-text index to retrieve the ids of items containing the string or having this tag/attribute name.

$\bowtie_{pred,i,j}$, for $pred \in \{ParentOf, AncestorOf\}$ gets an input two relations of node ids, and joins tuples where the i attribute of the first tuple contains an id of a parent/ancestor of the one in the j attribute of the second tuple. For brevity, in what follows we will omit i and j when clear from the context and simply use $\bowtie_{pred}$.

$\Pi_{i_1,\dots,i_k}$ is a standard projection operator that gets as input a relation of node ids and projects out the ids in columns $i_1, \dots, i_k$.

For example, the PatternScan of Figure 14 can be implemented using the algebraic expression shown in Figure 16 (presented for convenience as an expression tree).

We note that the *AncestorOf* join predicate is used to implement the "contains" predicate (e.g. see Query 1), while the *ParentOf* implements path expressions.

To complete the picture we need to describe the particular ids that Xyleme assigns to the nodes, and explain how they are used to implement the join operations and to determine the join ordering.

5.1 Node ids

The id of a node n consists of four parts: $tree(n)$, $pre(n)$, $post(n)$, and $level(n)$. $tree(n)$ is simply the identifier of the document to which n belongs. $pre(n)$ (and resp. $post(n)$) is the number of n in the preorder (postorder) traversal of $tree(n)$, and $level(n)$ is the distance of n from the root node. An example for a tree with the ids assigned to the nodes is given in Figure 17.

Given two nodes n and m, the following is easy to verify.

(i) n is an ancestor of m if and only if $tree(n) = tree(m)$, $pre(n) < pre(m)$, and $post(n) > post(m)$.

(ii) n is the parent of m if and only if n is an ancestor of m and $level(n) = level(m) - 1$.

The ids in each entry of the index hash table are sorted first by their tree identifier, and within each tree id by the preorder number. We chose this particular id scheme (and ordering) for three reasons. First, the preorder number associated to each node naturally corresponds to the order in XML documents. Second, the repository we use provides a fast access to document nodes given their preorder number. Finally third, and most important, as we shall see below, this id scheme has properties that allow us to evaluate the joins required by the *PatternScan* with minimal number of scans of the of the joined relations.

5.2 Implementing Joins

A large number of join algorithms have been proposed so far in the literature. However, many of them do not fit Xyleme needs since they use intermediate data structures. This is unacceptable here since we expect thousands of concurrent queries, so storing intermediate results or using any big data structure per query is impractical.

To understand the main idea behind the join algorithm that we developed, lets look first at a simple example. For a node id l we use below *l.tree* to denote

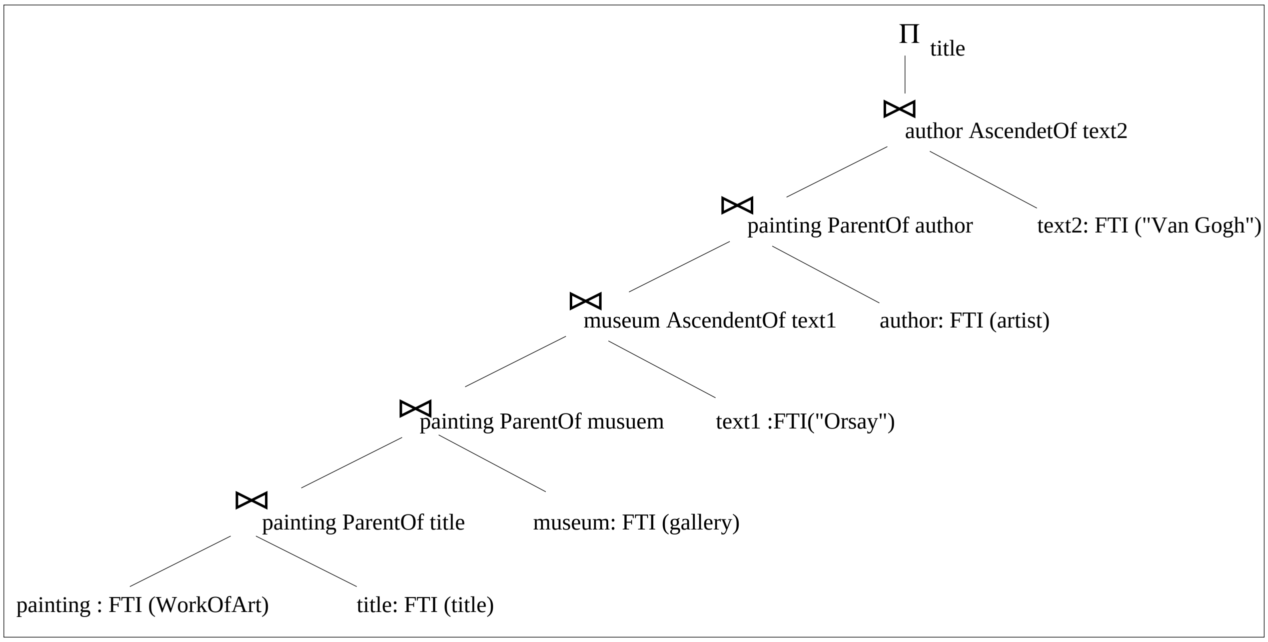


Fig. 16 Pattern Scan Implementation

the *tree* component of the id (and similarly *l.pre*, *l.post*, *l.level* for the other components).

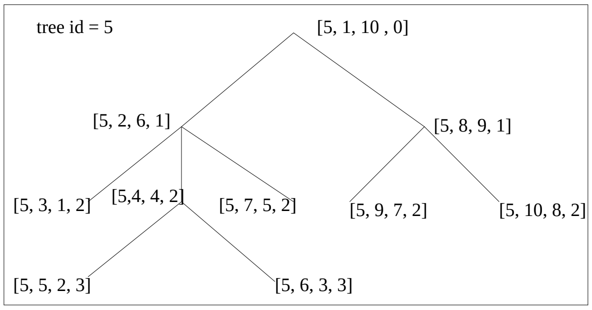


Fig. 17 An example of tree with id of the form :*tree id, preorder (n), postorder(n), level(n)*

Basic Joins

Let A and B be two sets of node ids, each sorted by tree identifiers, and within each identifier, by the preorder number. Assume we want to evaluate $A \bowtie_{AncestorOf} B$. The result is obtained using two nested merge-join like computations. In the outer loop, the two lists are merged based on tree identifiers. In the inner one, for each tree identifier t, the corresponding sublists A_t and B_t (each sorted by preorder number) are processed in a merge-join like manner to select the appropriate ancestor-descendant pairs. The process is based on the following two observations.

1. For each id $a \in A_t$, B_t can be divided into three disjoint consecutive sublists.
   preMatch(a) containing the elements $b \in B_t$ such that $b.pre \leq a.pre$.
   Match(a) containing the elements $b \in B_t$ such that $a.pre < b.pre$ and $a.post > b.post$, (in particular these are precisely the ids of descendents of a.)
   postMatch(a) containing the rest of the ids in B_t, (namely ids where $a.pre < b.pre$ and $a.post < b.post$,
2. For every two consecutive ids $a_1, a_2 \in A_t$, one of the following hold.
   (a) either $a2.post < a1.post$ (hence a_2 identifies a descendent of a_1), in which case $Match(a_2) \subseteq Match(a_1)$, or
   (b) $a2.post > a1.post$ (namely a_2 is not a descendent of a_1), and then $Match(a_2) \subseteq postMatch(a_1)$.

We use these properties to avoid re-scanning the full B_t list for each element $a \in A_t$. We start with the first a in A_t and iterate over the B_t list until a match is found. The beginning of $Match(a)$ is recorded, and the matches for a are produced while iterating over the $Match(a)$ sublist, until $postMatch(a)$ is reached. Then we move to the next element in A_t. If it is a descendent of the previous a, the above process repeats starting from the (recorded) beginning of $Match(a)$. (Recall that, according to observation 2a above, all the elements in $preMatch(a)$ need not be re-scanned). Otherwise, if the element is not a descendent of a than we can furthermore skip all the $Match(a)$ elements and continue from $postMatch(a)$. (See observation 2b above).

Finally note $A \bowtie_{ParentOf} B$ can be evaluated using essentially the same algorithm, outputting only the pairs where *level* of the A element is greater by one that that of the B.

Intermediate joins

The above algorithm works well when we join id lists that originate from the index, (since the index entries are ordered by tree id and preorder number). Note however that while the output of such a join is still ordered by tree id, the preorder order is preserved only for the A ids and not necessarily for the B's. Consequently, to use this output as an input for further joins on the B attribute, the join algorithm needs to be adjusted. We shall see below that such intermediate relations are in fact used only as the first argument for joins. Thus we only need to change the algorithm to deal with "wrong" sorting of the first relation: Whenever the A_t relation is iterated and an id in the joined attribute turns out to have a small preorder number than the previous id a, we need to start re-scanning B_t from the beginning (rather than just from $Match(a)$ or $postMatch(a)$ as before).

Ordering Joins

To conclude we explain how the order of joins is determined. Traditionally, this is achieved with the help of a good cost model and by applying dynamic programming. But in the context of Xyleme, a good cost model is something hard to come by. Indeed, Xyleme stores a large amount of highly heterogeneous and dynamic data. In this context, statistics are very hard to maintain. Given that, we have to rely on heuristics to order joins. Our join ordering is designed to allow for pipelined computation and avoid storing intermediate results. It is based on following two assumptions : (i) the joins should scan only once the intermediate computed relations, and (ii) the order of the output tuples must be adequate for computing the final projection, and in particular allow for simple duplicates elimination.

To understand (i), recall that in the join algorithm described above, the A relation is scanned only once, whereas some portions of the B relation may need to be re-scanned. If we allow B to be itself the result of some previous joins, to support such re-scan we either need to store it (which, as explained above is impractical given the large number of concurrent queries), or to recompute it (which is obviously very expensive). This motivates using intermediate results only as the first argument for further joins, namely constructing a left-deep join sequence.

For (ii) recall from Figure 16 that PatternScan is implemented by a sequence of joins followed by a projection. Projection requires the elimination of duplicate tuples from the result. To do this without having to sort the relation (which requires using a large intermediate data structure), the tuples must already be sorted according to the projected attributes. To construct a join ordering whose output has this property, we traverse the pattern tree twice in a left deep manner. During the first pass, we number consecutively a subset of the nodes that includes the projected nodes and their ancestors. The remaining nodes are numbered in the second pass. It is easy to see that joins that are ordered based on this numbering have the required sorting for the output. For example, the expression tree in Figure 16 obeys this order.

6 Improvements

So far, we have considered views that map a tree to a collection of trees (or a virtual document to a collection of concrete documents). In other words, only documents that contain all the items of information sought by a user will participate in the query result. E.g., the answer to the example query is empty unless there exists a document containing information about painter, painting and museums as well.

Most probably, given the size of the abstract DTDs and that of the Web documents, there will be queries without answers. There are two solutions to this problem. The first and most satisfying one is to find a way to follow the links from documents to documents. In database language, this means introducing joins in the view definition language. The second solution is to relax the query semantics until some answers can be found. Note that both solutions are not exclusive. Here, we present them.

6.1 Joins in Views

Obviously, for complexity reasons, we do not plan to extend the definition language with arbitrary joins. Our purpose is to introduce only those that will allow us to follow the links that can be found in XML documents. For instance, suppose that a document concerning a painting by "van Gogh" contains a link to another document containing information about the "Orsay" museum where this painting is exhibited. So the information about the museum is not in the same document, but can be found by following a link. We want to be able to add this piece of information to the query result.

The simplest way we found of achieving this goal is to simply record the fact that a concrete path contains a link. Consider the above example. Assume that the concrete path corresponding to the museum link is **picture/exhibition**, and that it belongs to a DTD of the "art" cluster. Then we add the following mapping to the view: **culture/painting/museum→ picture/exhibition, link**. Note that this extra "link" information is very easy to find.

Now, we have to add the appropriate joins in the translation process. Joins may introduce the need to communicate results from machine to machine (the link can point to documents belonging to another cluster).

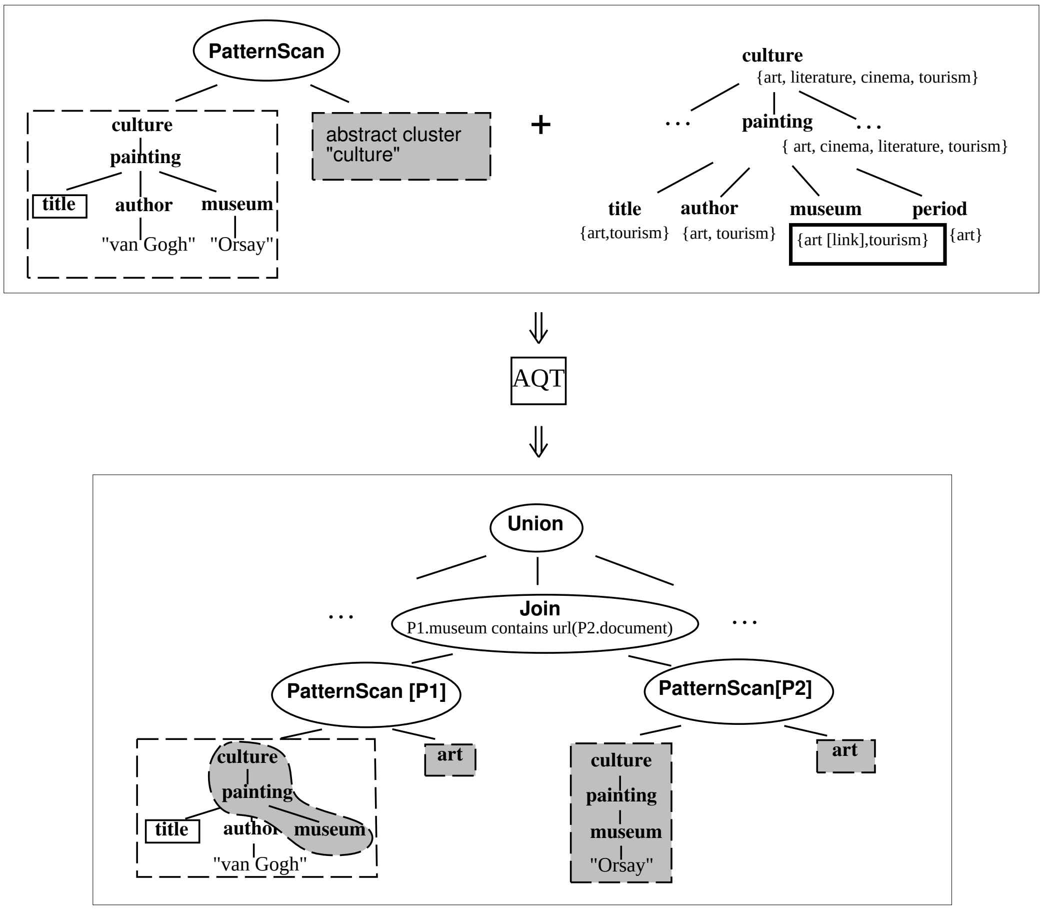


Fig. 18 Introducing Joins in the View Definition

Thus, to avoid re-distributing local execution plans, joins have to be introduced before the global plan is generated and installed. For this we re-define the way views are represented (and queries translated) on interface machines. This is illustrated by Figure 18.

The figure shows the global view information of Figure 10 revisited with links. Note that there is only one change: the node corresponding to **culture/painting/museum** is annotated with *art[link], tourism* (the bold framed annotation). This means that we can find elements corresponding to the concept of **museum** in the *art* and *tourism* clusters and that, in the former, there exists a concrete representation of that concept that is followed by a reference.

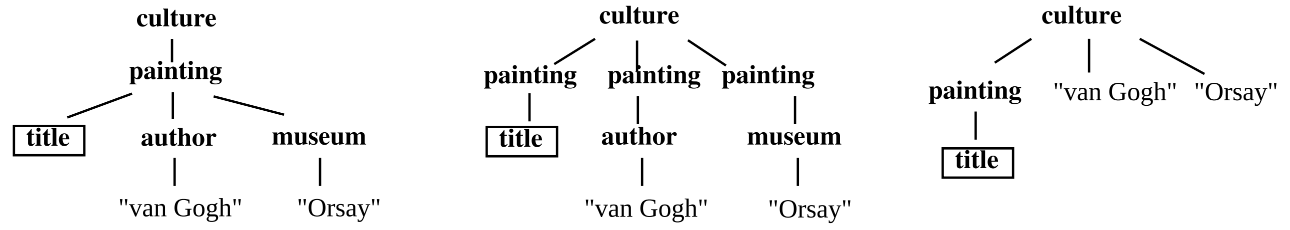
The bottom part of Figure 18 partially shows the AQT translation of the query *PatternScan* operator. It is an n-ary union whose arguments are those described in section 4 plus some more. More precisely, the figure shows one of the two joins that have been introduced because of the **museum** link annotation (the remaining joining paintings of the *art* cluster to museums of *tourism*). We notice that in this case AQT does more then just changing the cluster arguments of the PatternScan: it also modifies the structure of the abstract query tree, for the PatternScan operators on which the Join is defined. The gray framed part of Figure 18 represents the arguments of the PatternScan modified or added by the AQT.

Joins lead to a potential exponential growth of the query algebraic plan and, accordingly, to queries that are much too complex to be answered in a reasonable amount of time. In practice, plans remain relatively small because (i) abstract DTDs concern few clusters, (ii) queries are naturally small, and (iii) not all nodes have links. Still, worst cases can always occur.

One solution then consists in considering joins only as a backup when no or too few answers are found. Remember that they are part of a large union operation. Thus, it is easy to deactivate them in a first round, and incorporate them progressively (starting from the solutions with fewer joins) according to the user feedbacks. This part is fully implemented in the current version of the Xyleme system.

6.2 Query relaxation

Joins are one possible solution when a query returns too few results. We are considering here another solution called query relaxation that has been considered in different flavours in information retrieval systems.



Initial abstract query (levels 0 and 1) The abstract query at level 2 The abstract query at level 3 (keyword search)

Fig. 19 Query Relaxation

Database-like queries, such as those answered by Xyleme, may miss some interesting documents because they have a slightly different structure than the abstract query.

The query relaxation phase consists in relaxing some of the rules that guide the translation process (e.g. the *PreserveAncDesc* rule, presented in Section 4), and/or modifying the abstract query structure to find more results. We proposed in Xyleme three levels of relaxation illustrated by Figure 19:

- *Level 1.* We discard the rule *PreserveAncDesc* (Section 4) that constrains the A2C algorithm. Thus the path **painter/painting** is now an appropriate match for the abstract path **culture/painting/author**.
  Note that by removing this rule, we augment the complexity of A2C. More precisely, when constructing an upward path, we must now consider all combinations of mappings having the same concrete root. In other words, we add a complexity factor of m^h where m is the average number of mappings for an abstract node within one concrete DTD tree and h the height of the query tree. Still, remember that these numbers are very small.
- *Level 2.* We relax the conditions on joining nodes. For instance in the central query of Figure 19, the node painting is now repeated for each path. The A2C evaluates all the possible concrete paths without considering the painting node as an upward node, increasing the number of compatible concrete paths.
- *Level 3.* We remove the paths that lead to conditional nodes and keep only those leading to projecting nodes (e.g. title in Figure 19). This is the weakest level, adopted in case the previous levels keep no more solutions.

The *PreserveAncDesc* rule (Level 1) can be independently used or not when relaxing the tree structure (level 2 and 3).

However note that the results of level 2 are included in that of level 3. Thus we will be re-evaluating the same queries (at least partly). This is bad for two reasons: (i) it is a waste of CPU and (ii) users get several times the same answers. Still, starting by evaluating a relaxed query, (Level 2 or Level 3) and filtering the result to retrieve Level 0 (i.e. the original query) is also not satisfactory. Indeed, it implies working more than necessary (if Level 0 returns enough answers) and also, storing intermediate data structures. Remember that the processing of queries in Xyleme requires (near to) zero memory thanks to pipelining. We want to keep this nice property.

We need to perform more experiments on relaxation and see whether/how it can be supported.

7 Maintaining Views

We now explain how view definitions are maintained, and in particular how they are updated and installed. The Web is dynamic and DTD's are constantly added and modified. Consequently, the view definitions in which those DTD's may be involved need to be updated accordingly.

Let us consider the Xyleme architecture presented in Figure 8. On each interface machine, runs a process called *global process* in which queries are compiled and controlled. On all index machines, we find a process called *local process* in which local subplans are evaluated (see Figure 20). Views are managed by one object called *view manager* that is global (on interface machines) or local (on index machines).

When a query needs to access a view structure, it asks the corresponding view manager for a navigator (for trees) or an accessor (for tables). These are in fact simple objects pointing to the view data structures.

Within both global and local processes, we find a Corba server called *view server* (bottom part of Figure 20). View servers are accessed when one wants to install or update a view. They are mono-threaded objects, i.e., we support one view update or install at a time.

We now present the sequence of operations to update and to install a view. Both have to be done on global (interface) machines by the global processes and on local (index) machines by the local processes.

7.1 Update a View

Updates are performed off line using the persistent representation of the view. To perform an update, one sends

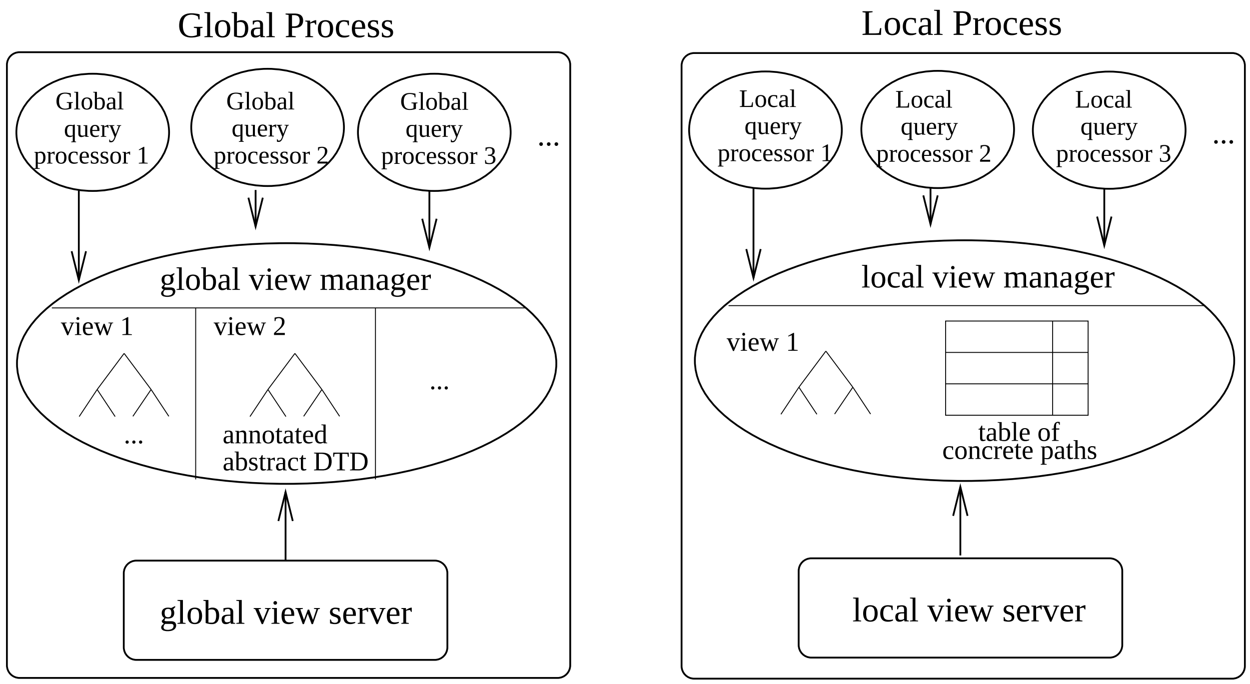


Fig. 20 Global and Local Processes in Xyleme

a message to the *global view server* with: (i) the name of the view and (ii) a file containing the new mappings. The update can be described by the following sequence of operations.

Global View Server

- Fetch the XML document containing information concerning the view (e.g. the names of its associated *local view servers*);
- Install in memory the tree structure connecting abstract paths to clusters (e.g. the annotated abstract DTD shown in Figure 10). Update the tree of abstract paths with the new mappings contained in the given file and partition the mappings according to clusters;
- For each cluster concerned by the new mappings, send an update message to the relative *local view server*(s) with the cluster name, the view and the concerned abstract to concrete mappings, and wait for an acknowledgment;
- If one local update goes wrong (e.g., a server is down), return failure (the persistent representation of the view is then inconsistent; only a successful replay of the full update will make it consistent again). Otherwise, save the new information concerning the view and return success.

Let us now follow the sequence of operations on an index machine when the local view server receives an update message.

Local View Server

- If it is not the first update concerning this view and this cluster, fetch the XML document that stores the previous updates and install the corresponding data structures in memory (e.g. the ones of Figure 11 [2]);
- Read the new mappings, and update the in-memory representation of mappings;
- Save the new view information and return success.

Once this process is completed, the new version of the view has to be installed in main memory to be used by future queries. The old image remains until all queries that pointed to it have been evaluated.

7.2 Install a View

In order to install the new version, one has to send an installation message to the *global view server*. In the following the sequence of operations executed by the Global view server.

Global View Server

- Fetch the XML view document and send an install message to all concerned *local view servers*;
- If one fails, return failure (this time, it is the distributed main memory representation of the view that is inconsistent. Some local subplans will be evaluated using the old version, others will use the new. In most probability, the user will not see the difference. Again, a successful replay of the installation

[2] In fact, the update representation of the table of concrete paths is slightly more complex than that used by queries. Notably, it allows to navigate in all directions.

will correct the situation). Otherwise install in memory the view data structure;
- Ask the *view manager* to swap this structure with the one it is currently managing. When no more queries use the old structure, delete it and return success.

As previously, let us follow the process on an index machine when the local view server receives an install message from the global view server.

Local View Server

- Fetch the XML document containing the local view and install the view structures in memory;
- Ask the *view manager* to swap this structure with the one it is currently managing;
- Only when no query uses the old structure, delete it and return success.

Finally we remark that an old version of the views is maintained by both global and local view managers, until no more queries address it.

8 Conclusions

The Xyleme project is building a data warehouse capable of storing the XML data available on the Web. Xyleme uses the properties of XML to provide database-like services that are otherwise difficult or impossible to realize with the current Web technologies: automatic data integration, structured queries, change notification, etc.

In this paper, we presented a view mechanism for Xyleme. We addressed the problems of defining, storing, maintaining and using views in a Web scale database. The work has been fully implemented (including joins) and is currently being tested. The next step is to develop appropriate benchmarks and tune the query translation strategy based on the results of such a benchmark.

Acknowledgments We would like to thank the whole Xyleme team and Claude Delobel, Marie-Christine Rousset and Catriel Beeri for many fruitful discussions. We also thank Jean-Pierre Sirot for discussions on mapping generation, Irini Fundulaki for useful remarks on query translation and Laurent Mignet for helpful suggestions. Finally we are deeply grateful to Serge Abiteboul for his useful remarks.

References

1. Serge Abiteboul, Jason Mc Hugh, Michael Rys, Vassilis Vassalos, and Janet Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, New York City, NY, USA, 1998.
2. Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.
3. V. Aguilera, F. Boiscuvier, and S. Cluet. Pattern Tree Matching for XML Queries. Technical report, INRIA, 2001.
4. V. Aguilera, S. Cluet, and F. Wattez. Xyleme Query Architecture. In *Proc. of the Int. WWW Conf.*, 2001. Hong-Kong.
5. B. Amann, C. Beeri, I. Fundulaki, M. Scholl, and A-M. Vercoustre. Mapping XML Fragments to Community Web Ontologies. Fourth Int. Workshop on Web and Databases (WebDB), Santa Barbara California, May 2001.
`http://cedric.cnam.fr/PUBLIS/RC255.pdf`.
6. B.Amann, C. Beeri, I.Fundulaki, M. Scholl, and A-M. Vercoustre. Rewritting and Evaluating Tree Queries with XPath. In *Proc. of the 17ièmes Journées Bases de Données Avancées (BDA)*, Agadir, Maroc, November 2001.
7. D. Beneventano, S. Bergamaschi, S. Castano, A. Corni, R. Guidetti, G. Malvezzi, M. Melchiori, and M. Vincini. Information Integration: The MOMIS Project Demonstration. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2000.
8. Biztalk.
`http://www.biztalk.org`.
9. Peter Buneman, Susan B. Davidson, Gerd G. Hillebrand, and Dan Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, 1996.
10. S. Castano and V. De Antonellis. A Schema Analysis and Reconciliation Tool Environment. In *Proc. of the Int. Database Engineering and Applications Symposium (IDEAS)*, Grenoble, 2001.
11. R. G. Cattell. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.
12. Vassilis Christophides, Sophie Cluet, and Jérôme Siméon. On Wrapping Query Languages and Efficient XML Integration. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Dallas, Texas, May 2000.
13. C. Delobel, C. Reynaud, M. Rousset, J. Sirot, and D. Vodislav. Semantic Integration in Xyleme: a Uniform Tree-based Approach. *Data and Knowledge Engineering Journal (DKE)*, To appear, 2002.
14. Alin Deutsch, Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. XML-QL: A Query Language for XML. In *Proc. of the Int. WWW Conf.*, Toronto, May 1999.
15. Philip A. Bernstein Erhard Rahm. A survey of approaches to automatic schema matching. *Very Large Data Bases (VLDB) journal*, 10(4):334–350, 2001.
16. 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.
17. Miller G.A. WordNet: a lexical Database for English. *Communication of the ACM*, 38(11), November 1995.
18. H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagivand J. Ullman, and J. Widom. The tsimmis project: Integration of heterogeneous sources. *Journal of Intelligent Information System*, 8(2):117–132, march 1997.
19. L. Haas, D. Kossmann, E. Wimmers, and J.Yang. Optimizing Queries across Diverse Data Sources. In *Proc.*

Int. Conf. on Very Large Data Bases (VLDB), Athens, Greece, 1997.
20. Alon Halevy. *Logic-based techniques in data integration*, volume 597. The Kluwer International Series in Engineering and Computer Science, 2000.
21. T. Kirk, A. Levy, Y. Sagiv, and D.Srivastava. The Information Manifold. In *AAAI Spring Symposium on Information Gathering*, 1995.
22. Zoé Lacroix, Claude Delobel, and Philippe Brèche. Object views and database restructuring. In *Int. Workshop on Database Programming Languages (DBPL)*, Estes Park, Colorado, August 1997. Springer.
23. W. Li and C. Clifton. SEMINT: a Tool for Identifying Attribute Correspondance in Heterogeneous Database using Neural Network. *Data and Knoweledge Engineering*, 33(1), 2000.
24. J. Madhavan, P.A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *VLDB*, Rome, Italy, sept 2001. Morgan Kaufmann.
25. David Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.
26. I. Manolescu, D. Florescu, and D. Kossman. Answering XML Queries over Heterogeneous Data Sources. In *VLDB*, Rome, Italy, sept 2001. Morgan Kaufmann.
27. Oasis. Organization for the Advancement of Structured Information Standards.
`http://www.oasis-open.org`.
28. L. Palopoli, G. Terracina, and D. Ursino. The System DIKE: Towards the Semi-Automatic Syntehsis of Cooperative Information Systems and Data Warehouses. In *ADBIS DASFAA*, 2000.
29. C. Reynaud, J.P. Sirot, and D. Vodislav. Semantic Integration of XML Heterogenneous Data Sources. In *Proc. of the Int. Database Engineering and Applications Symposium (IDEAS)*, Grenoble, 2001.
30. S.Cluet, C.Delobel, J.Simeon, and K.Smaga. Your Mediators Need Data Conversion! In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Seattle, 1999.
31. J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. Querying XML Views of Relational Data. In *VLDB*, Rome, Italy, September 2001. Morgan Kaufmann.
32. J. Ullman. Information Integration Using Logical Views. In *Proc. of the Int. Conf. on Data Transaction*, Delphi, Greece, January 1997.
33. J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I.* Computer Science Press, 1988.
34. Wordnet- a lexical database for english.
`http://www.cogsci.princeton.edu/ wn/`.
35. W3C XML Query Working Group. W3C XML Query Requirements. Technical report, W3C,
`http://www.w3.org/TR/xmlquery-req`, February 2001.
36. W3C XML Query Working Group. W3C XML Query Requirements. Technical report, W3C,
`http://www.w3.org/TR/xquery/`, June 2001.
37. Xyleme s.a.
`http://www.xyleme.com`.
38. Lucie Xyleme. A Dynamic Warehouse for XML Data of the Web. *IEEE Data Engineering Bulletin*, 24(2):40–47, 2001.
`http://osage.inria.fr/verso/PUBLI/`.