The Xyleme Project

S. Abiteboul, INRIA and Xyleme
S. Cluet, Xyleme
G. Ferran, Xyleme
M-C. Rousset, LRI, U. Paris 11

November 2001

Abstract

The current development of the Web and the generalization of XML technology [31] provides a major opportunity to radically change the management of distributed data. We have developed a prototype of a *dynamic warehouse for XML data*, namely Xyleme. In the present paper, we briefly present some motivations and important aspects of the work performed in the framework of the Xyleme project. (A short preliminary version of this report appeared in [35].) The project was completed at the end of 2000 with the implementation of a prototype. The prototype was then turned into a product by a start-up company also called Xyleme [34].

Xyleme: a complex tissue in the vascular system of higher plants that consists of vessels... functions chiefly in conduction of water and dissolved minerals but also in support and food storage...[Merriam-Webster]

1 Introduction

The Web is huge and keeps growing at a healthy pace with already billions of public pages. Most of the data is unstructured, consisting of text (HTML, pdf, doc, etc.), images, sound, videos. Some is structured, mostly coming from relational databases. All this data constitutes the largest body of information accessible to any individual in the history of humanity. The lack of structure and the distribution of data seriously limit access to the information. To provide functionalities beyond full-text searching ala Alta Vista or Google, specific applications are being built that require heavy investments.

The backbone of the web is made in HTML, an hypertext language. In an HTML document, content (data and structure) and presentation are mixed. It is thus very difficult to extract information from HTML pages and, consequently, to develop applications using web data. A major evolution is occurring that will dramatically simplify the task of developing such applications: the *eXtensible Markup Language*, *XML*. XML is a standard to exchange semistructured data [1]. An XML document only contains data and structure. Presentation is given elsewhere in a style-sheet. The extraction of data from XML pages is therefore much easier and can be performed automatically. Very hard problems in the context of HTML, such as integrating data from the web into a company's information system, become much simpler with XML.

The number of public XML pages is sill marginal, about 1% of the number of HTML pages. This is mostly because a large portion of web users are not yet equiped with XML browsers. On the other hand, communities (scientific, business, others) have defined or are defining standard XML formats for data in their particular areas. Indeed, XML is adopted very rapidly by the industry and is spreading very fast across companies intranet.

Because we believed in the XML future, we started in 1999 the Xyleme project. The goal was to study and build a *dynamic Warehouse for massive volume of XML data* obtained from the Web or from private repositories. We designed a warehouse that could scale to billions of XML pages, terabyte of indexes, thousands of simultaneous queries. This lead us to using a native repository, i.e., a repository specially designed to store (XML) trees. The problems we encountered are typical warehousing problems [33] such as query optimization, change control or data integration. These problems acquire a truly distinct flavor in the context of XML and of the web. More precisely, the research directions we studied and that we discuss here are the following:

- *efficient storage* for huge quantities of XML data (billions of documents).
- *query processing* with indexing at the element level for such a heavy load of documents.
- *data acquisition and refresh strategies* to build the repository and keep it up-to-date.
- *change control* with services such as query subscription to monitor changes of interest for this huge quantity of data.
- *semantic data integration* to free users from having to deal with many specific DTDs when expressing queries.

Certain functionalities of Xyleme are standard and are or will soon be found in many systems. For instance, from a query viewpoint, Xquery [32] processors are now available. It will not be long before some search engines start taking advantage of XML structure to provide label-equal-value search and some may soon support parts of Xquery. Other features such as change monitoring are already provided (to some extent) by systems, e.g., eWatch or NetCurrents. So, what is specific to Xyleme? The main distinguishing feature is perhaps that *Xyleme is based on warehousing*. The advantages of such an approach may best be illustrated by a couple of examples. First, consider query processing. Certain queries requiring joins over pages distributed over the Web are simply not conceivable in the current Internet context, for performance reasons. They become feasible with a warehousing approach. To see another example, consider monitoring. It is trivial to notify users when some pages of interest change. Now, if the pages are warehoused, one can support more precise alerts, e.g., detect when the description of a particular product in a catalog changes. More examples of new services supported by Xyleme will be encountered in the paper.

The Xyleme Project functioned as an open, loosely coupled network of researchers. Serge Abiteboul, Sophie Cluet (INRIA researchers) and F. Bancilhon (then CEO of Ariso) initiated the project. Xyleme was rapidly joined by Guy Ferran and by researchers from the Verso database groups from INRIA-Rocquencourt [30], Mannheim U. [22], the CNAM-Paris [11], the IASI Team of University of Paris-Orsay [16]. The project we describe here was completed at the beginning of 2001. A prototype had been implemented. This prototype has been turned into a product by a start-up company also called Xyleme [34]. We describe here the prototype and not the system used in production by Xyleme.

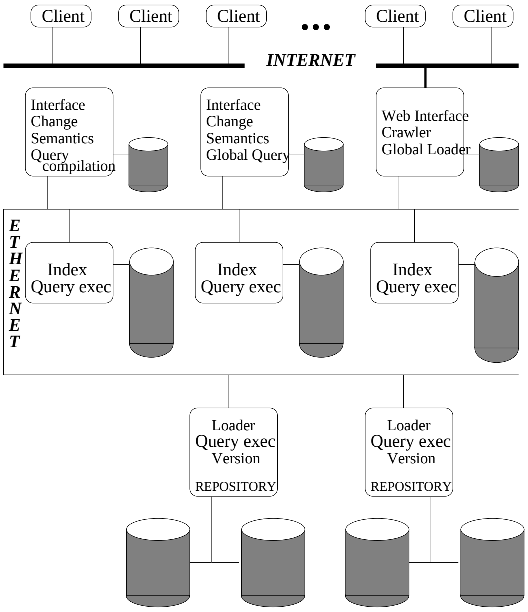

Figure 1: Architecture

The present paper is a survey. We provide references to longer reports describing specific aspects of the research. More references on related works may be found there.

The paper is organized as follows. Section 2 introduces the architecture. The repository is discussed in Section 3. Section 4 deals with query processing and indexing. Data acquisition is considered in Section 5 and change control in Section 6. The topic of Section 7 is data integration.

2 The Architecture

In this section, we briefly present the architecture of the system.

The system is functionally organized in four levels: (i) physical level (the Natix repository tailored for tree-data and an index manager); (ii) logical level (data acquisition and maintenance, loading and query processing); (iii) application level (change management and semantic data integration); (iv) interface level (interface to the web and interface with Xyleme clients). Figure 1 how the various fonctionalities are typically spread over various machines. Figure 2 show the architecture from a purely functional viewpoint.

The system runs on a network of Linux PCs. All modules are written in C++ and communicate using Corba. The Xyleme clients run on standard Web browsers using Java applets. The communication with clients uses HTTP, XML/HTML pages, forms or SOAP methods. These aspects will not be detailed here.

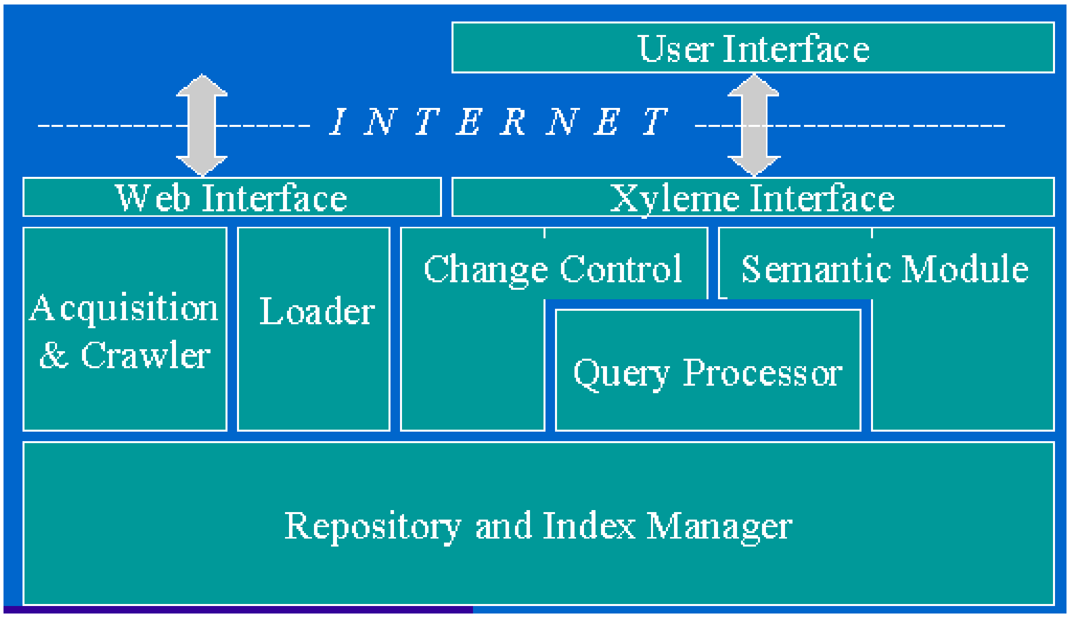

Figure 2: Architecture: Main modules

3 The Repository

In this section, we discuss the repository. More details may be found in [17].

Xyleme requires the use of an efficient, update-able storage of XML data. This is an excellent match for Natix developed at the U. of Mannheim. Typically, two approaches have been used for storing such data: (i) store XML pages as byte streams or (ii) store the data/DOM tree in a conventional DBMS. The first presents the disadvantages that it privileges a certain granularity for data access and requires parsing to obtain the structure. The second artificially creates lots of tuples/objects for even medium-size documents and this together with the effect of poor clustering and the additional layers of a DBMS tend to slow down processing. Natix uses a *hybrid approach*. Data are stored as trees until a certain depth where byte streams are used. Furthermore, some minimum structure is used in the "flat" part to facilitate access with flexible levels of granularity, Also, Natix offers very flexible means of choosing the border between the object and flat worlds.

Natix is built, in a standard manner, on top of a record manager with fixed-size pages and segments (sequences of pages). Standard functionalities for record management (e.g., buffering) or fast access (e.g., indexes) are provided by Natix. One interesting aspect of the system is the splitting of XML trees into pages. An example is shown in Figure 3. Observe the use of "proxy" nodes (p-nodes) and "aggregate" nodes (h-nodes). One can control the behavior of the split by specifying at the DTD level, which splits are not allowed, are forced, or are left to the responsibility of the system.

The performances of Xyleme heavily depend on the efficiency of the repository. By taking advantage of the particularities of the data, Natix does offer appropriate performances.

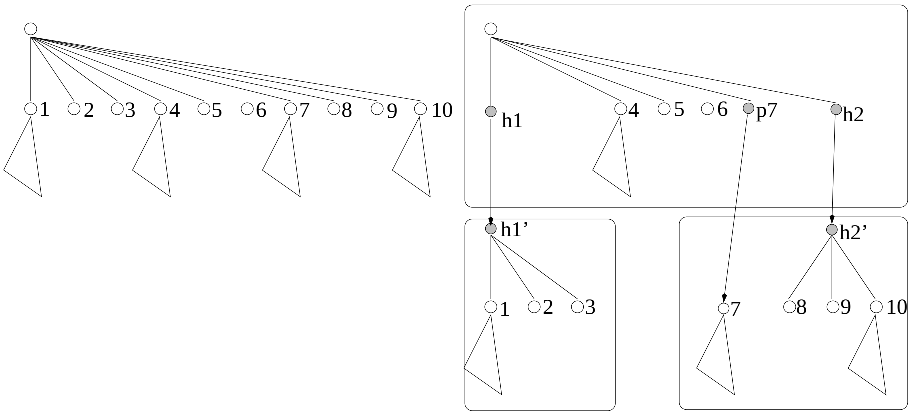

Figure 3: Distributing nodes on records

4 Query Processing

In this section, we consider processing of static queries. Change queries are studied in Section 6. A more detailed presentation of this work can be found in [4, 5].

The evaluation of structured queries over loosely structured data such as XML has been extensively studied during the last ten years. The techniques differ slightly depending on the system that is used to store the data, object-oriented, dedicated or relational. However, none of the existing 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.

Our approach differ from the state of the art in database literature in mainly two ways: (i) an efficient and scalable implementation of pattern tree matching for XML queries and (ii) the way queries against views are processed.

Efficient Pattern Tree Matching

The main paradigm underlying a query language for XML such as X-Query [32] is that of selecting data by matching tree patterns made of path expressions. The path expressions describe traversals along sub-elements with variables getting bound to some nodes along these paths. Hence, a key operation in query processing over XML is, given a tree pattern, to produce a set of bindings for the pattern variables.

We thus introduce a special pattern tree matching operator in an otherwise rather standard algebra [**?**]. The main difficulty is to give an efficient and scalable implementation to this operator.

As stated before, Xyleme applications are large scale with many users and the need to get first answers fast. Together, these three characteristics reduce the solution scope. Indeed, efficiency and large scale somehow imply in-memory indexes, many users forbid the use of large intermediate structures, and fast first answers means pipeline evaluation. As a consequence, our implementation of the pattern tree matching operator relies on merge joins and an adaptation of full text index technology.

XyIndexes: Some full text indexes (called full inverted indexes) store the offset associated to each word occurrence (or posting) in a document in order to answer queries about words proximity. For instance, queries such as: "name" **near** "person".

We improve this mechanism so as to answer structured queries such as "name" is an element whose value contains "Jean" or "person" is an ancestor element of both "name" and "address". For this, as in [20], we encode words using the Dietz's numbering scheme. More precisely, we associate to each word, its position in the document relatively to its ancestor and descendant nodes. This position is encoded by three numbers that we call pre-order, post-order and level. This is illustrated with a small document in Figure 4.

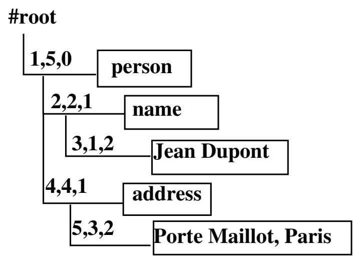

Figure 4: Pre-order, Post-order and Level Numbering

Given an XML tree T, the pre (resp. post) order number of nodes in T are assigned according to a left-deep traversal of T. Every time we enter (resp. exit) a node, we increment the pre (resp. post) counter and assign its value to the current node. The level number is the number of ancestors of the current node. Given two nodes n and m in T, the two following properties stand :

- n is an ancestor of m if and only if $pre(n) < pre(m)$ and $post(m) > post(n)$
- n is the parent of m if and only if n is an ancestor of m and $level(n) = level(m) - 1$

It is generally admitted that the best implementations of full text indexes that have posting lists consisting only of documents identifiers require 10 to 20% of the indexed database size. By simply adding offsets, full inverted indexes generally use up to 60% of the indexed data [**?**][1]. These numbers show what a naive implementation of the (pre-order, post-order, level) numbering can lead to in terms of space consumption.

Our implementation of the index mechanism stems from an observation that we made during first experiments: the difference between pre-order and post-order numbers is usually very small. We rely on this to compress the main memory representation of the index to about 50% of the database size. I.e., while supporting more complex structured queries, our index has a size similar or smaller to that commonly attributed to full inverted indexes.

[1]This ratio can be improved by removing commonly used words (called stopwords).

Pattern Tree Matching: Using the index mechanism described above, a pattern tree matching operation is equivalent to a sequence of joins among posting lists followed by a projection. For instance, consider the following X-Query:

FOR $p IN document("yellowPages")//person
WHERE $p/name **contains** "jean"
RETURN $p/address

Figure 5 shows, on the left, the pattern tree that needs to be matched against the document "yellowPages" and, on the right, one possible sequence of algebraic operations to evaluate it.

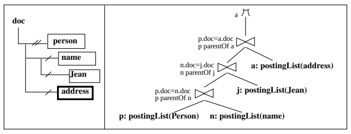

Figure 5: A pattern tree and one possible evaluation plan

The most efficient pipelined evaluation of a join with no intermediate data structure is the merge algorithm that requires ordered inputs.

Within the index, posting lists are ordered first by document identifier, then by prefix number. As explained in [5], we believe this is the best order. However, intermediate joins can break it. We address this problem in two ways. First, we adapt the merge algorithm to deal with partially ordered inputs. Note that, due to this partial order, it will be sometimes necessary, to read several times the same records (a partial nested loop within the merge). Second, we order joins so that (i) re-reads due to partial orderings are minimized, (ii) when re-reads are necessary, intermediate joins are never re-computed (remember that we do not have intermediate data structures) and (iii) the join sequence output order can be exploited to evaluate the final projection operation.

In this way, our implementation of the tree pattern matching operation has a complexity linear in the number of selected data items when the documents do not feature recursive elements. Note that documents rarely feature recursive elements even if DTD's do. In any case, our measures show that, in practice, we maintain a linear complexity even with recursive elements (although it is exponential in theory).

Large Scale views

As will be explained in Section 7, Xyleme partitions documents into clusters of documents, each corresponding to a domain of interest (e.g., *tourism*, *finance*, etc.). Also, Xyleme provides *a view mechanism*, that enables the user to query a single structure summarizing a cluster rather than the many heterogeneous DTDs it contains.

A view in a relational system builds one relation by combining information coming from others. Similarly, a view in Xyleme combines several clusters into a virtual one.

However, whereas the tuples of a relation share a common type, a cluster is a collection of highly heterogeneous documents. Thus, a view cannot be defined by one relatively simple join query between two or three collections but rather by the union of many queries over different sub-clusters (Section 7 explains how views are (semi-)automatically generated by the system). As a matter of fact, there is no *a priori* limit to the size of a view definition in Xyleme since it depends on the number of DTDs (that may be implicit in documents that are only well-formed) that one can find on the Web. Obviously, techniques that are used to maintain and query relational views have to be seriously re-considered to fit Xyleme's needs.

The view information in Xyleme is decomposed into two parts corresponding to the standard query processing steps: compilation and execution. The compilation part of a view is replicated on each upper-level machine (see Figure 1) and is mainly used to understand which index and repository machines are concerned by a query [10]. The size of this information is usually small, depending on the number of domains and the size of the view schema. The execution part of a view is distributed over the index machines (see Figure 1). Each stores only the view information that concerns its indexed cluster (or sub-cluster). This data is used to translate tree pattern matching operations just before they are evaluated by XyIndexes and is order of magnitude smaller that the indexes. The query translation algorithm has a linear complexity.

5 Data Acquisition and Maintenance

In this section, we consider the issues of data acquisition and maintenance. More details may be found in [24].

The warehouse acquires information from private repositories and from the web. We focus here on the management of data from the web. For such data, we are not informed when a new document is put on the web or when a document changes. Such information may only be acquired by visiting the web. Typically, automated web agents are used to visit the web, retrieving pages to perform some processing such as indexing, archiving, site checking, see, e.g., [13]. The robots use links in the retrieved pages to discover new URLs (i.e. new pages). Xyleme crawlers perform a similar task. We load HTML and XML pages to discover new links and to monitor these pages. Our goal was also to build an XML warehouse. So, we store the XML documents we find on the web. (We also store XML documents from private repositories.) We then need to refresh (public or private) pages we already know of to keep the repository up to date. Each crawler can read millions of pages per day. Several crawlers are used simultaneously to share the acquisition and maintenance work.

Since network, storage and processing resources are limited, we have to be satisfied with possibly visiting only part of the web and with maintaining the retrieved data up to date with some acceptable level of staleness. We see next how this optimization task is performed.

Classical search engines typically discover pages using a breadth first strategy. They perform refresh in a cyclic manner. As a result, each page is rarely refreshed and some interesting pages may be ignored. In contrast, to decide which page to read or to refresh next, Xyleme uses a sophisticate optimization strategy taking into account user interest, the importance of pages and their change frequency. This is in the spirit of previous works of Cho, Haveliwala and Garcia-Molina.

Acquisition and maintenance are first guided by user specification. If a user requests that a page be visited daily, then the system does so. The decision to read a page is also guided by the *importance* of pages. All pages on the web do not have the same importance. For example, Le Louvre homepage is more important that an unknown person's homepage. Page importance has first been used by search engines to display results in the order of page importance [14]. It turns out to be a most valuable tool for guiding the refreshing and discovery of pages. Intuitively, important pages should be refreshed more often[2] and when crawling for new pages, pages estimated to be important are fetched first [9].

As mentioned above, page discovery and maintenance are guided by page importance. Maintenance is also guided by the *change frequency* of pages, e.g., it is not necessary to read regularly a page that never changes.

Based on user interest, estimated page change frequency and page importance, acquisition and maintenance become optimization problems. More precisely, given specific resources and most notably the bandwidth of the crawler and the storage space, we have to maximize the quality of the repository, i.e., the relevance of the stored pages and their freshness. A Xyleme administrator specifies some general policies, e.g., whether it is more urgent to explore new branches of the web or to refresh the documents we already know of. This is achieved by assigning portions of the bandwidth to the various tasks. A page scheduler then cycles around all pages it knows of (already read or not) deciding for each page whether this page has to be refreshed/read during this particular cycle. The decision for each page is based on the *minimization of a global cost function under some constraint.* The constraint is the average number of pages that Xyleme is willing to read per time period. The cost function is defined as the dissatisfaction of users being presented stale data.

One of the most complex aspects in this framework is the computation of page importance, To conclude this section, we describe page importance more precisely. We then discuss how it is computed by Xyleme.

Page importance

Following some previous works of [18], Page and Brin proposed a notion of page importance based on the link structure of the web [7]. Intuitively, a page is important if there are many important pages pointing to it. More precisely, this can be defined as follows.

Consider the graph of the web represented as a matrix $L[1..n, 1..n]$ where $1..n$ are the pages of the web. If there is an edge for i to j, $L[i,j] = 1$, otherwise it is 0. Let $out[1..n]$ be the vector of out-degrees. We assume that the graph is strongly connected, i.e., for each i, j, there is a directed path from i to j. In practice, the web is not strongly connected and techniques are used to force this property without affecting much the result. We also assume that the graph is aperiodic [25]. The importance Imp of nodes as defined by [7] is the solution of the equation:

$$Imp[j] = \sum_i \left(\frac{L[i,j]}{out[i]} * Imp[i]\right)$$

[2]Google seems to use such a policy for refreshing pages.

It has been shown that, because the graph is strongly connected and aperiodic, there is a single solution with:

$$|Imp| = \sum_i Imp[i] = 1$$

and with only positive coordinates. In the following, Imp will denote this particular vector. This definition is mathematically founded: one can show that the importance of a page is the probability to be on that page after an infinite random walk on the web. This is the definition of importance used by search engines such as Google [14]. It works well in practice as shown by the success of such search engines.

Page importance computation

Typically, Imp is computed as the fixpoint of Imp^k where

$$Imp^{k+1}[j] = \sum_i \left(\frac{L[i,j]}{out[i]} * Imp^k[i]\right)$$

starting from some initialization vector. This fixpoint may be computed off-line by an iterative algorithm requiring the storage of the link matrix and intensive access to it on disk. The main issue in this context is the size of the web, billions of pages [6, 19]. Techniques have been developed to compute page importance efficiently, e.g., [15]. The web is crawled and the link matrix computed and stored. A version of the matrix is then frozen and one process computes page importance which may take hours or days for a very large graph. So, the core of the technology for the off-line algorithms is fast sparse matrix multiplication (in particular by extensive use of parallelism). This is a classical area; see e.g., [29].

We designed a new, on-line, adaptive algorithm for computing page importance [2]. Our algorithm does not require the storage and/or access to the link matrix. The algorithm computes the importance of pages on-line while crawling the web. Intuitively speaking, some "cash" is initially distributed to each page and each page when it is crawled distributes its current cash equally to all pages it points to. This fact is recorded in the history of the page. The importance of a page is then obtained from the "credit history" of the page. The intuition is that the flow of cash through a page is proportional to its importance. It is essential to note that the importance we compute does not assume anything about the selection of pages to visit. If a page "waits" for a while before being visited, it accumulates cash and has more to distribute at the next visit.

Thus our algorithm, named BAO (for basic, adaptive, on-line), requires less storage resources than standard algorithms. It also continuously refines its estimate of page importance and is tuned to adapt to changes of page importance on the web. (With the off-line algorithm, one needs to regularly perform computations of page importance.)

We implemented and compared the standard off-line algorithm and our on-line algorithm. We tested BAO with a real crawl of the web that lasted several months. We discovered close to one billion URLs; only 300 million of them were actually read, so we computed Page Importance over 300 million pages. Note that because of the way we select pages to read, the 300 million pages are in some sense the most important pages of the web. Among them, the most important ones were read very regularly (some dayly) whereas the least important ones were read very occasionally. This experiment was sufficient (with limited human checking of the results) to validate the algorithm in a production environment. We also performed some extensive tests with synthetic graphs. An advantage of our algorithm is that it rapidly detects the introduction of new important pages, so they can be rapidly crawled. More details may be found in [2].

6 Change Control

In this section, we discuss the query subscription system of Xyleme. More details may be found in [27]. Due to space limitations, other works concerning change control such as a versioning mechanism based on deltas [23] and a diff for XML [12] will be ignored here.

Web users are often concerned by changes in pages they are interested in. In Xyleme, we monitor the flow of documents that are discovered and refreshed by the system. We can detect that a page changed at the time we are fetching the page. For HTML pages, we have their signature and we can only detect whether they have changed or not. For some XML pages, we have the old version of the page and we can also monitor changes at the element level, e.g., a new product has been introduced in a catalog.

We designed a subscription language that allows users to specify monitoring queries. The language also supports *continuous queries* that request asking repeatedly the same query to monitor changes in this answer, see e.g., [21, 8]. We will ignore this aspect here. A major issue in designing the subscription system was *scalability*. Indeed, most of the problems we consider here would be relatively easy at the level of an intranet. This is not the case when confronted with the sheer size and complexity of the web. Our monitoring system is designed (and has been tested) to monitor the fetching of millions of XML and HTML documents per day, while supporting millions of subscriptions.

We next describe (by example) the subscription language. We briefly discuss the architecture of the system. We then consider the two main components, the alerters and the monitoring query processor.

Monitoring

In this section, we describe the functionality of the monitoring system and introduce the query subscription language.

Each Xyleme crawler reads a stream of web pages. We can abstractly view the crawlers as producing a stream of documents: $\mathcal{D} = \{d_i \mid i\}$, i.e., the list of pages fetched by the crawlers in the order they are fetched. The main task that is set before the change monitoring system is to find, for each document, whether there is a monitoring query interested in that document or in changes of the document. If this is the case, the monitoring system produces a *notification* that consists of the code of the monitoring query and some relevant information about the particular detection. Thus, each monitoring query can be viewed as a filtering over the list $\mathcal{D}$ of documents. It produces a stream of notifications. When a particular condition holds, the current value of this stream of notifications is used to produce a (subscription) report using a report query. The report is sent using some protocole such as SMTP (mail), HTTP/POST (web posting) or HTTP/SOAP (web service).

There is a last component in the specification of a subscription, namely *refresh statements*. A refresh statement gives the means to a user to explicitly specify that some page or a set of pages have to be read regularly, i.e., the means to influence the crawling strategy.

To illustrate, an example of a simple subscription using our subscription language is as follows:

```
subscription MyXyleme

monitoring
  select <UpdatedPage url=URL/>
  where URL extends ``http://inria.fr/Xy/''
        and modified self

monitoring
  select X
  from self//Member X
  where  URL = ``http://inria.fr/Xy/members.xml''
         and new X

report
  % an XML query Q' on the ouput stream
  ...
  when notifications.count > 100
```

This subscription includes two monitoring queries. The core of such a query ressembles an SQL query. However, the *from* clause is missing from some monitoring queries since the filtering applies, by definition, to the document that is being processed. This document is denoted *self* in the query. The keyword URL is a shorthand for URL(self).

In the above subscription, the first monitoring query requests to monitor the set of URLs with a certain prefix that are found to have been modified (since the last fetch). The second one requests to monitor new elements of tag *Member* in a particular XML document. All the resulting notifications are (logically) bufferized until the report condition becomes true, here, until more than 100 notifications are gathered. This is specified in the *when* clause. One can also request to be notified periodically, e.g., weekly. Once the reporting condition holds, a reporting query over all the notificatioons gathered so far is issued. The reporting query (omitted here) may, for instance, remove duplicates URL's of pages that have been found updated several times. The report is also an XML document.

The atomic conditions may bear on the content of the document, e.g., that it contains some words. They may bear on some metadata about the document, i.e., its type (e.g., HTML), its URL (e.g., its URL matches some pattern), its DTD in the case of XML. Furthermore, they may bear on some change information such as the last date of update of the document or the last access date. In case of XML, the conditions may be on elements of the current document.

The query subscription language is rather powerful, see [27] for details.

The subscription system

In this section, we discuss a simplified architecture of the subscription system.

Each monitoring query can be viewed as a conjunction of atomic events. For each document, the *Alerters* detect the set of atomic events raised by the document. If this set is nonempty, an alert is sent to the Monitoring Query Processor. An alert consists of the set of atomic events detected together with the requested data. The role of the *Monitoring Query Processor* is to determine whether the particular alert actually covers some monitoring query. If this is the case, a notification is sent by the Monitoring Query Processor to the Reporter. When the reporting condition holds for a subscription (it is time to issue a report), the Reporter fetches the set of notifications received so far, an XML document and post-processes it by applying the report query to it. This produces the report (also an XML document).

Some key modules and their roles are considered next.

The Subscription Manager receives the user requests and controls the other modules of the subscription system. Its main task is to handle insertions, deletions or modifications of subscriptions and store them in a MySQL database. The Reporter stores the notifications it receives in a MySQL database, produces the reports and issue them.

The main problem we had to solve was to detect that a document matches a subscription with the heavy rate of arrival of documents in Xyleme and with a huge number of subscriptions to process. To conclude this section, we consider the most critical components of the system performance-wise, the Alerters and the Monitoring Query Processor.

Alerters The whole monitoring is based first upon the detection of *atomic events*. The role of the Alerters is to detect these events for each document entering the system, and if that is the case, to send a global alert for this particular document. To avoid unnecessary network traffic, Alerters have to be set as close as possible to the modules they monitor. For instance, the URL Alerter is placed next to the module gathering metadata about documents, and the XML Alerter next to the loader of XML documents. An *Alerter* plugged into a specific module first of all interacts with its host to get the information it needs to detect the alerts. Our approach is based on a *document flow*. An Alerter sends the atomic events it detected on a document to the Alerter of the next module that will process the document for further investigation onto the same data. The last Alerter in the chain sends its alerts and their associated information to the *Monitoring Query Processor* for the following processing phase.

We implemented three kinds of alerters, namely, the URL, XML and HTML Alerters. The most complex one is the *XML Alerter*. For instance, this alerter can detect alerts such as some element tagged *product* contains the word *electronic* somewhere in its subtree, or that the word *electronic* occurs in a PCDATA directly beneath a *product* element. The XML alerters use sophisticate hashing techniques, for instance, to detect such patterns of interest. The XML Alerter can also detect changes at the element level, such as the insertion of an element with a given pattern. To do that, it obtains the last version from the store and computes the *delta* between the new document and its previous version (if available) [12]. Atomic events are then detected on the delta.

Monitoring Query Processor For each document d_i for which some alerts were raised, the Monitoring Query Processor receives the set of atomic events and has to decide whether this set of events contains all the atomic events required by a particular monitoring query.

Let $\mathcal{A}$ denote the set of all possible *atomic events*. A *complex event* C is a finite subset of $\mathcal{A}$. Let $\mathcal{C} = \{C_j \mid 1 \leq j \leq M\}$ be the finite set of complex events that is being monitored. (M is the number of complex events.) Let $A = \cup\{C_j\}$ represent the finite set of atomic events of interest. Let $A_i \subseteq A$ represent the set of atomic events detected for document d_i. The role of the Monitoring Query Processor is to determine for each i, the set $\{j \mid C_j \subseteq A_i\}$. We use some ordering on the atomic events, i.e., $A = \{a_1, ..., a_N\}$. Thus, each A_i and C_j are considered as ordered subsets of A.

The data structure we use is a *hash tree* [3], i.e., intuitively, a tree where all nodes are hash tables. The complexity analysis of the algorithm [26] shows that (within some range), the algorithm runs in time $O(l^2)$ where l is the number of atomic events raised by a document. The time complexity does not depend on the other parameters and in particular, does not depend on the number of monitoring queries. This has been validated experimentally. Our measures show that the algorithm can process several thousand documents per second on a standard PC even in the presence of millions of subscriptions. As a consequence, one *Monitoring Query Processor* can support the load of about tens of crawlers. The Monitoring Query Processor was designed so that its task could be distributed on several machines to scale to support more documents in input and more query subscriptions.

The subscription system has been implemented and is now used by customers of the Xyleme company.

7 Semantic Data Integration

In this section, we address the issue of providing some form of semantic data integration in Xyleme. More details may be found in [28].

Queries are precise in Xyleme 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, 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*, that enables the user to query a single structure, that summarizes a domain of interest, instead of many heterogeneous schemata.

Views can be defined by some database administrator. However, this would be a tedious and never ending task. Also, metadata languages such as RDF may be used by the designer of the DTD or by domain experts to provide extra knowledge of a particular DTD. We believe that this will become common in the long range. However, for now, the field is too young, standards are missing and such knowledge is not available. Thus, we studied how to automatically extract it using natural language and machine learning techniques.

The first task is to classify DTDs into domains (e.g., *tourism*, *finance*) based on a statistical analysis of the similarities between the words found in different DTDs and/or documents. Similarity is defined based on ontologies or thesauri such as Wordnet.

Once an *abstract DTD* has been defined to structure a particular domain, the next task is to generate the semantic connections between elements in the abstract DTD and elements in concrete ones. Each element is identified by a path from the root of its DTD (e.g., *hotel/address/city*). The problem is then to map paths to paths. Obviously, all tags along a path (i) are not always words and (ii) do not have the same importance. Concerning (i), we use standard natural language techniques (e.g., to recognize abbreviations or composite words) and a thesaurus. As for (ii), we mainly rely on human interaction. Notably, the abstract DTD designer can indicate to the mappings generator that some tag is not particularly meaningful in a path, or, on the contrary essential. For instance, if we consider *hotel/address/city*, we can imagine that *hotel* is meaningful but *address* not that much. In this fashion, we will be able to map *auberge/town* to *hotel/address/city*, but not *company/address/city*.

Conclusion Working in the Xyleme Project was a fascinating experience. Many problems were encountered that still require to be further investigated. The research and development are of course continuing in the various research groups and in the Xyleme start-up.

Acknowledgments: The Xyleme project involved many people and we want to thank them all for making it possible. We want to thank in particular the following people who worked on the project: V. Aguilera, S. Ailleret, B. Amann, F. Arambarri, G. Cobena, G. Corona, A. Galland, M. Hascoet, C-C. Kanne, B. Koechlin, D. Le Niniven, A. Marian, L. Mignet, G. Moerkotte, B. Nguyen, M. Preda, M. Sebag, J-P. Sirot, P. Veltri, D. Vodislav, F. Watez and T. Westmann. We also want to thank researchers who discussed with us some aspects of the technology, and in particular, G. Jomier and F. Lliarbat. Finally, we want to thank INRIA's management, and in particular, J.-P. Banatre and L. Kott, for their support.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web.* Morgan Kaufmann, California, 2000.

[2] S. Abiteboul, M. Preda, and G. Cobena. Adaptive on-line page importance computation. INRIA, Verso Internal Report, www-rocq.inria.fr/verso, 2001.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Int. Conf. Very Large Data Bases*, pages 487–499. Morgan Kaufmann, 1994.

[4] V. Aguilera, S. Cluet, P. Veltri, and F. Watez. Querying XML documents in xyleme. *ACM SIGIR Workshop on XML and information retrieval*, 2000.

[5] Vincent Aguiléra, Frédéric Boiscuvier, Sophie Cluet, and Bruno Koechlin. Pattern tree matching for xml queries, 2001.
Available from `http://www-rocq.inria.fr/verso/PUBLI`.

[6] K. Bharat and A. Broder. Estimating the relative size and overlap of public web search engines. In *7th International World Wide Web Conference*, 1998.

[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine, www7 conference, computer. In *Networks 30(1-7)*, 1998.

[8] J. Chen, D. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continous query system for the internet databases. *ACM SIGMOD*, page 379, 2000.

[9] J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through url ordering. In *In Proceedings of 7th World Wide Web Conference*, 1998.

[10] S. Cluet, P. Veltri, and D. Vodislav. Views in a large scale xml repository. In *VLDB'01*, 2001.

[11] Database Group, Vertigo, CNAM-Paris. http://sikkim.cnam.fr/.

[12] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in xml documents. In *Data Engineering*, 2002.

[13] Search engine watch. www.searchenginewatch.com/.

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

[15] T. Haveliwala. Efficient computation of pagerank. Stanford Database Group Technical Report, 1999.

[16] IASI group, Orsay Univ. http://www.lri.fr/LRI/iasi/introduction.en.html.

[17] C.-C. Kanne and G. Moerkotte. Efficient storage of XML data. Technical Report 8/99, University of Mannheim, 1999. `http://pi3.informatik.uni-mannheim.de/`.

[18] J. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46:5:604–632, 1999.

[19] S. Lawrence and L. Giles. Accessibility and distribution of information on the web. *Nature*, 400, 1999.

[20] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. 2001.

[21] L. Liu, C. Pu, W. Tang, and W. Han. Conquer: A continual query system for update monitoring in the www. *International Journal of Computer Systems, Science and Engineering*, 2000.

[22] Pi_3 database Group, Mannheim Univ. http://pi3.informatik.uni-mannheim.de/mitarbeiter.html.

[23] A. Marian, S. Abiteboul, G. Cobéna, and L. Mignet. Change-centric management of versions in an XML warehouse, 2001. Proc. VLDB.

[24] L. Mignet, M. Preda, S. Abiteboul, B. Amann, and A. Marian. Acquisition and maintenance of xml data from the web. Verso Internal Report, 2001.

[25] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, 1995.

[26] B. Nguyen and S. Abiteboul. A hash-tree based algorithm for subset detection: analysis and experiments. INRIA, Verso Internal Report, 2001.

[27] B. Nguyen, S. Abiteboul, G. Cobena, and Mihai Preda. Monitoring XML data on the web. In *ACM Sigmod*, 2001.

[28] C. Reynaud, J.P. Sirot, and D. Vodislav. Semantic integration of xml heterogeneous data sources. In *International Database Engineering and Applications Symposium, IDEAS*, 2001.

[29] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of research and development*, 41(6), 1997.

[30] Verso group, INRIA. http://www-rocq.inria.fr/verso/.

[31] World Wide Web Consortium pages. http://www.w3.org/.

[32] W3C. XQuery. http://www.w3.org/XML/Query.

[33] J. Widom. Research problems in data warehousing. *International Conference on Information and Knowledge Management (CIKM)*, 1995.

[34] Xyleme. http://www.xyleme.com/.

[35] L. Xyleme. A dynamic warehouse for the XML data of the web. IEEE Data Engineering Bulletin 24(2): 40-47 (2001).