Views in a Large Scale XML Repository

Sophie Cluet
Xyleme S.A.
St Cloud France
Sophie.Cluet.@xyleme.com

Pierangelo Veltri
INRIA VERSO
Rocquencourt France
Pierangelo.Veltri@inria.fr

Dan Vodislav
Cedric/CNAM
Paris France
vodislav@cnam.fr

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 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.

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, the Xyleme [16] project [17] is building a warehouse which will store and provide sophisticate database-like services over all the XML documents of the Web. 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., [4, 12]), 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 schemata.

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 one 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 subclusters. In other words, the size of a view definition in Xyleme is orders of magnitude larger than that of a relational view. Obviously, techniques 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 [13]. 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

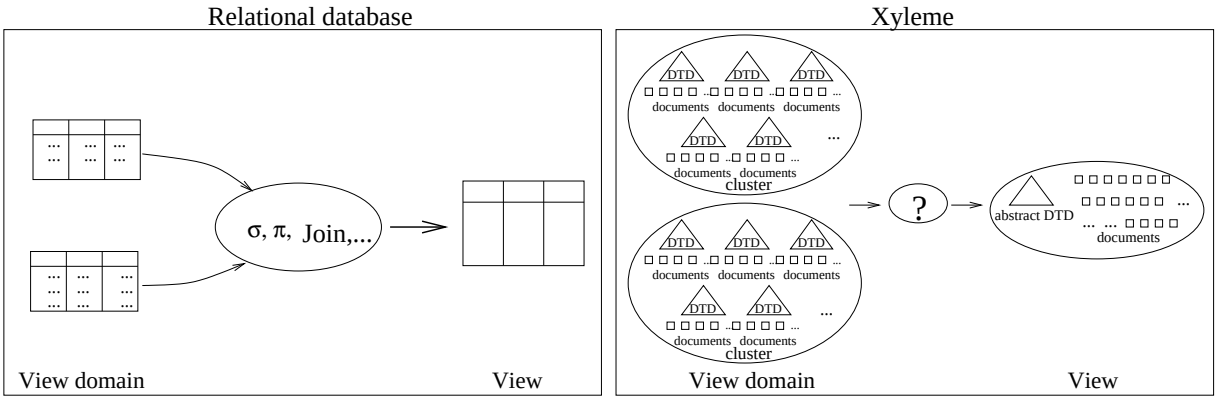

Figure 1: Relational Vs. Xyleme Views

the process, the user can interact to refine the definition. Whereas a human cannot cope with large scales, softwares do 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 three main issues: (i) view definition model, (ii) view implementation and distribution, (iii) translation of queries against views. 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.

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 [11, 14], object [10], semistructured [1] databases or mediators [9, 7], but they are not adapted to heterogeneity on a large scale.

The paper is organized as follows. In Sections 2 to 4, we address the problems of, respectively, (i) definition, (ii) implementation and (iii) translation of views. In Section 5, we extend our work by introducing joins in query translation and considering query relaxation. Finally, Section 6 concludes the paper.

2 View Definition

We first briefly introduce the Xyleme query language before discussing the view definition model.

2.1 Queries

The Xyleme query language is an extension of OQL [6] with path expressions and the full text *contains* predicate. It is consistent with the requirements published by the W3C XML Query Working Group [15] and similar to many languages proposed by the database community for text or semistructured databases (e.g., [2, 5, 8]). The most basic query filters a collection of XML documents according to some predicates. For instance, consider the following query that retrieves the titles of van Gogh's paintings exposed at the Orsay museum.

Example 2.1

select p/title
from *doc* **in** culture,
p **in** *doc*/painting,
where p/author **contains** "van Gogh"
and p/museum **contains** "Orsay"

Note that, as opposed to standard database queries, structure is used here as an added means for filtering the documents. The collection named *culture* contains documents of many 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). Xyleme query language supports both child (/) and descendant (//) relations. In this paper, we forget about this distinction.

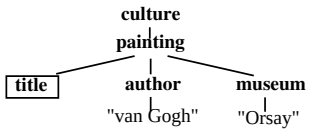

Figure 2: Query Pattern Tree

This simple kind of queries that filters a collection according to some tree pattern is called a tree query. It constitutes the basic brick of Xyleme query language. A complex 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]. Naturally, tree queries are also the basis of the view mechanism.

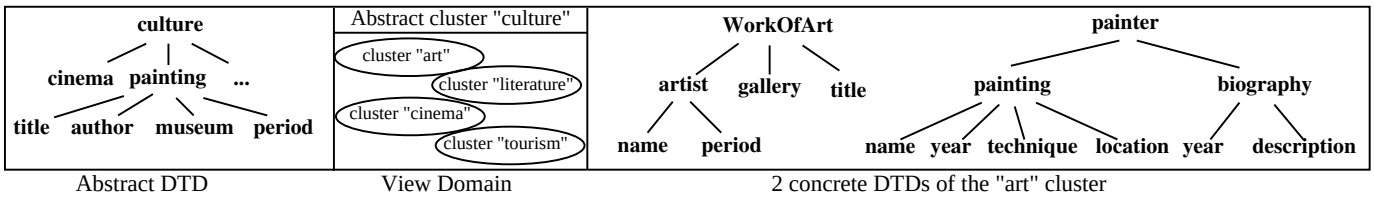

Figure 3: An Example of View over the *culture* Domain

2.2 Views

As in a relational system (see Figure 1), a Xyleme view has a domain, a schema and a definition. In the sequel, everything related to the view domain is called **concrete** 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 contains 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*). The right part of Figure 3 shows two concrete DTDs 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 DTD summary. 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 and we ignore cardinality and alternatives.

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 on the left part of Figure 3. All edges must be interpreted as 0-n (i.e., "*"). Links are interpreted as providing 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 simpler 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. In the remaining of this section, we define the semantics of such views and justify our choice by comparing it with other alternatives.

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 concepts* to *concrete terms*. An *abstract concept* is the label of a node in the abstract DTD tree (e.g., "painting" in the abstract DTD of Figure 3) and a *concrete term* is the label of a node in some concrete DTD (e.g., "WorkOfArt" or "painting" in the concrete DTDs of Figure 3). An abstract concept, respectively a concrete term, may be involved in several mappings.

To translate a query against a view, we then simply replace abstract concepts by their corresponding concrete terms. All possible combinations are considered, each resulting in a tree query. The various tree queries are unioned to form the final result.

From a storage point of view, this solution is the less costly, because concepts or terms occurring several times are factorized. It is also the easiest to generate automatically (it is simple enough to find semantic analogies between words).

However, it is also the less precise because it completely ignores the structure of the documents. This has two implications. The first is that it does not allow to capture relevant mappings. For instance, consider a query involving the abstract path: **painting/title**. Also, assume that a concrete DTD contains the path: **painting/description/title**. 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 **painting/title** would be translated to **painting//title**. But in this case, the title below **painting/description/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 to consider all tags combinations,

(i.e. k^n if k is the average number of mappings per abstract concept and n is the size of the query) many of which are useless because they do not appear in any concrete DTDs. Thus, this may become inefficient.

DTD-to-DTD (tree-to-tree) mappings

Another possibility to define views is to map abstract DTD to concrete DTD. This is illustrated by Figure 4 where dotted lines represent the correspondence between abstract nodes (on the left) and concrete ones.

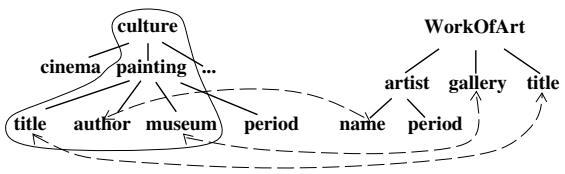

Figure 4: A Mapping from DTD to DTD

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 4 corresponds to the following Xyleme query:

Example 2.2

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

The **select** clause builds the abstract DTD subtree that is bounded in Figure 4 and indicates the connection between abstract and concrete nodes (e.g., title and t, which stands for WorkOfArt/title).

Query translation with this definition is 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 a software will ever achieve the level of precision that is required here.

Still tree-to-tree mappings offer several advantages. Notably it is the most precise method, because it preserves the interpretation context of each document structure. Unlike the others, DTD-to-DTD mappings describe the abstract representation of a whole concrete DTD. Consequently, query translation produces no useless and no irrelevant query.

Also, the query translation algorithm is very 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 6, it combines the advantages of the two previous approaches. The idea here is to preserve the context of interpretation of abstract and concrete nodes, i.e., their path from the root. This is illustrated by Figure 5 that shows a set of path-to-path mappings. Note that each mapping must be interpreted independently. I.e., 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/author− > WorkOfArt/artist/name**. This mapping simply states that the abstract concept "author of a painting" is equivalent to the information found at the end of the concrete path **WorkOfArt/artist/name**.

Query Translation is performed in two steps. First, we select the abstract concepts that are used in the query. For instance, **culture/painting**, **culture/painting/title**, **culture/painting/author** and **culture/painting/museum** in the query of Figure 2. Then, among all the possible ways of combining 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 the concrete queries built from the valid tree combinations. For instance, given the mappings of Figure 5, the query will have a translation in the DTD rooted at WorkOfArt but not in the other one that reverses the painting/painter relationship. We will see in the sequel that there exists some good reasons for that constraint (efficiency and semantics) that may look too strong in the current case. Also we will see how it can be relaxed.

```
culture/painting −−> WorkOfArt
culture/painting −−> painter/painting

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

culture/painting/author −−> WorkOfArt/artist/name
culture/painting/author −−> painter

culture/painting/museum −−> WorkOfArt/gallery
culture/painting/museum −−> painter/painting/location
                    ...
```

Figure 5: Path-to-path Mappings

Path-to-path mappings save space by factorizing

DTD similarities and allow semi-automatic mapping generation.

How mappings are generated is not the topic of this paper ([13]). In a nutshell, to each node in the abstract DTD we associate zero, one or more ancestor nodes that constitute its context of interpretation. For instance the left most title node in the abstract DTD of Figure 3, is associated to the **painting** node. then mappings are generated in two steps. First, words are mapped to words. Second, paths are mapped to paths by considering (i) their leaf words and (ii) their context. For instance, **culture/painting** is mapped to **WorkOfArt** because there is a specialization relationship between the two leaf words. **culture/painting/title** is mapped to **WorkOfArt/title** because their leaf nodes are equal and there is a mapping between the context of title (**culture/painting**) and a sub-path of **WorkOfArt/title**.

The main problem of the path-to-path mappings is precision. The descendant relationship constraint in query translation may lead to miss relevant concrete queries. Still remember that, in any case, automatic generation of mappings cannot be completely precise. As a matter of fact, the precision we miss here is typically that which cannot be infered by a 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.

As a concluding remark, notice that views as presented here map structure to structure in unconditional way. 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.

Feature / Method	Storage	Precision	Query processing	Automatic view generation
tag-to-tag	very efficient	very bad	slow	easy, imprecise
DTD-to-DTD	less efficient	best	fast	as imprecise as path-to-path
path-to-path	efficient	good	fast	acceptable

Figure 6: Comparison between Mapping Methods

3 View Implementation

In order to explain our implementation, we need to describe the distribution of data and processes in Xyleme. Thus, we next briefly describe the Xyleme machine architecture and query evaluation process. Then, we present the view implementation from two different angles: data structures and processing.

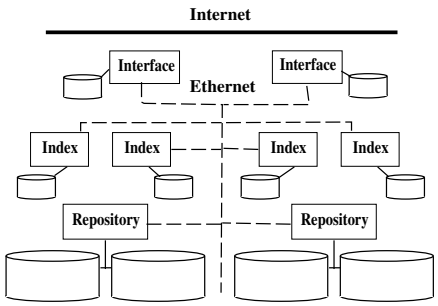

Figure 7: Xyleme Machine Architecture

3.1 Distribution in Xyleme

The Xyleme system runs on a cluster of Linux PCs. As illustrated in Figure 7, we distinguish (functionally) three kinds of machines (from bottom to top):

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 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.

Figure 8 illustrates the query evaluation process when views are not used [3]. The input query is partly optimized and compiled on one interface machine. The result of the compilation is a distributed execution plan consisting of several so-called local subplans. Each subplan is shipped to its respective machine where it is eventually further optimized and evaluated. In the example, there are three subplans. The one in charge of controlling the overall execution and sending the result back to the user stays on the interface machine. The two others are sent to, respectively, a repository and an index machine. Obviously, there are cases where more index and repository machines are involved.

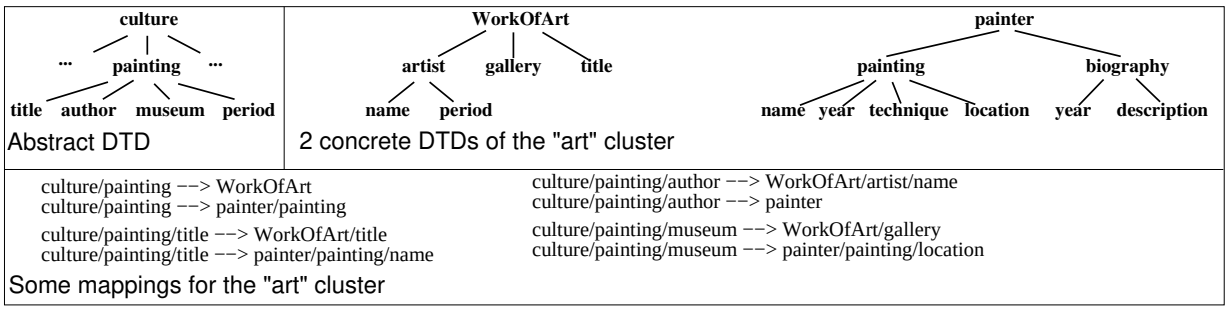

Figure 9: A View Over the *culture* Domain

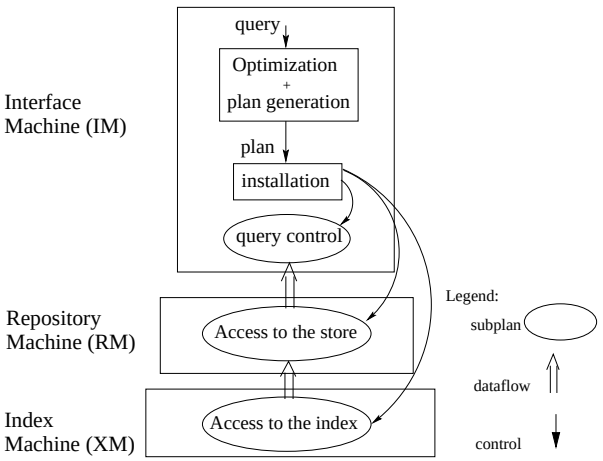

Figure 8: Query Evaluation in Xyleme

Evaluation always starts on the machines indexing the queried clusters. There, the identifiers of documents and elements that match the (sub-)query tree pattern are retrieved. E.g., assuming Query 2.1 is concrete and *culture* is a cluster, then an index on *culture* will be used to return the identifiers of documents and title that match the query pattern tree of Figure 2. The identifiers returned by the index are then shipped:

- either, to the control subplan along with some summary information, if the user asked for URLs;
- or, to another index machine if some join operation has to be evaluated (we support only index joins);
- or, to a repository machine to extract the values associated to the selected elements.

Indexes are ordered so as to allow an efficient pipeline evaluation process. This has two advantages: (i) no intermediary data structures consumes memory (only small buffers are used to pass data from machine to machine), and (ii) we can evaluate queries incrementally, eventually stopping when the user is not interested in more results.

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. So, obviously, we 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 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 Query 2.1, the pattern tree of Figure 2 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. Remember that 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 details.

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 introduced in the previous section and presented in Figure 9. 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 5 that this tree is used to support joins in view definitions.

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

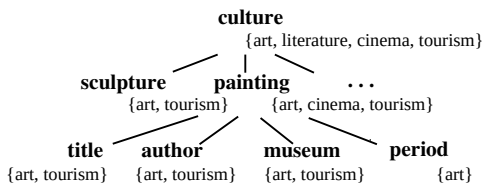

Figure 10: View on an Interface Machine

- 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 5. 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., 2 identifies **WorkOfArt/artist/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).

  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.

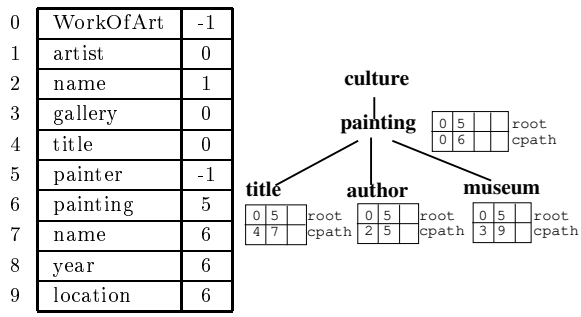

Figure 11: View on an Index Machine

Updates are performed off line using the persistent representation of the view. Once this process is completed, a new main memory image is constructed that will be used by future queries. The old image remains until all queries that pointed to it have been evaluated.

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 two 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 (see Section 2). At this stage the PatternScan is defined on an abstract pattern and on an abstract cluster. The *Abstract Query Translator* (AQT) translates the PatternScan in a union of PatternScan

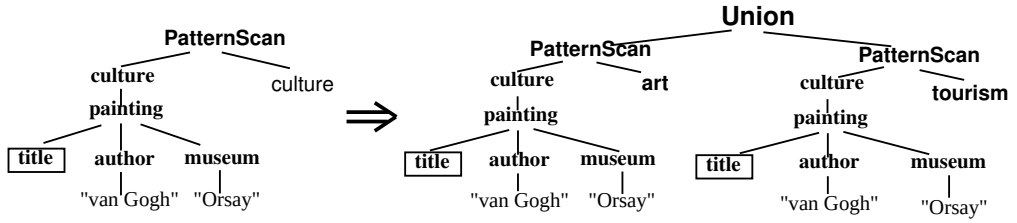

Figure 12: AQT Transformation of an Abstract Pattern Scan

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. Each generated pattern is given to a physical algebraic operator called *FTIscan* that will match it efficiently against the indexed documents and return the selected elements and documents. Eventually, those will be further processed by other operators on other machines.

We are not concerned by the full query evaluation process here (see [3]), just by the part concerning the view translation. More precisely, we describe 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.

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 5.1, 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 transforms abstract pattern trees into concrete ones. Its input is an abstract pattern and a view identifier. Its output is a flow of concrete patterns. Figure 13 illustrates this process with an abstract pattern matched against the view of Figure 9.

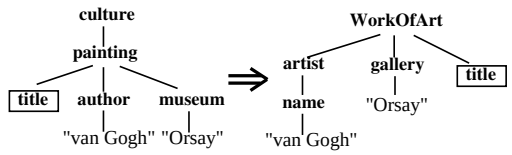

Figure 13: A concrete pattern generated by A2C

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, 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 details.

Rule PreserveAscDesc 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 an 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**). Still, we chose to impose this rule because it reduces the complexity of the A2C algorithm. In Section 5, we discuss its relaxation.

In order to further reduce the complexity, we also impose a technical rule that may seem somewhat arbitrary but is rarelly violated in practice.

Rule NoTwoSubpaths 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

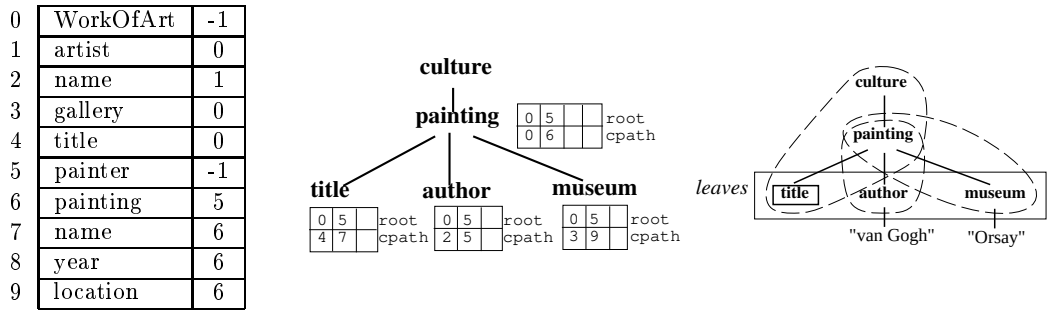

Figure 14: Upward Paths from the Pattern Leaves

only if there does not exist c_1, c_2 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. Even if exceptions may exist, this rule is naturally respected in practice.

The A2C algorithm

Consider the leftmost branch on the abstract pattern tree of Figure 13. Rule **PreserveAscDesc** 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 14 (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 Section 3 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 14, 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 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 14 (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 14 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 are 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 14). 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 concept is rarely mapped to more than one concrete path of a 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 relatively 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 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 to 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 many 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 study them briefly.

5.1 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 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 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, and assume that the concrete path corresponding to the museum link is **painting/museum**, then we add the following mapping to the view:

culture/painting/museum→painting/museum,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 (e.g., if **museum** and **painting** are part of two distinct clusters). 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 15.

The middle part of 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*. 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 right part of Figure 15 partially shows the AQT translation of the query **PatternScan** operation. 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*).

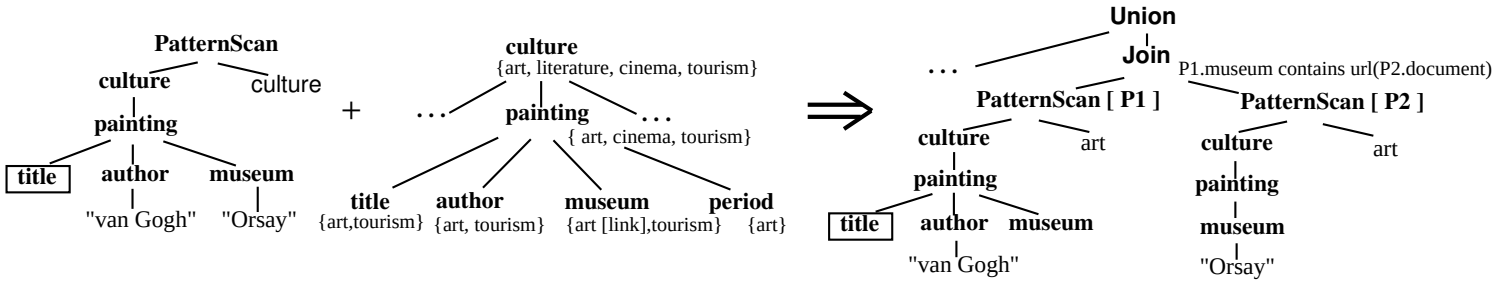

Figure 15: Introducing Joins in the View Definition

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.

5.2 Query relaxation

Joins are one possible answer 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.

Database-like queries, such as those answered by Xyleme, may miss some interesting documents because they have a slightly different structure. The goal of the query relaxation phase is to relax some of the rules that guide the translation process. Below are the relaxation policies used in Xyleme, going from stronger to weaker.

- *Level 1* discards Rule **PreserveAscDesc** from 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* removes all query nodes that are not directly needed by selections or projections. Figure 16 shows the result of applying this policy on the example query.
- *Level 3*, the weakest, is the equivalent of the keyword search provided by search engines.The abstract query tree is reduced to root, constant leaves and projection nodes (see Figure 16).

Note that the result of each level is included in that of the next. In other words, if we relax queries level by level, 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 Level 3 and filtering the result to retrieve Levels 0, 1 and 2 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.

6 Conclusion

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 and using views in a Web scale database. This work has been fully implemented (including joins) and is currently being tested. The next step will be to tune the query translation strategy given the results of the benchmark.

Acknowledgments:

We would like to thank the whole Xyleme team for many fruitful discussions and particularely Claude Delobel, Marie-Christine Rousset and Catriel Beeri. Finally we want to thank Serge Abiteboul and Tova Milo for their useful remarks.

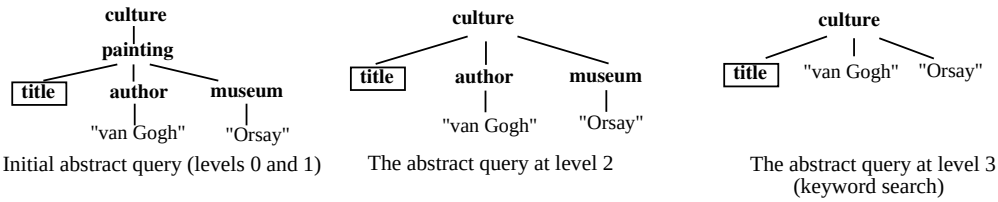

Figure 16: Query Relaxation

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)*, 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, apr 1997.

[3] V. Aguilera, S. Cluet, and F. Wattez. Xyleme Query Architecture. Proc. of the Int. WWW Conf., 2001. Hong-Kong.

[4] Biztalk. `http://www.biztalk.org`.

[5] 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*, pages 505–516, Montreal, Canada, 1996. ACM.

[6] R. G. Cattell. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[7] 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.

[8] 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.

[9] 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.

[10] Zoé Lacroix, Claude Delobel, and Philippe Brèche. Object views and database restructuring. In Sophie Cluet and Richard Hull, editors, *Database Programming Languages, 6th International Workshop, DBPL-6, Estes Park, Colorado, USA, August 18-20, 1997, Proceedings*, volume 1369 of *Lecture Notes in Computer Science*, pages 180–201. Springer, 1997.

[11] David Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[12] Oasis. Organization for the Advancement of Structured Information Standards. `http://www.oasis-open.org`.

[13] C. Renaud, J.P. Sirot, and D. Vodislav. Semantic Integration of XML Heterogenneous Data Sources. In *IDEAS*, Grenoble, 2001.

[14] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I.* Computer Science Press, 1988.

[15] W3C XML Query Working Group. W3C XML Query Requirements. Technical report, W3C, http://www.w3.org/TR/2000/WD-xmlquery-req-20000131, January 2000.

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

[17] Lucie Xyleme. A Dynamic Warehouse for XML Data of the Web. *IEEE Data Engineering Bulletin*, 2001. http://osage.inria.fr/verso/xyleme/short_paper.htm.