Pattern Tree Queries in Xyleme

Vincent Aguiléra   Frédéric Boiscuvier   Sophie Cluet
name.surname@inria.fr

June 27, 2001

INRIA, projet Verso
BP 105
F-78153 Le Chesnay Cedex
France

Abstract Given the commitment of the major software companies to promote XML, we believe that, soon, XML editors and browsers will be largely distributed and that XML will take a lion share of the data published on the Web. The Xyleme project plans to exploit this major opportunity of radically changing the face of the Web. Building on the good properties of the new standard, we have developed a distributed system to provide high level database services over the World Wide XML Web.In this paper, we focus on the implementation of a special purpose algebraic operator, called PatternScan, using the full text index technology.

1 Introduction

There are two available technologies to query the Web: portals and search engines. Portals are built by hand and offer a very limited and quite rigid view of the Web. Search engines are built automatically providing a much better coverage. A query on a search engine is usually a keyword search that returns a set of documents ordered according to some ranking algorithm. Users then have to crawl the documents manually to find the required information. Often, results are too large and of low quality. Some recent ranking algorithms (e.g., as used in Google [7]) fairly improve the quality. Still, most users are frustrated by both poor query capabilities and poor results.

The coming of XML [1] may radically change the situation. We strongly believe that XML will soon take an important share of the Web, progressively succeeding to HTML as the preferred publication format. One key feature of XML is its ability to support high level structured queries [25].

Thus, it should be just a matter of time before we can, at last, query the Web in an intelligent manner.

The objective of the Xyleme project[1] [26] is to be ready when the XML revolution takes place. We are building a dynamic XML warehouse, fed by a systematic crawling of the Web in search of all its XML documents. Among many other features, Xyleme users will be able to query the (XML) Web as any intelligent information system. E.g., to retrieve precisely the name and revenues of companies selling XML editors. In this paper, we explain how queries are evaluated in Xyleme.

Our contribution

The evaluation of structured queries over loosely structured data such as XML has been extensively studied during the last ten years [3]. The techniques differ slightly according to the system used to store the data: object-oriented (e.g., [2]), dedicated (e.g., [20]) or relational (e.g., [18]). However, none of these approaches scale to the Web. We are considering here billions of documents (Google recently indexed more than one billion HTML documents) and millions of queries per day. So, query processing acquires in Xyleme a truly distinct flavor. In this paper, we provide a solution to this new challenge.

As in [8, 11, 12, 17, 5], we represent queries using an algebra. The novelty of our approach lies in the efficient implementation of an complex algebraic operator, named *PatternScan*, that captures so-called tree queries, i.e., queries that filter collections of documents according to some tree pattern and extract information from the selected documents. We capture the added expressive power provided by database-like XML query languages using standard operators (e.g., *Join*, *Map*, etc.).

The *patternScan* operation is implemented using an index mechanism, named *XyIndex*, that is an extension of the full text index (FTI) technology. Whereas a standard FTI returns the documents in which a word occurs, a *XyIndex* adds annotations to position each occurence of a word within a document relatively to the other words. Also, a *XyIndex* respects an order relation that allows to use efficient and pipelinable merge algorithms to combine the various sets of words occurences corresponding to a tree query.

In this paper, we introduce XyIndexes and show how they are used to evaluate tree queries. Notably, we present the join ordering and join processing algorithms that we have implemented to combine words occurences

[1]Xyleme is now a company. Originally, is was a research 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.

in the most efficient way. Our approach is validated by various experiments.

Overview of the paper

The paper is organized as follows. In the next section, we shortly present Xyleme main functionalities, its architecture and its query language. In Section 3, we present related work and give a definition of the *PatternScan* operator. Then, Section 4 introduces XyIndexes, Section 5 concerns joins implementation and Section 6 addresses the join ordering problem.

2 Xyleme

In this section, we sketch Xyleme functionalities and architecture. Then, we introduce Xyleme query language with an example.

2.1 Functionalities and Architecture

Xyleme supplies three main services: a monitoring of the Web, automatic classification/integration of documents and a query language. The latter will be described in the sequel. Here we briefly mention monitoring and classification/integration.

Xyleme crawls the Web constantly looking for new XML documents. Some documents may be versioned [19] so as to query their changes over time (e.g., to find out the evolution of the number of furniture companies over the last six months). Also, Xyleme supports query subscription [22] allowing users to be notified of some interesting events (e.g., a company change of address).

Automatic classification and integration of documents [24] is crucial considering the huge scale of the warehouse. Indeed, although some efforts are made to promote standard DTDs in some areas (e.g., [23, 6]), it is reasonable to assume that no human being will ever be able to make sense of the many different structures that will be found on the Web. Yet, structure is the key to precision in queries. This is why, Xyleme provides the means to construct simple query-able and structured views of heterogeneous, though semantically related data. This relies on two technologies. (i) The classification tool partitions the documents into semantic domains (e.g., *tourism*, *culture*). (ii) The integration tool constructs views over domains. More precisely, given one or more domains and a (simplified) DTD that represents the structure of the view, it (semi-) automatically generates mappings from paths in the view structure to paths in actual documents.

Xyleme runs on a cluster of Linux PCs (see Figure 1). We (logically) distinguish between three kinds of machines: (i) repository machines responsible for storing large collections of XML documents, (ii) index machines with enough main memory to fit the indexes they are maintaining (there may more than one index machine for a given repository machine) and (iii) interface machines connected to Internet, in charge of running Xyleme applications and of dispatching tasks/processes to the other machines.

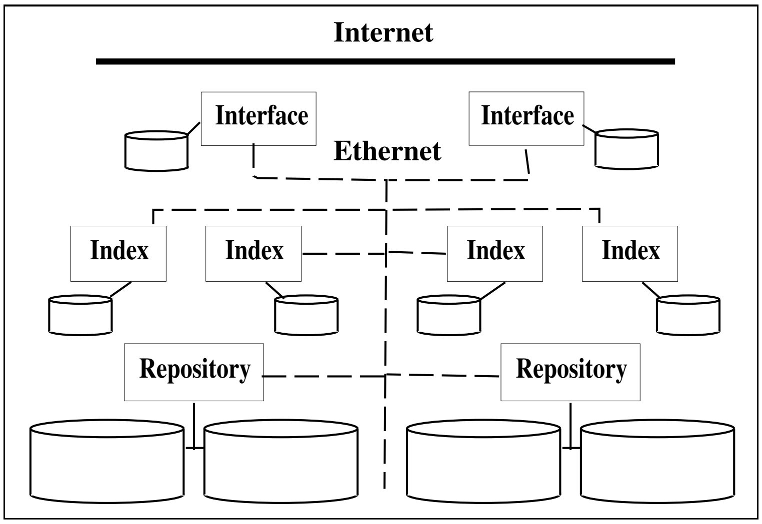

Figure 1: Xyleme Machine Architecture

2.2 The Query Language

Xyleme query language is an extension of OQL [9, 15] that provides a mix of database and information retrieval functionnalities. It is consistent with the requirements published by the W3C XML Query Working Group [25] and similar to many languages proposed by the database community for text or semistructured databases like Lorel [5], UnQL [8], POQL [10] or XML-QL [16].

In Xyleme, one queries collections of semantically connected documents that we call *domains*. Because each domain potentially contains many heterogeneous DTDs, queries are usually formulated against some unique view before being translated into concrete queries against actual data. In this paper, we explain how these latter are evaluated. The view translation problem is considered in [14].

Let us take as an example the following query:

Example 2.1

select t, y
from doc **in** cinema,
m **in** doc//movie,
t **in** m/title
y **in** m/year
where t **contains** "Pauline"
and m/author **contains** "Rohmer"

It returns the title and year of movies authored by Rohmer with a title containing "Pauline", from documents in the *cinema* domain. Its result is an XML document with as many *result* elements as answers, each having two sub-elements: *title* and *year*. The / (resp. //) operator is taken from XPath [13]: it states a parent/child (resp. ascendant/descendant) relationship between two elements (or an element and an attribute). Note the introduction of the m variable in the **from** clause to state the connection between *title, year* and *author*: they all belong to the same movie.

3 PatternScan Operator

Many previous works have adressed the problem of query evaluation on XML data. However this problem in Xyleme is guided by requirements that are not those of traditionnal database systems, notably (i) the need to return the first answer fast and (ii) since we expect thousands of concurrent queries, the need to maintain few additional structures per query. In this section, we first present related work, then introduce the $PatternScan$ operator. We will see in following sections how we adapted the inverted file data structure in order to evaluate efficiently a $PatternScan$.

3.1 Related Work

There have been many works on query evaluation for queries such as 2.1. We discuss next some that are typical of the techniques that have been proposed.

The most straightforward approach is to explore every document in *cinema*. In this spirit, [**?**] introduces the X-scan operator to evaluate regular path expressions on streaming XML data. This approach is legitimate when one does not want to store documents locally and has no knowledge about the data. However it is not appropriate when we consider large collections, even on local storage. Indeed, first experiments on Xyleme repository showed that we can hardly do better than an average rate of 500 documents per second, for various queries and sets of documents.

In [18] keyword search facilities are added to XML-QL. They propose an implementation on top of a traditional RDBMS, by storing an inverted file in the granularity of XML elements in a relation. Experimental results show that the size of the relational database is about ten times the size of the original data, which is not acceptable in our context.

In [2] an object-oriented database and query language are extended in order to efficiently evaluate path expressions. This approach mainly relies on shema informations (*structuring schema*). We believe that it does not apply in the Xyleme context since (i) many XML documents may not have even a DTD and (ii) the schema information may be huge and hard to maintain.

Finally, [21] describes a cost based query optimizer for XML data, based upon the LORE system. The query optimization process follows classical steps. First, a logical plan describing the query is built, then various physical plan are generated and compared according to a cost model. This approach relies on the various type of indexes provided by LORE, as well as on statistical information, notably statistics concerning all possible subpaths of length k. Such indexes and statistics would be hard to maintain in a repository such as Xyleme.

3.2 The PatternScan Operator

The main paradigm underlying the various query languages for XML is that of selecting data by matching patterns described with path expressions. These path expressions describe traversals along subelements with variables getting bound to nodes along these paths. Hence, a key operation in query processing over XML is to produce a set of bindings for variables, given a pattern consisting of several path expressions. The $PatternScan$ operator captures this paradigm. It filters a forest of XML documents according to some pattern and returns a set with one tuple per matching tree structure found somewhere in the forest. This is illustrated by Figure 2 that shows, on

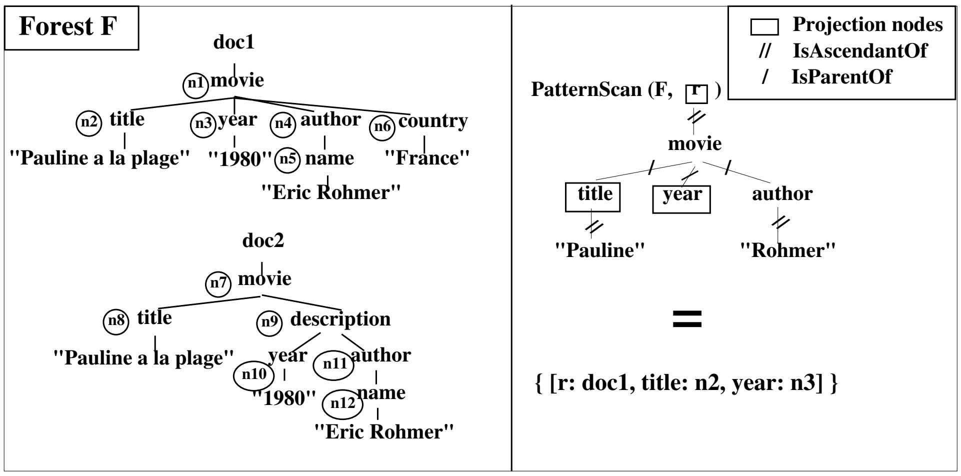

Figure 2: A PatternScan Operation

the left hand side, a forest with two XML trees whose internal nodes have been given an identifier, and, on the right hand side, a *patternScan* operation and its resulting set (with only one tuple). Let us consider the second parameter of the operation, i.e., the pattern tree. The pattern tree root node represents document roots. Other internal nodes represent elements or attributes. Leaves represent elements, attributes or text. Elements and attribute nodes are annotated by a tag name, text nodes by a constant string. Some nodes, always including the root and excluding text nodes, are marked as "projection nodes", denoting the fact that they are part of the operation result (i.e., they are represented by one attribute in the resulting tuples). The pattern tree edges represent either child or descendant relationships[2].

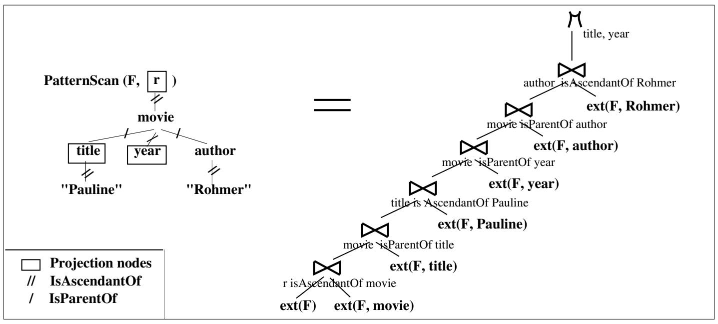

Figure 3: Translation of a PatternScan

3.3 PatternScan Definition

We define the PatternScan operation in terms of more standard algebraic operators. For this, we adopt the following notations:

Arguments of the PatternScan: F is the filtered forest and $P = (N, E)$ it the pattern tree. $N = \{n_1,\ n_2,\ \ldots\}$ is the set of tree nodes and E is the set of edges, each represented as a predicate $pred(n_i, n_j)$ where (i) n_i is the parent of n_j in the tree and (ii) $pred$ is either the *isParentOf* or *isAscendantOf* predicate.

Nodes: Wlog, we do not distinguish a node from its annotation (e.g., n is the tag/constant annotating n). Also, we call N_p the subset of N that contains all projection nodes.

Finally, we have the following functions:

$ext(F, n)$: this function returns the extension of Node n within Forest F. It is defined as:

- If r is the root node of P, $ext(F, r) = \{[r : t] \mid t \in F\}$

[2]These relationship are trivially extended to documents and constants.

- If n is a tag (resp. constant) node $ext(F, n) = \{[n : Id] \mid Id$ is the identifier in F of some element/attribute (resp. PCData) whose tag is n (resp. that contains x).

$translation_n(F, P)$: this function returns the algebraic translation corresponding to Node n of P within F. It is defined by:

- If n is a leaf, then:

$$translation_n(F, P) = ext(F, n).$$

- If n is an internal node and n_1, n_2, $\ldots n_m$ are its children, then:

$$\begin{gathered} translation_n(F, P) \\ = \\ ext(F, n) \bowtie_{pred(n,n_1)} ext(F, n_1) \bowtie_{pred(n,n_2)} ext(F, n_2) \ldots \\ \bowtie_{pred(n,n_m)} ext(F, n_m) \end{gathered}$$

We are now ready to define the semantics of the $PatternScan$ operation.
PatternScan Definition:

$$PatternScan(F, P) = \Pi_{n \in (N_p)}(translation_r(P, F))$$

where r is the root node of P. Remember that the $PatternScan$ operation returns a set. This means that the projection operation involves some duplicate elimination. Figure 3 illustrates the translation with an example.

The definition of the $patternScan$ operation relies on the existence of tag/constant extents. Extents are implemented in Xyleme by indexes that, for a given forest, associate to each tag/constant its occurences within the forest.

So, a $PatternScan$ corresponds to a sequence of joins. The problem is thus to understand (i) how to efficiently evaluate join predicates, (ii) which join algorithms to use and (iii) what is the best joins sequence. (Note that by re-ordering joins we may inverse Xyleme predicates, i.e., $isParentOf/isDescendantOf$ may become $isChildOf/isAscendantOf$). The three following sections study respectively Problems (i), (ii) and (iii).

4 Xyleme Full Text Indexes

XyIndexes are full text indexes with two specific features:

- Each occurence of a word in a XyIndex consists of (i) a document identifier, (i) a kind (element, attribute or simple word), and (iii) a code that allows to position it relatively to the other words of the document. In this way, we can, e.g., find the documents that contains the word “Pauline” below some “title” element.

- Occurences are ordered first by document identifier, then by code.

In this section, we present XyIndexes words encoding before briefly discussing the index order relation.

4.1 Words encoding

The words encoding should follow two requirements.

- We are dealing with huge amount of data and, for reasons of efficiency, we want indexes to reside in main memory. Thus, the encoding should be dense.

- Queries that construct new elements from old ones, as is the case with our example, require that we retrieve the value of some elements from the store (e.g., **year**). The encoding should then give a direct (or at least fast) access to the stored element.

Note that in comparison with standard databases, Xyleme (i) has much weaker constraints of consistency and (ii) does not update data, but replaces it. Thanks to these two particularities, index updates are not really a problem. They can be performed off-line on the persistent representation of the index, not damaging query performances. At some appropriate time, the updated index is re-loaded into memory.

The encoding scheme that is currently implemented in Xyleme and that we present here provides a rather good solution to the two requirements we have. In [4], a denser encoding is proposed but it does not fit our second requirement. The encoding relies on the fact that, notwithstanding links, XML documents are trees. The idea is then to capture the interval defining each internal node of a document to position them relatively to each other: e.g., the descendant relationship corresponds to interval inclusion. One way to achieve this is to give a distinct number to each leaf and to associate to each node the numbers of its left-most and right-most leaves[3]. Another way is to encode words according to a left-deep traversal of the tree: every time we enter (resp. exit) a node, we give it a number representing its prefix (resp. postfix) position[4]. We preferred the latter solution because prefixes can be used to access elements in the repository in an efficient way.

Figure 4 illustrates the prefix, postfix encoding. Each node is identified by (i) its document identifier, (ii) the prefix position of the node in the tree, (iii) the posfix position of the node in the tree and (iv) the level of the node in the tree. Looking at Figure 4, we know that *Pauline* is a descendant of *title* because (i) it belongs to the same document, (ii) its prefix is larger and (iii) its postfix smaller. Furthermore, since (iv) their difference of levels is 1, *Pauline* is also a child of *title*.

[3]Special thanks go to Anne-Marie Vercoustre who proposed this encoding.

[4]Special thanks go to Divesh Srivastava who suggested that we use this encoding.

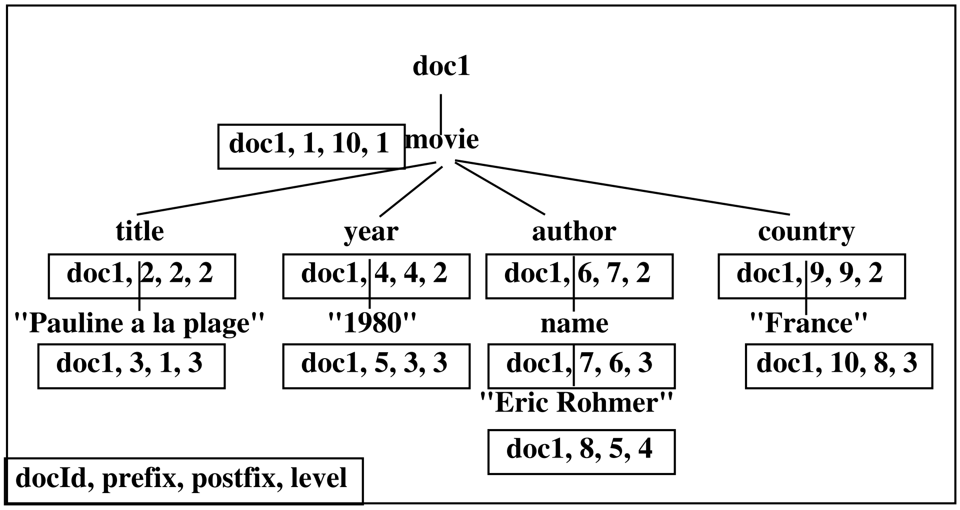

Figure 4: Identifying Nodes

4.2 Ordering Occurences

As stated before, in a XyIndex, occurences are ordered first by document identifier then by code. A code consists of several numbers: prefix, postfix, level. Accordingly, we can consider several ways to order codes: e.g., level then prefix, prefix, etc. We chose to order codes by prefix because, as we will see in the next section, this order has interesting properties that the others do not.

5 Implementing Joins

As explained in Section 2, a $PatternScan$ operation corresponds to a sequence of joins between sets of words occurences. In the next section, we address the problem of finding the best ordering for this sequence. Here, we are concerned with join algorithms.

In Xyleme, we are expecting many concurrent queries and cannot consider using join algorithms involving intermediary data structures. Also, it is important that we return first results fast. These constraints leave mainly two algorithms: nested-loop and merge (i.e., sort-merge when the inputs are already sorted). Obvioulsy, we would prefer the merge implementation since it is more efficient than the nested loop. However, this requires ordered inputs which is sometimes difficult to provide. In this section, we study merge-joins in the context of Xyleme. We first briefly recall the merge algorithm and how it uses the order provided by its input collections. Then, we study (i) the relationship between Xyleme predicates and the order maintained by XyIndexes, (ii) the number of re-reads needed by joins relatively to their predicate and (iii) the joins output order. Finally, we sketch the merge algorithms that we implemented in Xyleme.

5.1 Simple Merge Algorithm

The merge-join algorithm is performed between two ordered lists that we call outer and inner lists. Let us first consider a simple case where the join predicate is =. The lists are ordered in an ascendant way. The algorithm starts with an initialization phase where both lists are prepared to be scanned and the first *outer* and *inner* elements are read (we call *outer*, resp. *inner*, the last element read from the outer, resp. inner, list. The read instruction returns an EOF when there is no element left.). Then, a sequence of steps follows, each step producing the resulting tuples corresponding to one outer value. The sequence ends when a step returns an END status. Each step performs the following actions:

1. while **not**(EOF(*inner*)) and *outer* $>$ *inner*
  `// Find first matching inner`
    read *inner*
2. if EOF(*inner*) **return** the END status
3. if *outer* $<$ *inner*
  `// No matching inner`
    read *outer*
    if EOF(*outer*)
        return the END status
    else
        return the OK status
4. memorize *inner* position in $firstInnerMatch$
  `// At least one matching inner`
5. while **not**(EOF(*inner*)) **and** *outer* $=$ *inner*
  `// Produce results`
    add [*outer*, *inner*] to the result
    read *inner*
6. memorize *outer* in *lastOuter*
7. read *outer*
8. if EOF
    return the END status
  elseif *outer* $=$ *lastOuter*
  `// Same matching inners, position inner to re-read`
    inner $= firstInnerMatch$
9. **return** the OK status

In the above join algorithm, the predicate is in adequation with the input order relation. In Xyleme, although predicates are certainly related to the order maintained by XyIndexes, matters are a bit more complex. This is why we need to understand clearly how the order property of the join inputs is used by the merge algorithm. This is explained below (see Figure 5):

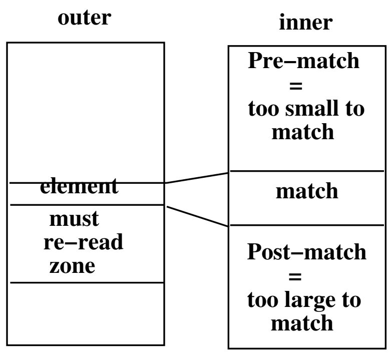

Figure 5: Merge Algorithm

- Given an *outer* element, the order relation allows to view the *inner* collection as three consecutive partitions relatively to the predicate. We call these collections pre-match$_{pred}$ (as in $outer > inner$), match$_{pred}$ (as in $outer = inner$) and post-match$_{pred}$ (as in $outer < inner$).

- The order relation is also used to understand when it is necessary to re-read elements from the inner collection. For instance, in the algorithm above, we re-read the inner elements that matched a given outer element iff it is followed by an equal element.

5.2 On XyIndex Order and Match Relationship:

The basic collections on which we perform joins when evaluating a *Pattern-Scan* are the lists of occurences maintained by XyIndexes. Remember that each occurence consists of a document identifier and a code representing prefix, postfix and level. Also recall that the lists of occurences are ordered first by document identifier, then by prefix.

In Xyleme, a join can use one four predicates: (i) $isParentOf$, (ii) $isAscendantOf$ and their counterparts (iii) $isChildOf$ and (iv) $isDescendantOf$.

First, note that whatever the predicate, if we consider only document identifiers (i.e., the first part of the key), n1 is a pre-match (resp. post-match) of n2 iff n1 $<$ n2 (resp. n1 $>$ n2). Thus, assuming that there are not two elements in each joined collections that come from the same document, we have a simple merge algorithm with no re-reads. So, let us forget about documents identifiers and concentrate on the second part of the keys. In other words, we now assume that we join two collections of nodes (a and b nodes) that belong to the same document. Note that, in many cases, these collections are small. Thus, we could consider a merge-like algorithm to find out words that occur within the same document, and use a nested-loop to join occurences within one single document. However, there is a non negligible number of killer cases: mostly, all queries involving large structured documents (e.g., catalogues). To be convinced, consider the following query:

Example 5.1
select p
from p **in** catalogue/product
where p/name **contains** "name100"

We ran it on a catalogue containing 100,000 products, a hundred of which were named "name100". We used two algorithms to evaluate the joins: a merge and a nested-loop. The merge sequence goes down from *product* to *name100* (as we will see, the best way to go), the nested loop goes the other way round (the best for this algorithm). In this experiment, the merge algorithm was approximatively 30 times faster than the nested-loop.

So, obviously it is worthwhile to consider merge to join sets of occurences belonging to the same document. Remember that the standard merge algorithm relies on (i) a partitioning of the inner sorted collection according to the join predicate and (ii) the ability to reduce the number of re-reads of inner elements. Let us first study the partitioning problem. Let us assume

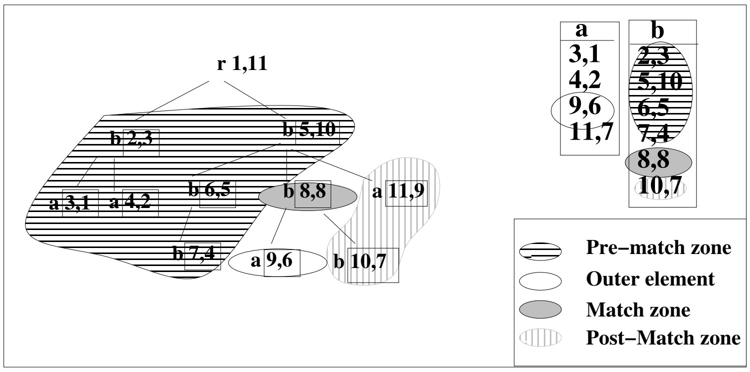

Figure 6: A Good Partitioning: isChildOf

that we are joining two collections of a and b occurences belonging to the same document according to $a\ isChildOf\ b$ (the outer collection is always on the left). Figure 6 illustrates, both on tree and collection, the partitioning of the b *inner* elements when considering (9,6) as the outer a node. As we can see, the partitioning is perfect.

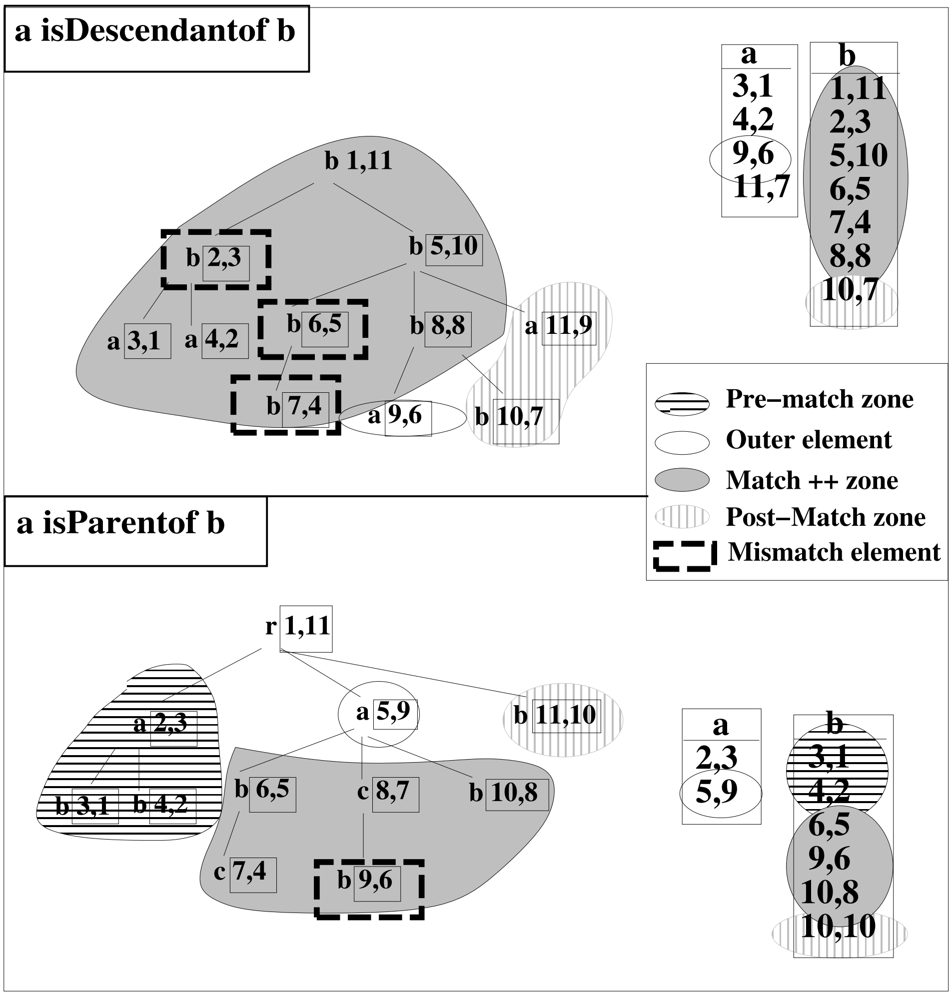

Figure 7: Two Bad Partitionings: IsDescendantOf and isParentOf

This is not always the case as is illustrated by Figure 7 that shows the a $isDescendantOf$ b (top part) and a $isParentOf$ b predicates (lower part). First, note that the match zone of $isParentOf$ (which is too large) corresponds to that of the $isAscendantOf$ predicate. But, more important, remark that pre-match and match zones are mixed with the $isDescendantOf$ predicate. I.e., each step re-reads the inner elements from the start. Obviously, we should be careful when ordering joins not to generate $isDescendantOf$ joins. So, in the sequel, we forget about the $isDescendantOf$ predicate.

5.3 Minimizing Re-Reads:

In the merge algorithm given above, re-reads occur only within a matching step, i.e., a step in which we find *inner*(s) matching an *outer*. In general, this is correct only when order relation and join predicate satisfy the following property:

$$outer1 < outer2$$
$$\Rightarrow$$
$$\forall\ inner1 \in \text{pre-match}_{pred}(outer1),\ \forall\ inner2 \in \text{match}_{pred}(outer2)\ (inner1 < inner2)$$

This property is true for $isParentOf$ and $isAscendantOf$ but not for $isChildOf$. E.g., in Figure 6, $a[9,6] < a[11,9]$ but $b[5,10]$ is a pre-match for the former and a match for the latter. Thus, when joining according to $isChildOf$, at each step, we must re-read the inner relation from the start. As we will see in the sequel, documents featuring recursive elements (e.g., a *subpart* descendant of another *subpart* element) cause problems. It is thus important to note that, although there is a recursion on b elements, it is not the cause of the problem here. To be convinced, replace all b's except $b[5,10]$ by arbitrary tags and the property still does not hold. So, it seems appropriate to avoid $isChildOf$ joins when re-ordering the sequence of joins.

In order to test this property, we ran the following query:

Example 5.2
select n
from n **in** catalogue/product/name

where *catalogue* was a recursive document containing 10.000 products, a product being composed of either name and price or name, price and product, with at most three level of recursion. The plan using $isParentOf$ predicates was 2×10^4 faster than the plan using $isChildOf$ predicates.

Thus, we have only two predicates left: $isParentOf$ and $isAscendantOf$. Let us now consider the number of re-reads they require. In the algorithm given above, re-reads occur only when two consecutive elements are equal. In general, even if two consecutive elements *outer*1 and *outer*2 are not equal, re-reads are necessary if $\text{match}_{pred}(outer2) \subseteq \text{match}_{pred}(outer1)$. Alternatively, when two consecutive elements are not equal, we can avoid re-readings iff:

$$outer1 < outer2$$
$$\Rightarrow$$
$$\text{match}_{pred}(outer2) \cap \text{match}_{pred}(outer1) = \{\}$$

If there is some recursivity in the *outer* elements, $isParentOf$ and $isAscendantOf$ do not have this property. Otherwise, they do. Figure 8

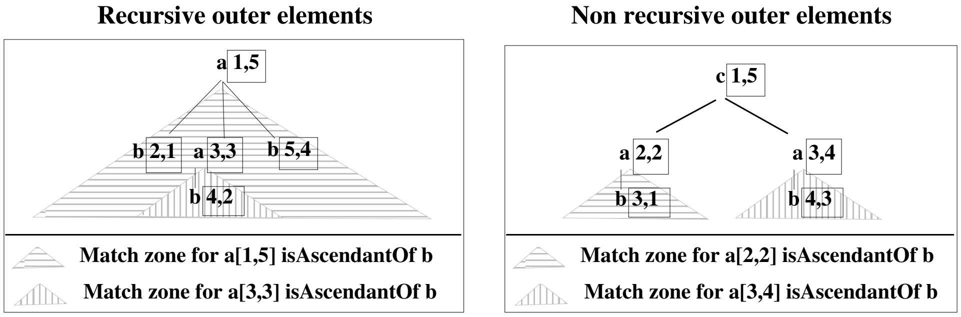

Figure 8: Match Zones for Consecutive Outer Elements

illustrates this by showing the the match zones of two consecutive a nodes when there is recursion in a's (on the left) and when there is not (on the right). Remember that $isParentOf$ and $isAscendantOf$ have the same match zones.

Proving the property in the non recursive case is rather trivial. The match zone is composed of the descendants of an *outer* node. If two outers cannot be descendant one of the other (no recursion), their match zone cannot intersect. More simply, if two consecutive elements do not share an ascendant/descendant relationship, we know that there is no need to re-read the match zone. Note that we rarely find recursive elements within XML documents (remember that our encoding stops at links). Thus, by using merge joins with $isParentOf$ and $isAscendantOf$ predicates, in most cases we will simply scan once the sets of words occurences.

5.4 Ordered Results

In general, if no re-reads are performed, a merge-join returns a set ordered by both *inner* and *outer* keys. However, if re-reads are necessary, the set is ordered first by *outer* then by *inner*, i.e., *inner* by themselves are not ordered. Furthermore, duplicates may be generated by a projection on the inner elements. Figure 9 illustrates this situation. Note that joins using the $isParentOf$ predicate cannot generate duplicate inners since a node has only one parent. Also note that duplicates are the cause for the lack of order of the *inner* elements when joining according to $isAscendantOf$. Indeed, with this predicate, each re-read only produces duplicate *inner*s. (Remember that (i) a match zone with $isAscendantOf$ is perfect, and (ii) that it is either included in that of the previous *outer* element or does not intersect with it - see Figure 8).

As explained before, re-read occurs only when outer elements are recursive. Thus, recursive elements are the cause of broken order. To avoid or

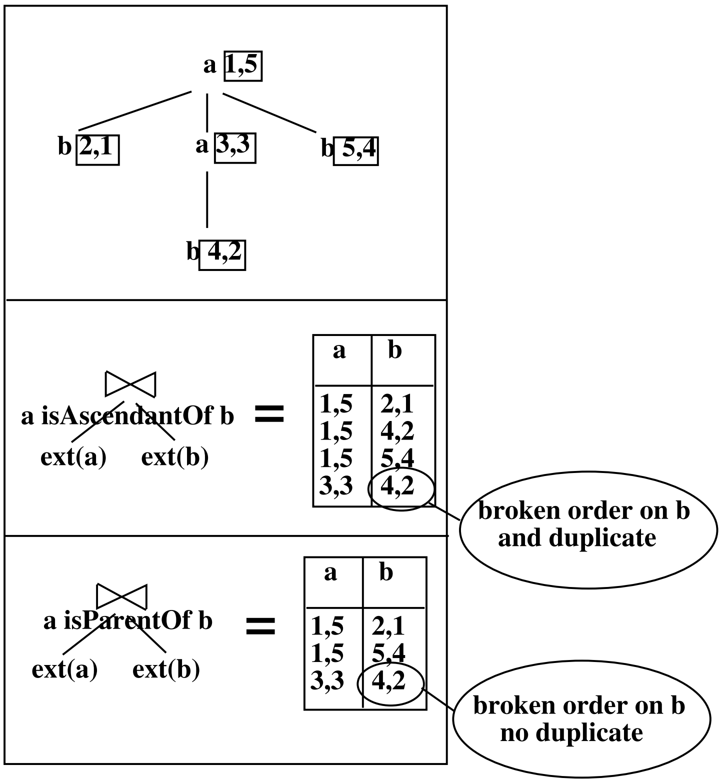

Figure 9: On Merge-Joins, Order and Duplicates

palliate this problem, we have three possibilities:

- We can order the sequence of joins so that each operation uses the previous join outer key (which is always ordered). As we will see in the next section, this is not a good idea since it leads to computed inner collections whose re-reads are expensive.

- We can forget about recursive elements by picking only the highest element in the recursion (e.g., the highest component in a chain of components). This is a reasonnable choice with the $isAscendantOf$ predicate but, with $isParentOf$, we may miss interesting non recursive elements (e.g., $b[4,2]$ in the lower part of Figure 9).

- We can adapt the merge algorithm to deal with un-ordered elements. For instance, if we consider a join between a (slightly) un-ordered outer collection and an ordered inner one, we can do as follows: when an *outer* is smaller than its predecessor (the order is broken), go back to the root of the *inner* collection. Otherwise, do as in a standard merge algorithm. This simple solution is the one we chose. In most case, we have an efficient algorithm with no re-reads. In the rare cases where we encounter recursion within a query, we simply go back a few steps (bounded by the maximum number of occurences of a word within a document). Note that we could improve it by keeping some additional information to go back exactly where we should in the inner collection. However, the algorithm gets complex and since recursive elements are rare, we prefer the simple solution.

So far, we have considered order mainly as a means to use merge algorithms. Another important use of order is to eliminate duplicates. Indeed, each join returns a set. However, the sequence of joins corresponding to a $PatternScan$ is followed by a projection that may generate duplicates that we have to eliminate. Unless we can rely on some order, duplicate elimination requires an intermediary data structure and is generally expensive.

Still, as explained above, results of joins are indeed ordered. E.g., consider a sequence of joins between, in that order, $a_1,\ a_2, \dots a_n$ elements. The result is ordered by a_1, then a_2, ..., then a_n. Thus, if we project on $a_1,\ a_2, \dots a_i$ $(0 < i < n)$, we can eliminate duplicates by simply scanning the collection. As we will see in Section 6, we can organize the joins in an efficient way and still guarantee this property.

5.5 Join Algorithms in Xyleme

So, as explained above, we avoid expensibe re-reads by considering only two predicates: $isParentOf$ and $isAscendantOf$. In the next section, we will come back on this show that it is indeed a reasonnable choice. For now, we see how the algorithm given in the previous section should be modified to fit Xyleme needs.

Joins in Xyleme exist in two flavours: standard join and left semi-join. Below is the standard merge join step. It considers collections of nodes belonging to several documents. It relies on a perfectly ordered inner collection but accepts an outer collection with some disorder among the elements belonging of one document (although, the more ordered is the collection, the more efficient is the algorithm). Finally, $pred$ is either $isParentOf$ or $isAscendantOf$.

`// Find match or post-match zone`

1\. while **not**(EOF($inner$)) **and**

  ($outer.docId > inner.docId$ **or**

  ($outer.docId = inner.docId$ **and**

  $outer.prefix > inner.prefix$))

    read $inner$

2\. if EOF($inner$) **return** the END status

3\. if **not** ($outer$ $isAscendantOf$ $inner$)

`// No match zone`

```
        read outer
        if EOF(outer)
                return the END status
        else
                return the OK status
4. memorize inner position in firstInnerMatch
5. while not(EOF(inner))
              and outer isAscendantOf inner
          // Produce results
        if outer pred inner
                add [outer, inner] to the result
        read inner
6. memorize outer in lastOuter
7. read outer
8. if EOF(outer) return the END status
9. if lastouter.prefix < outer.prefix
   // Broken order between elements
   // belonging to the same document
   // Reread the document elements from the start
        inner = first inner element within the current document
        return the OK status
    elseif lastouter isAscendantOf outer
   // Included match zone, re-read match zone
        inner = firstInnerMatch
        return the OK status
    else
        return the END status
```

The left semi-join ($\ltimes$) produces only one result per outer element with matching inners, instead of one per matching inner. It is used typically, when a join is followed by a projection that removes the inner elements. More precisely, given two sets of tuples, ls and rs, whose attributes are respectively $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_m$:

$$\Pi_{a_1,a_2,\ldots,a_n}(ls \rhd\!\lhd_{pred}\ rs)$$
$$\equiv$$
$$\Pi_{a_1,a_2,\ldots,a_n}(ls \ltimes_{pred}\ rs)$$

The left semi-join algorithm differs from the join's only by Instruction 5:

```
5. while (not(EOF(inner)) and
              outer isAscendantOf inner and
              not(outer pred inner)     \\ Always false when pred = isAscendantOf
                    read inner
```

if **not**($EOF(inner)$) **and** *outer pred inner*
add [*outer*, *inner*] to the result

6 Ordering Joins

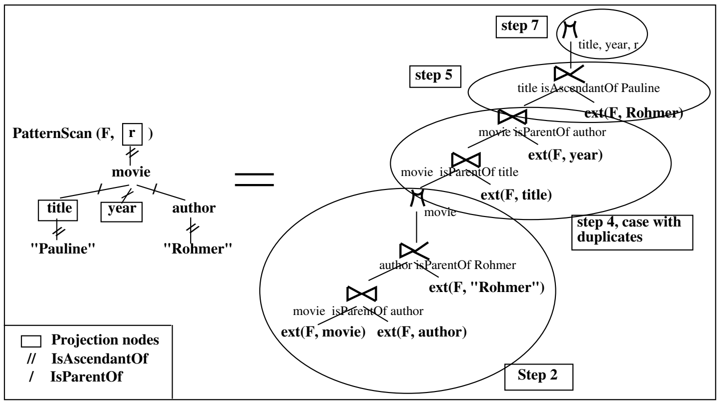

Figure 10: Constructing a Pattern Scan sequence of Joins

In this section, we consider the best way to order the sequence of joins corresponding to a *patternScan* operation. 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.

Xyleme stores large amount of highly heterogeneous and dynamic data. In that context, statistics are simply impossible to maintain. As a matter of fact, we only know the size of each word/tag extent within XyIndexes. Given that, we have to rely on heuristics to order joins. Our join ordering is based on the following assumption.

Ideally, the patternScan operation should be implemented by a sequence of merge joins, joins whose output order can be exploited to evaluate the final projection operation. Furthermore, the joins should not perform re-reads on computed collections.

This seems fairly reasonnable. Indeed, as shown in the previous section, merge algorithms on fully sorted collections beat by far nested loop algorithms. This is also true for projections on sorted collections as compared to projections on unordered collections (notwithstanding the fact that we cannot afford the intermediary data structure required by the latter case).

Finally, since we do not store intermediary results, re-reads on computed collections involves re-computing previous results which should obviously be avoided.

We ran some experiments with the following query:

Example 6.1

select p1,p2
from p1 in catalogue/product
p2 in catalogue//product
wher p2 **contains** $name_x$

where *catalogue* was the recursive catalogue considered in 5.2. $name_x$ was either "name1" or "name10", and *catalogue* contains 1 (resp. 10) product with a name equal to "name1" (resp. "name10"). We considered two sequence of joins, (i) a left deep tree and (ii) a bushy tree, the inner branch of the latter corresponding to variable $p2$. The measures we obtained showed that (i) was three times faster than (ii) with $name_x$="name1" and six times faster with $name_x$="name10".

Combined with other constraints (such as no join involving $isChildOf$ or $isDescendantOf$ predicates), this goal leaves few possibilities, which is good. Indeed, the choices that reduced our field of investigation, were justified and validated by experiments. Furthermore, note that, in any case, the worst complexity with merge joins is rather good: $n \times m \times (1 + r)$ where n is the number of nodes in the pattern tree, m is the average number of occurences per word and r is the average re-read factor (near to 1 since we have few recursive elements).

The followings explain the main characteristics of the ordering algorithm that we give next:

- In the definition given in Section 2, the first joined collection is the set of all trees within the forest. Since nodes are identified, among other things, by their document identifiers, this join is useless. So, we remove it. Still, when a document node has several branches in a pattern tree, we must connect them. For this, we join nodes according to document equality.

- We construct a left-deep join sequence because it is the only way to avoid re-reads on computed collections. Remember that these were very bad.

- We only have two predicates: $isParentOf$ and $isAscendantOf$ (remember that we made this choice to minimize re-reads). This means that, in a left-deep join sequence, a node of the pattern tree always comes after at least one of its ascendants. In other words, this means that the join tree is constructed by a traversal of the pattern tree starting from the nodes just below the document node (that we call *first nodes*) and going down.

- Since the final join order must be adapted to the projection operation, the "projection" nodes must come first in the join sequence (remember that a join preserves only the outer collection order). Given the above point and the fact that the ascendants of a projection nodes are projection nodes, this is not really a problem.

We construct the algebraic expression corresponding to a pattern scan by traversing the pattern tree several times. Let us first consider a pattern tree with only one first node. The algorithm works in four steps which are illustrated by Figure 10:

1. If the first node is connected to the document node by an $isParentOf$ edge, we start the translation by a selection to filter out its occurences whose levels are larger than 0.

2. We traverse the tree in a left deep manner and generate the joins corresponding to the projection nodes using standard joins to connect them. Remember that this nodes are always above the others and that they must come first in the sequence if we want to avoid costly duplicate elimination.

3. We traverse the tree in a left deep way and we add a join with the appropriate predicate for each node that has not been included in the sequence. We use left semi-join for leaf nodes, standard join for the others. Note that we add nodes in a somehow arbitrary order (left deep). So far, we have not found out a better way to go.

4. We add the projection operation. Note that we can remove duplicates by simply checking equality between the new projected tuple and the previous one.

If a pattern tree has several first nodes, we work in a recursive way as follows. We perform the above steps for one branch. Then, for each remaining branch:

- We perform Step 1.

- We join the previous branch result with the current one according to document equality.

- We perform the remaining steps, adding the projected elements of the previous branch to the current branch projection.

Acknowledgments:

This research has benefited from many working sessions with people from the Xyleme team [26]. Notably, we are particularely grateful to Bruno Koechlin to whom we owe the implementation of the XyIndexes. Also, we stole some good ideas about indexing to Thomas Schwentick and Divesh Srivastava who visited us at INRIA. We thank them all deeply.

References

[1] Extensible markup language (XML) 1.0. Technical Report REC-xml-19980210, W3C, February 1998.

[2] S. Abiteboul, S. Cluet, V. Christophides, T.Milo, G. Moerkotte, and J. Simeon. Querying documents in object databases. *International Journal on Digital Libraries*, 1, 1997.

[3] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the web: from relations to semistructured data and XML*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999.

[4] Serge Abiteboul, Haim Kaplan, and Tova Milo. Compact labeling schemes for ancestor queries. To appear in the proceedings of the Twelfth Symposium on Discrete Algorithms (SODA01).

[5] 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.

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

[7] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. 7th Int. World Wide Web Conf.*, 14–18 April 1998.

[8] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 505–516, Montreal, 1996.

[9] R. G. Cattell, editor. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[10] Vassilis Christophides, Serge Abiteboul, Sophie Cluet, and Michel Scholl. From structured documents to novel query facilities. In *Proc. ACM SIGMOD Symp. on the Management of Data*, Minneapolis, Minnesota, 1994. ACM.

[11] 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.

[12] Vassilis Christophides, Sophie Cluet, and Jérôme Siméon. On wrapping query languages and efficient XML integration. In *Proceedings of the*

ACM SIGMOD Conference on Management of Data, Dallas, Texas, May 2000.

[13] James Clark and Steve DeRose. XML Path Language (XPATH), version 1.0, W3C Recommendation. Technical report, W3C, 1999.

[14] S. Cluet, P. Veltri, and D. Vodislav. Views in a large scale xml repository. Submitted to the SIGMOD 2001 Conference.

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

[16] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(11–16):1155–1169, May 1999.

[17] 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.

[18] Daniela Florescu, Ioana Manolescu, and Donald Kossmann. Integrating keyword search into xml query processing. In *Proceedings of the 9th World Wide Web Conference*, Amsterdam, May 2000.

[19] Amélie Marian, Serge Abiteboul, and Laurent Mignet. Change-centric management of versions in an XML Warehouse. In *Bases de Données Avancées*, October, 24-27 2000.
Available from `http://www.xyleme.com/publications.html`.

[20] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *ACM SIGMOD Record*, 26(3):54–66, 1997.

[21] Jason McHugh and Jennifer Widom. Query optimization for XML. Edinburgh, Scotland, August 1999. to appear.

[22] Benjamin NGuyen, Serge Abiteboul, Laurent Mignet, and Gregory Cobena. Query subscription in an xml webhouse. To appear in the proceedings of the First DELOS workshop on Information Seeking, Searching and Querying in Digital Libraries, December 2000.

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

[24] Alexandre Termier, Michèle Sebag, and Marie-Christine Rousset. Combining statistics and semantics for word and document clustering. Technical report, Xyleme, 2000.
Available from `http://www.xyleme.com/publications.html`.

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

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