Xyleme Query Architecture

Vincent Aguiléra, Sophie Cluet, Fanny Wattez
INRIA, projet VERSO
BP 105, 78153 Le Chesnay Cedex
France
name.surname@inria.fr

ABSTRACT

We present the implementation of a special-purpose query operator, namely **PatternScan**, to support efficient evaluation of queries on top of Xyleme, a Web-scale XML warehouse.

Keywords

XML, query processing, indexation, structured documents

1. INTRODUCTION

The coming of XML may radically change the face of the Web. We strongly believe that XML will progressively succeed to HTML as the preferred publication format. One key feature of XML is its ability to support high level structured queries. The objective of the Xyleme project 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 provides a complete query language that mixes database with full text query capabilities. In this paper, we briefly sketch the architecture of the system and then explain how queries are evaluated in Xyleme.

2. XYLEME

Our main concern when designing the system was scalability. To achieve that, we distribute data and meta-data over a cluster of Linux PCs. We distinguish between three kinds of machines. (1) Repository machines store XML documents that are clustered together into semantic domains. (2) Index machines have large memories. Each are used to index part of one repository machine. The partitioning insures that Xyleme indexes reside in main memory. (3) Interface machines are connected to the Internet. They are in charge of running Xyleme applications and of dispatching tasks/processes to the other machines. They maintain some meta-information about the repositories and indexes, so as to be able to compute and distribute query execution plans.

3. QUERIES IN XYLEME

The evaluation of structured queries over loosely structured data such as XML has been extensively studied during the last ten years [1]. However, query evaluation in Xyleme is guided by requirements that are not those of traditional database systems. We are considering here billions of documents (Google recently indexed more than one billion HTML documents) and (hopefully!) millions of queries per day. This impose some severe constraints, notably maintaining as few as possible additionnal data structures per query (this prevent for instance the use of sort or hash-based algorithm) and returning first answers fast.

3.1 Xyleme Query Language

Xyleme query language is an extension of OQL [3] and provides a mix of database and information retrieval characteristics. It is consistent with the requirements published by the W3C XML Query Working Group. Since each semantic domain potentially contains many heterogeneous DTDs, queries are usually formulated against some view. Views are generated semi-automatically and consist of mappings from paths in some schema to paths in real documents. Then, they are translated into a union of so-called concrete queries, i.e., queries against real documents. We simplify matters here and will consider two domains, *tourism* and *sport*. The following concrete query will be used as a running example. It returns the names and hotels of places where one can practice tennis. Its result is an XML document with as many *result* elements as answers, each having two sub-elements, *name* and *hotel*. A document *d1* in *tourism* may provide results if it contains an element *d/tour/stop/townDescription* that references a document *d2* in *sport*, such that a subelement *sports* at any depth in *d2* contains the word *tennis*.

EXAMPLE 3.1.

```
select  s/name, s/hotel
from    doc1 in tourism,
        doc2 in sport,
        s in doc1/tour/stop
where   s/townDescription contains url(doc2)
        and doc2//sports contains ”tennis”
```

3.2 The PatternScan Operator

Although we rely mainly on database optimization techniques, notably the use of an algebra, we have adapted them by introducing, as in [2], a special-purpose query operator called **PatternScan**. We will see in the next section how this operator can be efficiently evaluated using a full text index. Figure 1 shows the straightforward algebraic translation of query 3.1. Above every operator we see the form of its returned tuples. Evaluating this plan as such would not be a good idea. Notably, it would involve to retrieve from the repository every pair of documents in *tourism* and *sport* and then reject probably most of them because they do not satisfy the **Selection** operation predicate. We now see how we can do better using query rewriting techniques. Figure 2 shows the algebraic plan computed by the optimizer for the example query 3.1. The **PatternScan** operator is the key operator of Xyleme algebra. It filters a forest of XML documents according to some pattern and returns a set of tuples, one per tree

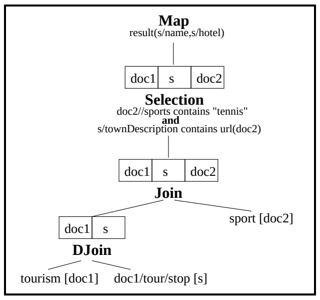


Figure 1: A straightforward algebraic plan

structure in the forest that matches the pattern. The pattern trees contain two kinds of edges (corresponding to the / and // operators). It also features three kinds of nodes representing respectively (i) document roots (e.g., **doc1**); (ii) elements or attributes (e.g., **tour**), some of which are framed because they are part of the query result (e.g., **stop**); and (iii) keywords (e.g., *tennis*). Assuming that

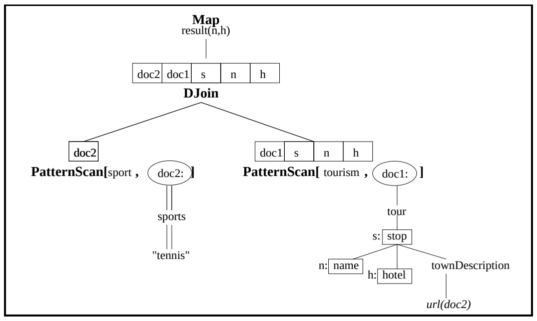


Figure 2: Algebraic rewriting using PatternScan

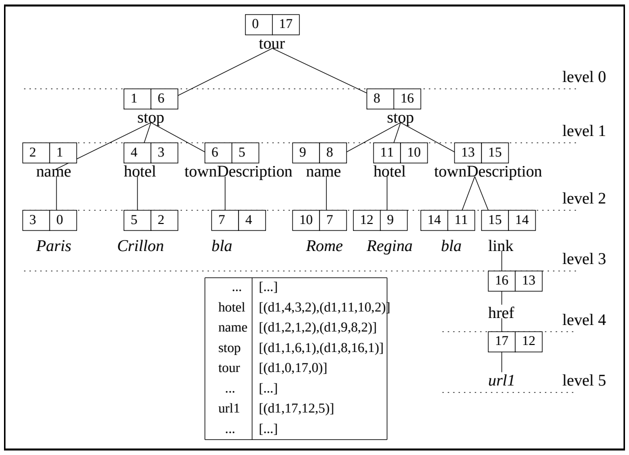
PatternScan can be evaluated efficiently, this plan is far better than the previous one since we need to acces the repository only for selected tuples.

4. INDEXING STRUCTURED DOCUMENTS

In Xyleme, pattern filtering is applied to huge collections of heterogeneous documents. Given the fact that it is the most frequent and, in practice, the most expensive operation that has to be evaluated, it is crucial to evaluate it fast. We believe that this can only be achieved by using indexes residing in main memory. We rely on the fact that, notwithstanding links or intra-document references, XML documents can be modeled as trees. The idea is to encode simply the structural position of each word, element or attribute in a document. We then use a full text index in the following way: to each word occurrence in the index is associated (i) its kind (element, attribute or simple word), (ii) the document in which it appears, and (iii) its position in this document.

4.1 Words encoding

The idea of the encoding presented here is to capture the interval defining each internal node of a document. Then, the descendant relationship can be deduced by interval inclusion. Figure 3 shows, in the upper part, a small document, called **d1**, annotated with prefix-postfix encoding, computed during 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). In the lower part, we give a partial image of the full text index associated to the



...	[...]
hotel	[(d1,4,3,2),(d1,11,10,2)]
name	[(d1,2,1,2),(d1,9,8,2)]
stop	[(d1,1,6,1),(d1,8,16,1)]
tour	[(d1,0,17,0)]
...	[...]
url1	[(d1,17,12,5)]
...	[...]

Figure 3: Indexing a Document in Xyleme

tourism collection after this document has been entered. We know that the first occurrence of **hotel** is a descendant of **stop** because it belongs to the same document, its prefix is larger and its postfix smaller. Note that, in the full text index, we add to each word occurrence its level in the document tree so as to be able to differentiate a child from a descendant.

4.2 Implementing the PatternScan Operation

Let us now come back to pattern filtering and see how full text indexes are used to evaluate **PatternScan** operations (we call this implementation **FTIScan**). The inputs of an **FTIScan** are the index corresponding to a collection of documents and a pattern tree. The output of the operation is a collection of tuples with one attribute per extracted item of information. Let us take as an example the evaluation of the **PatternScan** on the domain *sport* (left part of figure 2). Let I_{sport} denote the index for this domain, $I_{sport}(w)$ the set of occurences in this index for the word w. Let I_s (resp. I_t) denote $I_{sport}(sports)$ (resp. $I_{sport}(tennis)$). A document d matches the given pattern if there exists a couple (s, t) in $I_s \times I_t$, such that t is a descendant of s and s and t belong to d. Typically, this can be evaluated by a join between I_s and I_t. Assuming that I_s and I_t are sorted, this join requires at most $(Card(I_s)+Card(I_t))$ comparisons. Similarly, a more complex **PatternScan**, as the one on the right part of figure 2 can be seen as a sequence of joins. We are currently implementing a heuristic that allow to evaluate such join sequences with a worst case complexity of $N \times M$ where N is the number of nodes in a pattern tree p and M the average number of occurences of words in p.

5. REFERENCES

[1] 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.

[2] 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.

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