Querying XML Documents in Xyleme

Vincent Aguilera, Sophie Cluet, Pierangelo Veltri, Dan Vodislav, Fanny Wattez
INRIA, projet Verso
BP 105, 78153 Le Chesnay Cedex
France
email: name.surname@inria.fr

1 Introduction

XML [4] is a standard to exchange structured data over the Web. It is being widely adopted. Undoubtedly, more and more data on the Web will be in XML form, as more and more DTDs will be available to describe it (see [16, 3]). Communities (scientific, business, others) will define their own DTD to provide for a uniform representation of data in specific areas. So we strongly believe that a very large and content-wise portion of the Web will soon be open and available in XML. Given this, the Xyleme project[1] [22] proposes to study and build a *dynamic World Wide XML warehouse*, i.e., a data warehouse capable of storing all the XML data available on the planet.

Thanks the properties of this emerging language, Xyleme will be able to deliver very high level services which are difficult or impossible to support with the current Web technologies. Notably:

- A smart and user friendly research engine, able to answer queries relying on the structure of the XML documents but also, to synthesize information coming from distinct documents,
- Version management and temporal queries, allowing to follow the evolution of the XML documents,
- Query subscriptions to notify end-users about interesting events in the repository,
- Semantic data integration to provide a single view of heterogeneous, though semantically related, data.

Among the many problems addressed by the Xyleme team, we focus in this paper on query processing. The query language supporting Xyleme graphical user interface is an extension of OQL [6, 10] and provides a mix of database and information retrieval characteristics. It is consistent with the requirements published by the W3C XML Query Working Group [20] and similar to many languages proposed by the database community for text or semistructured databases [1, 5, 7, 11]. Still, due to the fact that we are considering millions of documents, query processing acquires in Xyleme a truly distinct flavor. Although we rely mainly on database optimization techniques (notably, the use of an algebra as in [5, 8, 9, 12, 15]), we need to adapt them to meet new and interesting challenges:

[1] The Xyleme project is functioning as an open, loosely coupled network of researchers. The Verso Group at INRIA was at the origin of the project together with F. Bancilhon, formerly CEO of O_2 Technology. The database groups from Mannheim U. and the CNAM-Paris as well as the IASI Team of University of Paris-Orsay, rapidly joined as well as a number of individual researchers from a number of places.

- **Queries against Views:** We want to free users from having to deal with specific DTDs when expressing queries, when typically there will be *many* DTDs available for the specific domain of interest. Views are used in Xyleme to provide a simple and clever interface to a huge number of documents described by heterogeneous DTDs. As opposed to a typical database situation where a view is defined by a simple query against large collections of data, views in Xyleme involve many queries (because there are many different DTDs) against possibly small collections of documents.
- **Full-Text Indexes:** Obviously, since we are dealing with large numbers of documents, full text indexes should be at the core of our system. As in [21], we extend this technology so as to be able to answer structured queries, rather than simple keywords search. More precisely, we annotate each occurrence of a word in a document with some code that allows to position it relatively to the other words. To fit Xyleme needs, this words encoding must provide some good properties such as (i) density, (ii) updatability or (iii) fast selective access to the repository.
- **Distribution and Algebraic Optimization:** Distributed database systems have existed for quite some time and most of them rely on algebraic optimization. Still, no database algebra provides the pattern matching facilities that we need to, e.g., express the following query: *Find the documents in which there is a product somewhere below a company whose name resembles "Xyleme".*
  Furthermore, data distribution in Xyleme is guided by requirements that are not those of traditional database systems (e.g., the need to return the first answers fast).
- **Incomplete and Approximative Answers:** One intended use of Xyleme is as a sophisticate search engine. As opposed to database programs, people searching the Web are usually very impatient, do not care for large answers and certainly do not accept "no" as a satisfactory answer. Thus, we must be able to process partial answers fast and be able to relax a query when its result is too small. Although provided by full text indexes, these functionalities raise many non trivial problems when queries are distributed, involve structure and combine information from several documents.

The Xyleme project is still in its infancy (it started in September 1999) and we do not have a solution to all the above problems. Still, we have partially addressed them and started implementing Xyleme query processor. In this paper, we relate this work. The paper is organized as follows. In the next section, we briefly present Xyleme architecture. Then, Section 3 introduces the concept of views and shows how it is used to support the query interface. Section 4 explains the index encoding scheme. Finally, Section 5, sketches the full query evaluation process.

2 An Overview of Xyleme Architecture

A complete description of the system architecture is out of the scope of this paper. We give here only a brief and partial description of Xyleme main software components as well as the machine configuration of the running system.

The functional architecture of Xyleme is given in Figure 1. The lowest layer consists of the XML *repository* and *index manager*. The repository, called Natix, is tailored for tree-data and is developed at Mannheim University [14]. The index manager provides two kinds of indexes: (i) standard database B-trees for indexing, e.g., dates or integers, and (ii) full text indexes.

Above the repository, on the left hand side, are the three modules in charge of populating and updating the XML warehouse. Since the functionalities of both *crawler* and *loader* should be

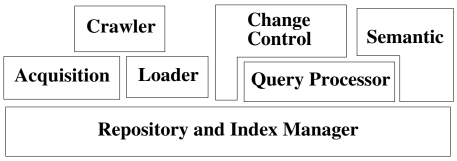


Figure 1: Xyleme Functional Architecture

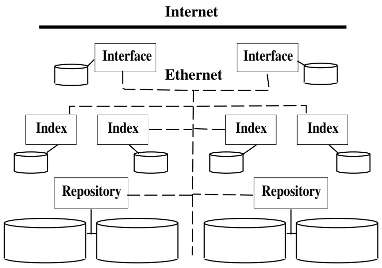


Figure 2: Xyleme Machine Architecture

obvious, let us just say a few words about the *data acquisition* module. Its task is to decide when to refresh a document in Xyleme. This decision is based on criteria such as the importance of a document (evaluated by Xyleme in a manner similar to that used by Google [13]), its size, etc.

On the right-hand side, we find the *query processor* that is the subject of the present paper. Above it are the *change control* and *semantic* modules. The former adds a temporal dimension to Xyleme. It is in charge of (i) deciding which documents to version, (ii) managing query subscriptions, and (iii) pre-processing temporal queries. The *semantic* module is in charge of analyzing documents and defining views. More precisely, it partitions Xyleme in *domains* and, for each of them, provides a structured tree of concepts that can be queried directly by users.

Xyleme will run on a cluster of Linux PCs (see Figure 2). We (logically) distinguish between three kinds of machines: (i) repository machines responsible for a large collection of XML documents, (ii) index machines with enough main memory to fit the indexes they are maintaining (i.e., there will be several index machines for one repository machine) and (iii) interface machines connected to Internet, in charge of running Xyleme applications and of dispatching tasks/processes to the other machines. Control messages are shipped thanks to Orbacus 4.0 [17], data is shipped efficiently and transparently by Natix.

3 Queries Against Views

Queries in Xyleme are more than simple keyword searches: they involve an analysis of the document structure and, in some cases, of information coming from several documents. For instance, consider the following query:

Example 3.1
select p/name
from d1 **in** culture,
p **in** d1//painting,
d2 **in** tourism,
m **in** d2//museum
where m **contains** "Amsterdam" **and** p/exhibition **contains** d2

(The // and / operators have the XQL [18] semantics, *culture* and *tourism* denote collections of XML documents.) The query retrieves, from the *tourism* collection, documents with a *museum* element that contains "Amsterdam". Then it extracts the information found below *painting/name* in *culture* documents if, below its *exhibition* sub-element, *painting* contains a reference to one of the selected tourism documents.

Note that even with the help of a sophisticated interface, one can hardly expect an end-user to formulate the above query. Indeed, it requires that one (i) knows fairly well the structure of the documents managed by Xyleme (and we hope there will be millions of them) and (ii) understands the connections existing between these documents. This is where Xyleme semantic module steps in. How this module operates is out of the scope of the current paper. Let us just say that, using (i) a thesaurus, (ii) data mining techniques and (iii) some human interaction, it is able to partition the documents in domains (e.g., the *culture* collection of documents) and provide abstract summaries that can easily be queried by end-users. The connection between summaries and real documents is called a view.

3.1 Views in Xyleme

Views connect abstract summaries to real documents. A summary is identified by a name (e.g., culture) and is simply defined as a tree of concepts. The left-hand side of Figure 3 shows a part of the cultural summary. (Note that the tree root node is named after the summary.) A view consists of a collection of mappings between paths in an abstract summary (aka **abstract paths**) and paths in some concrete documents (aka **concrete paths**). This is illustrated by the right part of Figure 3. (Note that each concrete path starts with the name of a collection of documents, e.g., *culture* or *tourism*). Although views in reality are clearly more complex (involving many mappings, composite or abbreviated element names, DTD identifiers, etc.), this example illustrates some interesting features of Xyleme views:

- An abstract path defines a concept, a concrete path a collection of elements corresponding to this concept. In many cases, the associated elements do not exactly match the concept but are either more general or more specific. Still, each mapping gives a rather good indication of where to find the information relative to a concept.
- It is sometimes difficult to assign a document to a domain. This is why there is not a strict correspondence between abstract summaries and collections of documents. This situation is illustrated here with the last mapping that involves documents contained in the *tourism* collection.
- Some concrete paths ends with a reference to a document (e.g., the fifth mapping). By recording this information, we are able to consider queries that require information coming from multiple documents as illustrated by Example 3.1.

3.2 Querying Views

Xyleme graphical user interface relies on abstract summaries to help users formulate queries. Below is an example of a simple query against the *culture* abstract summary.

Example 3.2
select p/name
from p **in** culture/painting,
where p/museum **contains** "Amsterdam"

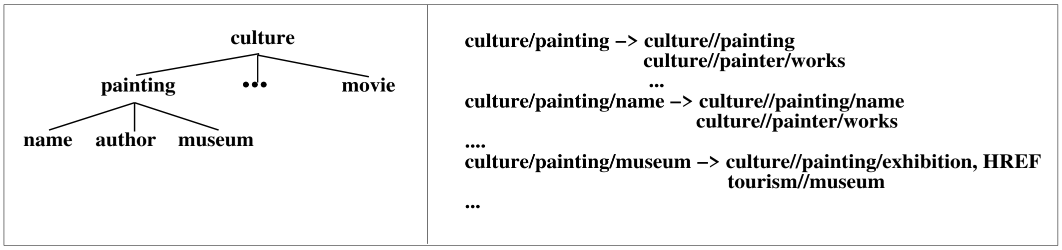


Figure 3: A View in Xyleme

Queries against summaries (aka *abstract* queries) must be translated into queries against actual data to be processed. For instance, the above query is translated into a union of *concrete* queries, one of which is that of Example 3.1 (obtained using the first, third, and last two mappings). For lack of space, we do not detail here the translation process. We only give some intuition of how views are managed by the system and queried by the query processor.

Since Xyleme is based on a distributed architecture, communication between machines is a major performance issue. In order to minimize communications, both in number and in size, we split the information/processing relevant to a view into two parts.

- As briefly explained in Section 2, a query enters Xyleme on an arbitrary interface machine but is actually evaluated on the machines indexing the documents that have to be queried (eventually also on the machines storing the documents). Thus, to avoid addressing irrelevant index machines, the translation from abstract summaries to actual collections of documents must be made on the interface machine. In other words, the correspondence between summaries and collections must be stored on all interface machines.
- However, there may be thousands of concrete paths associated to one concept. Thus, keeping all mappings on every interface machine would lead to (i) loss of scalability (every new document potentially adds new mappings and the information is not distributed) and (ii) large communications between semantic and index machines. Accordingly, we partition the correspondences between abstract and concrete paths according to the collections over which the concrete paths are defined and store them on the appropriate index machines.

To summarize, an abstract query is translated in two steps. First, we generate queries with abstract paths over real collections of documents. These queries are then pre-processed locally and code is generated to be evaluated on the appropriate index machines (see Section 5). The second step occurs on each index machine where we translate trees of abstract paths into trees of concrete paths before querying the indexes.

4 About Full Text Indexes

The optimization of queries in database management systems relies on the fact that data is strongly typed. Typically, a database application consists of a strict schema whose size is several orders of magnitude smaller than that of its associated data. We are considering here the opposite situation: (i) XML does not provide strong typing (notably, due to the alternative | separator found in DTDs and the notion of well-formedness) and (ii) the size ratio between DTD and validating documents is rather big (nearing one in some not-so-rare cases). Thus, indexes as found in database systems are inappropriate.

Since we are manipulating documents, full-text indexes seem a reasonable solution. Traditionally, these structures associate to each indexed word, the documents in which the word occurs and, eventually, its position(s) in the document(s) (offset or relative position). This is fine for answering (possibly sophisticated) keyword searches but is lacking when considering Xyleme query language. Indeed, our goal is to support efficiently thousands of simultaneous database-like queries over millions of documents. The only way we can reasonably expect to achieve this goal is to maintain structures in main memory that: (i) cover the whole database, (ii) allow to answer fast the most frequent queries, and (iii) minimize access to the secondary memory when this is un-avoidable. This main memory approach is followed by search engines such as Altavista ([2]) or Google ([13]). However, given Xyleme graphical interface and semantical views, we expect our most frequent queries to be structured queries involving joins over collections of documents.

More precisely, we must encode words in the full-text index so as to satisfy the following requirements.

Structural Encoding: In order to be able to answer Query 3.1 fast, the index encoding must capture the structure of the documents.

Density: Obviously, since we intend to keep our indexes in memory and we do not want to manage more index machines than necessary, the words encoding should be dense.

Updatability: The update of a full-text index is an expensive operation. We hope that, at some point in the future, Xyleme will be (re-)loading hundreds of thousands of documents every day. Thus, minimizing the update cost is vital. One way to do this is to make a **diff** between old and new versions of existing documents and to add/remove only the elements that have been changed. This means that the insertion/deletion of one element in a document should not invalidate the codes of all others.

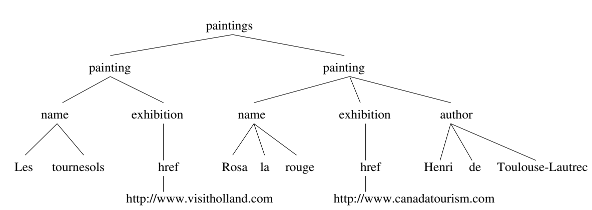
Fast Access to the Store: Some queries require that we retrieve the value of some elements, i.e., that we access the documents themselves. This is the case of Query 3.1., that extracts *painting/name* elements. Indexes can obviously be used to filter the documents that will be fetched from the store. An additional and interesting use of indexes would be to give direct (or at least, accelerated) access to the element itself (remember that the Natix store is structured).

It is relatively easy to find solutions to these various requirements independently. However, we did not find yet a scheme having all the required properties. We believe this is still an open and interesting issue. Currently, we are experimenting with two encoding schemes that allow to answer structured queries. Our goal is to collect some real data about the codes/indexes size, the cost of *bad* updates and *un-helped* access to the store. We present these two encoding schemes next.

4.1 Encoding Words

We distinguish two kinds of XML documents: those that are generated from some structured data source and the others (that we call *real* documents). An interesting characteristic of the former is that, if we consider their tree representation, the number of element/attribute nodes is usually more or less equal to the number of leaves/words. Indeed, consider an XML representation of a relation with $\boldsymbol{n}$ rows and $\boldsymbol{a}$ attributes The chances are that we will find $\boldsymbol{n} \times \boldsymbol{a}$ (resp. $\boldsymbol{n}$) internal nodes representing the attributes (resp. rows), and $\boldsymbol{n} \times \boldsymbol{a}$ words representing their values. On the contrary, in *real* documents, the number of internal nodes as compared to the number of words is insignificant.

```
<paintings>
  <painting>
    <name> Les tournesols </name>
    <exhibition href = "http://www.visitholland.com"/>
  </painting>
  <painting author = "Henri de Toulouse-Lautrec">
    <name> Rosa la rouge </name>
    <exhibition href = "http://www.canadatourism.com"/>
  </painting>
</paintings>
```



paintings	d1	(1,10)
painting	d1	(2,4) ; (6,9)
name	d1	(3,1) ; (7,5)
exhibition	d1	(4,3) ; (8,7)
href	d1	(5,2) ; (9,6)
http://www.visitholland.com	d1	(5,2)
...		
rouge	d1	(7,5)
...		

Prefix-Postfix encoding of d1

paintings	d1	1	
painting	d1	1.1	{2}
name	d1	1.1.1	
exhibition	d1	1.1.2	
href	d1	1.1.2.1	
http://www.visitholland.com	d1	1.1.2.1	[1]
...			
rouge	d1	1.1.1	[2]
...			

Bit-structured encoding of d1

Figure 4: Indexing a Document in Xyleme

Our two encoding schemes are designed to capture these particularities. The bit structured scheme factorizes the structure of a document: this is very important for database-like documents. The prefix-postfix scheme[2] is denser but replicates internal nodes. This should not be a problem for *real* documents. In both schemes, we encode elements and attributes the same way and associate to words the encoding of their element/attribute parent[3]. This is illustrated in Figure 4 that shows, in the upper part, an XML document and its tree representation and, in the lower part, the index entries corresponding to the bit structured (right) and prefix-postfix (left) schemes.

The Bit Structured Scheme encodes words according to their "structural" position in a document. The positional information consists of two parts. (i) The first is computed from the DTD (or any structural summary) and indicates, e.g., that *name* is below *painting* in a document (1.1 is a prefix of 1.1.1). (ii) The second part is computed from the document itself and indicates, e.g., that there are two *painting* elements ({2}) and that "Rosa" is below the *name* of the second *painting* ([2]). These two parts are then coded as a sequence of bits.

Note that: (i) This encoding lacks density but should minimize both the size and the access to the index when database-like documents are considered (since internal nodes are entered only once in the index); (ii) Updates can only occur within the space that is lost due to the bits representation (e.g., when 4 bits are used to encode 5 situations). Obviously, this is not a satisfactory solution. One possibility is to loose some more space adding extra bits at "important" places (e.g., the number of rows in a relation) and to re-compute all codes when an inserted elements cannot be coded (which hopefully will not happen too often); (iii) Whether this code provides fast access to the repository obviously depends on the way data is stored. Currently, it does not and requires some pre-processing before accessing the store.

The Prefix-Postfix Scheme encodes words according to a left-deep traversal of the document tree: every time we enter (resp. exit) a node, we give it a number representing its prefix (resp. postfix) position. Additionally, to be able to recognize that e.g., *author* is a **child** (in

[2]Special thanks go to Divesh Srivastava who suggested that we use this encoding.
[3]Some added information allows to distinguish words, elements and attributes.

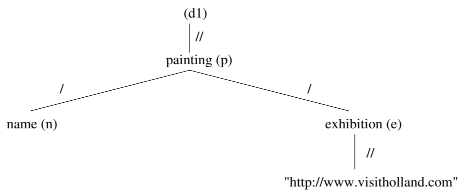


Figure 5: A Pattern Tree

opposition to descendant) of *painting*, we append the node level. We do not illustrate the child relationship here.

(i) This encoding is denser than the previous one but duplicates element nodes; (ii) In order to take updates into account, we must consider increments larger than one between the prefix (resp. postfix) number of two subsequent nodes; (iii) This encoding is more appropriate than the previous one for accessing Natix store.

4.2 Using Full-Text indexes

Now that we have encoded words, the question is: How can we check that, e.g., *rouge* is a descendant of an element *painting* containing *Lautrec*?

With the Bit Structured Scheme: both *rouge* and *Lautrec* have a unique occurrence (resp. 1.1.1[2] and 1.1.3[2]). The first part of the codes indicate that they are descendant of a *painting* (1.1 is a prefix of 1.1.1 and 1.1.3), the last part indicates that they descend from the same second *painting*.

With the Prefix-Postfix scheme: there exists an occurrence of *painting*(6,9) whose prefix (resp. postfix) number is smaller (resp. bigger) than that of both *rouge*(7,5) and *Lautrec*(10,8).

More generally, Xyleme full text indexes answer queries that are represented as trees (aka *pattern trees*) whose nodes are words/tags and whose edges indicate the relationship between two nodes (i.e., child or descendant). Furthermore, some nodes are annotated with variables indicating which information should be returned. For instance, Figure 5 shows one pattern tree that can be used to query the index over the cultural documents. It returns the document ($\boldsymbol{d1}$), painting ($\boldsymbol{p}$) and name ($\boldsymbol{n}$) identifiers satisfying the condition expressed by the pattern tree.

Note that there are many ways to evaluate this index query. For instance, one can start by processing the occurrences associated to *name* and *painting* in order to retrieve ($\boldsymbol{p}$, $\boldsymbol{n}$) couples. Then, one can check that each $\boldsymbol{p}$ contains an exhibition and, finally, check that this exhibition contains the appropriate URL. Evaluating queries one way or the other does not have the same cost. For instance, if we consider a prefix-postfix encoding, the above evaluation proposal is probably disastrous (since we can expect to have many names) and starting with the lower right part of the tree should give better results. To find out the best evaluation strategy, we apply a heuristics well-known in the database community: we try to minimize the sum of estimated intermediate results.

5 Processing Queries in Xyleme

Xyleme query language is an extension of OQL [6, 10]. Thus, to process queries, we adopt a database approach that we extend to capture the XQL-like [18] pattern matching facilities that have been

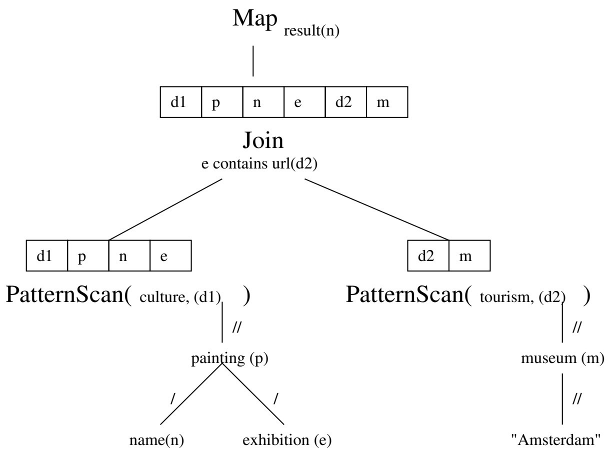


Figure 6: A logical query plan

added to the language. A query is processed as follows: (i) it is translated into an algebraic expression; (ii) relying on heuristics, we optimize this expression using algebraic equivalences and some meta-information; (iii) a distributed execution plan, notably involving full text indexes, is generated and installed; (iv) this plan is evaluated; (v) eventually, the query is relaxed to return more answers.

In this section, we give some details about the algebra and the generated execution plan. Note that we are dealing here with concrete queries (see Section 3 for queries against views).

5.1 The Algebra

There are mainly two differences between Xyleme query language and OQL. (1) We operate on XML documents that can be viewed as trees (e.g., DOM trees [19]) whereas OQL is defined on graphs of objects. (2) We extract information based on pattern matching of trees whereas OQL does not provide this facility. Obviously, (1) is not really a problem. To deal with (2), we adopt an approach similar to that found in [8, 9] and extend the algebra of [7] with one operator that we called **PatternScan**. The purpose of this operator is to extract the relevant information out of XML trees according to some pattern tree and to produce tuples. A possible implementation of this operator uses full text indexes as explained in the previous section. Figure 6 shows the algebraic translation of Query 3.1.

Going from bottom to top, the query algebraic operations are the following:

The PatternScan operations retrieve from, respectively, the *culture* and *tourism* collections the information described by their input pattern trees. Each returns a set of tuples, with one attribute per extracted piece of information (above every operator of Figure6, we see the form of its returned tuples). Note that the attributes returned by a **PatternScan** correspond to the variables of its input pattern.

The Join operation combines the information returned by the two previous operations if the value associated to Variable **e** contains the URL of the document denoted by **d2**.

The Map operation applies a function to all the elements of its input. In the example, the function is **result(n)**: it constructs an XML element named *result* whose value is given by

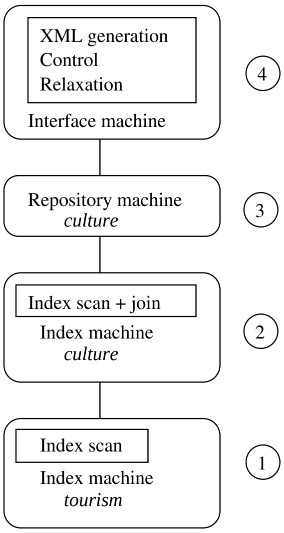


Figure 7: A distributed query plan

n. The returned elements will be collected in one document that will form the answer to the query.

5.2 Optimization and Distribution

The goal of the query optimizer is to find the best possible execution plan for a given query. This usually involves reordering of joins, some factorization, etc. Also, the optimizer decides which indexes to use and how to distribute the query. Then, an execution plan is generated. Figure 7 illustrates one possible execution plan corresponding to the logical expression of Figure 6.

From bottom to top, it consists of four subplans.

- The first subplan is executed on the machine indexing ***tourism*** (Machine 1). Using the index as explained in Section 4, it evaluates the **PatternScan** operation corresponding to the bottom-right operation of Figure 6.

- The second subplan is executed on the machine indexing ***culture*** (Machine 2). For every document returned by the first subplan, it queries the full text index using the pattern tree of Figure 5. Thus, this subplan processes two algebraic operations through iterative index scans: a **PatternScan** and a **Join**. Note that the communication between the two machines (1 and 2) is not performed on a per-document basis but uses a buffer whose size will be tuned according to our experiments.

- The third subplan is executed on the repository machine that stores the cultural documents. For every couple (***d1***, ***n***) returned by the previous subplan, it extracts from ***d1*** the element whose identifier is given by ***n*** and constructs the appropriate *result* elements.

- Finally, the last subplan is executed on the interface machine that pre-processed the query. It acts as a "chef d'orchestre" and controls all the operations.

Acknowledgments:

This research has benefited from many working sessions with people from the Xyleme team [22]. Also, we stole some good ideas about indexing to Tova Milo, Thomas Schwentick and Divesh Srivastava who visited us at INRIA. We thank them all deeply.

References

[1] 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.

[2] Altavista.
`http://www.altavista.com`.

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

[4] Tim Bray, Jean Paoli, and C.M Sperberg-McQueen. Extensible Markup Language (XML), W3C recommendation. Technical report, World Wide Web Consortium, http://www.w3.org/TR/1998/REC-xml-19980210, 10 February 1998.

[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, June 1996.

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

[7] Vassilis Christophides, Serge Abiteboul, Sophie Cluet, and Michel Scholl. From structured documents to novel query facilities. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 313–324, Minneapolis, Minnesota, May 1994.

[8] Vassilis Christophides, Sophie Cluet, and Guido Moerkotte. Evaluating queries with generalized path expressions. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 413–422, Montreal, Canada, June 1996.

[9] 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.

[10] Sophie Cluet. Designing OQL: Allowing objects to be queried. *Information Systems*, 23(5):279–305, 1998.

[11] 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.

[12] Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. A query-language and processor for a web site management system. In *Workshop on Management of Semistructured Data*, Tucson, Arizona, May 1997. In conjunction with PODS/SIGMOD.

[13] Google.
`http://www.google.com`.

[14] Carl-Christian Kanne and Guido Moerkotte. Efficient storage of xml data. In *International Conference on Data Engineering ICDE*, 2000.

[15] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, September 1997.

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

[17] Orbacus. `http://www.ooc.com/ob/`.

[18] Jonathan Robie, Joe Lapp, and David Schach. Xml query language (xql). Technical report, World Wide Web Consortium, http://www.w3.org/TandS/QL/QL98/pp/xql.html, 1998.

[19] W3C DOM Working Group. Document Object Model (DOM) level 1 specification, version 1.0, W3C recommendation. Technical report, World Wide Web Consortium, http://www.w3.org/TR/REC-DOM-Level-1/, 1 October 1998.

[20] W3C XML Query Working Group. W3C xml query requirements. Technical report, World Wide Web Consortium, http://www.w3.org/TR/2000/WD-xmlquery-req-20000131, 31 january 2000.

[21] Xdex. `http://www.xmlindex.com`.

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