Monitoring XML Data on the Web

Benjamin Nguyen Serge Abiteboul Gregory Cobena Mihai Preda

ABSTRACT

We consider the monitoring of a flow of incoming documents. More precisely, we present here the monitoring used in a very large warehouse built from XML documents found on the web. The flow of documents consists in XML pages (that are warehoused) and HTML pages (that are not). Our contributions are the following:

- a subscription language which specifies the monitoring of pages when fetched, the periodical evaluation of continuous queries and the production of XML reports.
- the description of the architecture of the system we implemented that makes it possible to monitor a flow of millions of pages per day with millions of subscriptions on a single PC, and scales up by using more machines.
- a new algorithm for processing alerts that can be used in a wider context.

We support monitoring at the page level (e.g., discovery of a new page within a certain semantic domain) as well as at the element level (e.g., insertion of a new electronic product in a catalog).

This work is part of the Xyleme system. Xyleme is developed on a cluster of PCs under Linux with Corba communications. The part of the system described in this paper has been implemented. We also mention first experiments.

Keywords

XML, HTML, Change Control, Monitoring, Continuous Query, Warehouse, Web, Subscription.

Strange fascination, fascinating me
Changes are taking the pace I'm going through.
Turn and face the strange
Ch..ch..changes.

David Bowie, Changes

1. INTRODUCTION

The web constitutes the largest body of information accessible to any individual throughout the history of humanity and keeps growing at a healthy pace [4]. Most of it is unstructured, and heterogeneous. A major evolution is occurring that will dramatically simplify the task of developing applications with this data, the coming of XML [1, 28, 27, 18]. XML is still in its infancy and the number of XML pages found on the web is modest (although growing very fast). What was once only a huge collection of documents is turning into a massive database. In this paper, we study the following problem : monitoring for a very large dynamic warehouse built from XML documents found on the web.

Web users are often concerned by changes in pages they are interested in. The present work has been developed in the context of Xyleme [31], a dynamic XML warehouse for the web. We monitor the flow of documents that are discovered and refreshed by the system. The monitoring of a large database is a typical warehousing problems [30]. However, a basic distinction is that since we do not control the (external) web sites, we have to detect changes at the time we are fetching the pages. For HTML pages, we have their signature and we can only detect whether they have changed or not. For XML, we also monitor changes at the element level, e.g., a new product has been introduced in a catalog.

A first contribution of the paper is a subscription language based on two somewhat complementary ways of observing changes, namely:

1. Change monitoring that consists in *filtering* the flow of documents acquired by the system to detect changes that may interest certain users based on specifications of these users' subscriptions.
2. Continuous queries that consist in *regularly asking* the same query of interest to the user over the entire warehouse.

The subscription language we propose is tailored to XML and HTML and offers the possibility to specify complex monitoring and continuous queries with interactions between them. It also provides means of specifying the nature and time of delivery of subscription reports.

Another contribution of the paper is the presentation of the general architecture of our subscription system. A major issue in this context is *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. The Xyleme system, and in particular, the subscription system we describe here, have been implemented and tested.

The core of the monitoring system consists in (what we call) the *monitoring query processor* that receives the alerts detected for each document and tests whether they correspond to one or more subscriptions. Towards this end, we propose a new algorithm. We would like to stress that this part of the system can be used in a much larger setting. In general terms, each alert consists of a set of atomic events and the problem can be stated as finding in a flow of sets of atomic events, the sets that satisfy a conjunction of properties. Our algorithm was designed to support a flow of millions of alerts per day and millions of such conjunctions. Preliminary measures show that indeed it does so.

Related works A lot of works in databases involving time is about versioning data [7, 6, 17]. Although we do have a versioning mechanism in the Xyleme system, this is not the topic of the present paper. Temporal query languages have also been extensively studied [25]. In particular, a temporal query language on XML-like data is described in [6]. Our subscription language can also be viewed as a temporal query language, although a particular one because of its focus on changes.

Some of the techniques we use originate from active databases [29]. A main distinction is that we are not directly aware of the changes of data on the web. This modifies somewhat the issues and introduces real challenges in terms of fast reaction to changes [12].

Continuous query system have recently triggered a lot of interest. For instance, The Conquer system [16, 9, 15] can run SQL-like queries on a given HTML document. Like ours, their system provides email notification. However, building a complex SQL-like query with Conquer requires having a solid knowledge of the web page that is being monitored. The factorization of database triggers is studied in [13]. The optimization of XML continuous queries is studied in [8, 22]. These works are to some extent complementary to ours and we intend to introduce in our system optimizations in their style.

It should be pointed that change monitoring is becoming very popular on the web. For instance, NetMind [20] offers change monitoring on HTML web pages that you submit, and notifies you via email. Some search engines (e.g., Northern Light [23]) also provide such functionalities. Compared to such systems, our main advantage is that because of XML, we can support more elaborate a subscription language with monitoring at the element level. In absence of information, we cannot really compare our performance to theirs on more basic subscriptions.

Organization The paper is organized as follows. In Section 2, we present a quick review of the subscription system's functionalities. Then we give in Section 3 a general overview of the architecture. In Section 4, we discuss the Monitoring Query Processor that forms the core of the system, and in particular the algorithm it is based on. Then, we describe the parts that are specific to the application, the subscription language in Section 5 and the alerters in Section 6. The last section is a conclusion.

2. MOTIVATIONS

In this section, we explain the general motivations of our system, spiced up by examples.

2.1 The Xyleme system, in brief

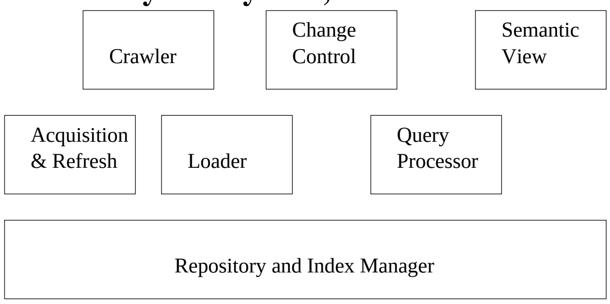

Figure 1: Xyleme Functional Architecture

A complete description of the Xyleme system is beyond the scope of the present paper. We only give here a brief and partial description of the main functionalities of the system. The main modules are shown in Figure 1. The lowest layer consists of the XML *repository* and *index manager*. The repository, called Natix, being developed at Mannheim University, is tailored for storing tree-data [21], e.g., XML pages. Above the repository, on the left hand side, are the three modules in charge of populating and updating the XML warehouse. Since the functionalities of both *crawler* and *loader* should be obvious, let us just say a few words about the *data acquisition and refresh* module [17, 19]. Its task is to decide when to (re)read an XML or HTML document. This decision is based on criteria such as the importance of a document, its estimated change rate or subscriptions involving this particular document. On the right-hand side, we find the *query processor* [2] that is an XML-tailored query processor that supports our own XML query language in the absence of a standard for the moment. Above it are the *change control* and *semantic* modules. The main role of the *semantic* module is to classify all the XML resources into semantic domains and provide an integrated view of each domain based on a single *abstract* DTD for this domain. The *change control* module is the topic of the present paper, and will be detailed further.

Xyleme runs on a cluster of Linux PCs [24]. All modules and in particular the XML loaders and the indexers are distributed between several machines. The repository itself is distributed. Data distribution is based on an automatic semantic classification of all DTDs. The system tries to cluster as many documents as possible from the same domain on a single machine. The entire system is written in C++ and uses Corba [10] for internal communication and HTTP for external ones.

2.2 Monitoring

The subscription language will be detailed in Section 5. The role of this section is to give a flavor of the possibilities of the system and thereby motivate the work.

Web users are not only interested in the current values of documents. They are often interested in their evolution. They want to be notified of certain changes to the web. They also want to see changes presented as information that can be consulted or queried. There are two main components in the control of changes in such a system:

- Monitoring queries: Changes to a page are discovered when the system reads the page. The available information consists of the new page, some meta-data such as its type or the last date of update and eventually the signature of the old page or the old page itself (if the page is warehoused).
- Continuous queries: Changes may also be discovered by regularly asking the same query and discovering changes in the answer. In that sense, the *versioning* of query answers (not detailed here) is an important aspect of a change control system.

A subscription consists of a certain number of monitoring queries and continuous queries. Let us consider more precisely monitoring first. The Xyleme system reads a stream of web pages. We can abstractly view this stream as an infinite list of documents: $\mathcal{D} = \{d_i \mid i\}$, i.e., the list of pages fetched by Xyleme in the order they are fetched. The main task that is set before the change monitoring system is to find, for each document entering the system, if there is a monitoring query interested in this 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 document. Thus, each monitoring query can be viewed as a filtering over the list $\mathcal{D}$ of documents. It produces a stream of notifications.

The role of continuous queries is different. Each continuous query is regularly evaluated. Thus the continuous query processor produces a stream of pairs consisting of the code of a query together with the result of the evaluation of this query. By analogy, we also call such a pair, a notification. Indeed, these two streams are combined to form a single 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.

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 has to be read regularly, i.e., to influence the crawling strategy. We have not yet implemented this part of subscriptions and will not discuss it much in the paper. We mention it here for completeness. In our current implementation, subscriptions influence the refreshing of pages only by adding importance to the pages they explicitly mention [19]. Such pages will be read more often.

Thus, a *subscription* more precisely consists of (see Figure 2):

1. one or more *monitoring queries* over the input stream that filter the stream of documents and produce notifications that feed the unique notification stream.
2. one or more *continuous queries* over the warehouse that also feed the notification stream. Each continuous query comes with a condition to specify when to issue the query, i.e., a frequency (e.g., weekly) or a particular notification.
3. refresh statements that indicate, for instance, that pages for a particular site should be visited at least weekly.
4. an indication, called the *report condition* that specifies when to produce a report and a query, called *report query*, over the notification stream that produces the *subscription report* that is (for instance) emailed to the subscribers.

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

continuous ReferenceXyleme
  % a query Q that computes, e.g., the list of
  % sites that reference Xyleme
  try biweekly

refresh ``http://inria.fr/Xy/members.xml'' weekly

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

This subscription will monitor the set of URLs with a certain prefix that are found to have been modified (since the last fetch) and the new elements of tag *Member* in a particular XML document. It also requests to evaluate query Q (omitted here) biweekly. All the resulting notifications are (logically) bufferized until the report condition becomes true, i.e., until more than 100 notifications are gathered. Once this is the case, a reporting query over all the data gathered so far is issued. The reporting query (also omitted

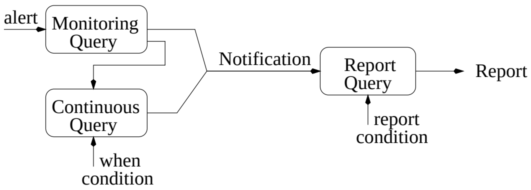

Figure 2: Main Components of a Subscription

here) may, for instance, remove duplicates URL's of pages that have been found updated several times.

For instance, the previous subscription could return:

```
<Report>
<UpdatedPage url=‘‘http://inria.fr/Xy/index.html’’/>
<UpdatedPage url=‘‘http://inria.fr/Xy/members.html’’/>
<Member><name>jouglet</name><fn>jeremie</fn></Member>
<Member><name>nguyen</name><fn>benjamin</fn></Member>
<Member><name>preda</name><fn>mihai</fn></Member>
  ...
<ReferenceXyleme>
<site url=‘‘http://www.yahoo.com’’>
<site url=‘‘http://www.amazone.com’’>
...
</ReferenceXyleme>
</Report>
```

3. THE SUBSCRIPTION SYSTEM

In this section, we present the general architecture of the subscription system shown in Figure 3. This architecture can be broken down into two groups of modules.

- Some generic modules that can be used in a more general setting of change control. These are the Monitoring Query Processor, the Subscription Manager, the Trigger Engine, and the Reporter.
- Some application specific modules that are dedicated to the control of change in the Xyleme environment. These include the specific Alerters we are using, the Xyleme Query Processor, and some modules to input subscriptions (Xyleme Subscription Manager) and send results (Xyleme Reporter).

In the figure, the dotted lines are used for the flow of commands and the filled lines for the dataflow. The generic part of the system is within the thick line rectangle.

The global system For each document, the Alerters detect the set of atomic events of interest. If the set is nonempty, an alert is sent to the Monitoring Query Processor that consists of the set of atomic events detected together with the requested data. The Monitoring Query Processor determines whether some subscriptions are concerned with these alerts or whether they should trigger some particular processing. For instance, the Trigger Engine may start the evaluation of a query. Notifications coming from the Monitoring Query Processor (for monitoring queries) or the Trigger Engine (for continuous queries) are sent to the Reporter. When some condition holds, the Reporter sends the set of notifications received so far, an XML document, to the Xyleme Reporter that post-processes it by applying an XML query to it. This produces a report that is either sent by email, or consulted on the web, with a browser. The Subscription Manager is in charge of controlling the entire process.

The various modules and their roles are considered next.

Alerters (see Section 6) The whole monitoring is based upon the detection of *atomic events*. These depend on the type of documents that are being processed. For instance, for HTML documents, typical atomic events we consider are the matching of the URL of the document against some string pattern or the fact that the document contains a given keyword. For XML documents, we would also consider, for instance, the fact that the DTD of the document is a DTD we are interested in, or that it contains a specific tag, or that a new element with a tag we are monitoring has been inserted in the document.

The role of the Alerters is to detect these events for each document entering the system, and if that is the case, to send an alert for the particular document. Thus we see that these modules are *application dependent*. We distinguish between three kinds of alerters: (i) URL alerters that handle alerts concerning some general information such as the URL of a document or the date of the last update, (ii) XML alerters that are specific to XML documents and (iii) HTML alerters for HTML documents. (Only the first two have been implemented.)

Monitoring Query Processor (see Section 4) The system must detect conjunctions of atomic events that correspond to subscriptions. We call *complex event* such a conjunction of *atomic events*. The role of the Monitoring Query Processor is, based on the alerts raised by a document (i.e. a set of atomic events), the detection of the complex events that this document matches. When such a complex event is detected, the *Monitoring Query Processor* sends a *notification* that consists of the code of the complex event[1] along with some additional data (see the *select* clause further) to the *Reporter* and/or the *Trigger Engine*.

Trigger Engine The *Trigger Engine* can trigger an external action either upon receiving a notification, or at a given date. In our setting, it is in charge of evaluating the continuous queries either when a particular notification is detected or regularly (e.g., biweekly). The query code combined with the result of the query forms a notification that is sent to the Reporter.

[1]In fact all the complex events are detected on a document simultaneously and thus are sent to the *Reporter/Trigger Engine* in one batch.

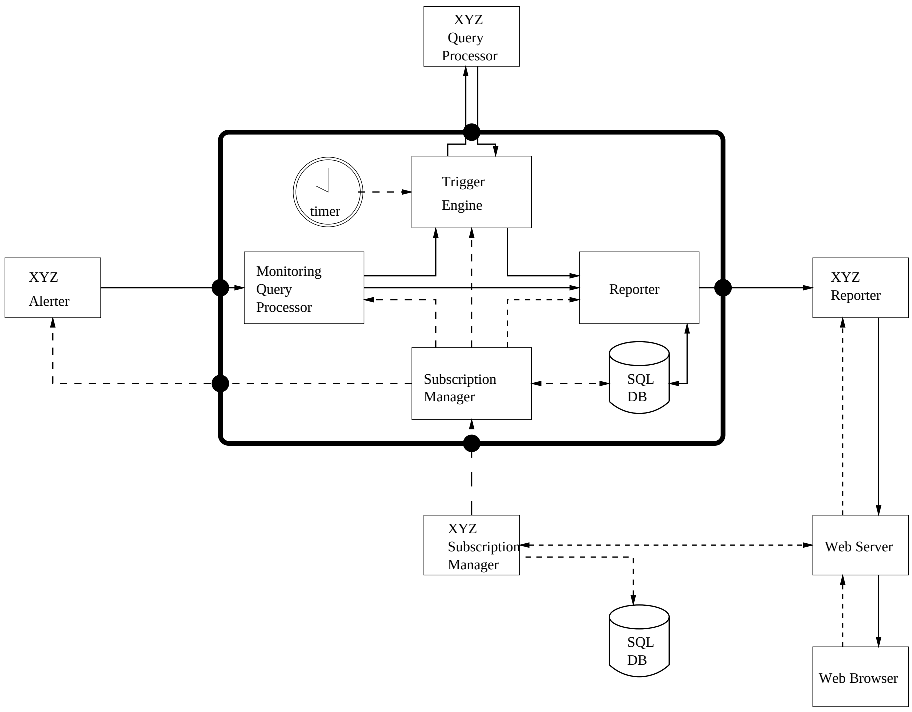

Figure 3: Architecture for the subscription system

The (Xyleme) Reporter The *Reporter* stores the *notifications* it receives. When a report condition is satisfied, it sends these notifications as an XML document. The Xyleme Reporter post-processes this report, basically by applying an XML query to it. Reports are for the moment sent by email. We are considering the support of an access to reports via web publication which seems better for very large reports. The main difficulty for the reporter is the management of a heavy load of emails. In our implementation, the Reporter supports hundreds of thousands of emails per day on a single PC. This limitation is due to the UNIX send-mail daemon implementation.

On a single PC, the subscription system can process over 2.4 million notifications per day when connected to the rest of the Xyleme system and hundreds of thousands of emails. It is designed to be distributed although distribution has not been tested yet.

The (Xyleme) Subscription Manager The *Subscription Manager* receives the user requests and manages the other modules of the subscription system. Indeed, the *Subscription Manager* guides and controls the activity of the other modules.

To construct a subscription, a user posts it to our Apache [3] server with his browser. The form is subsequently processed by the *Subscription Manager*. Its first role is to serve as an interface for the insertion of new subscriptions and the deletion or modification of existing ones. To be more precise, the Subscription Manager is split into a generic module and an Xyleme specific module. The specific one is in particular in charge of parsing the subscriptions. In our implementation, both modules use the same MySQL database [5] for recovery. Information about users such as email addresses is also stored in this database.

The second role of the manager is to control the various components of the subscription system. For instance, it chooses the internal codes of atomic events and (dynamically) warns the *Alerters* of the creation of new events, their codes and semantic. It controls in a similar manner the *Monitoring Query Processor* for managing complex events, the Trigger Engine for continuous queries and the Reporter(s) for reports.

The *Subscription Manager*'s task is not as intensive as that of other modules, since it only depends on the number of people that decide to subscribe to our system at the same time (a few hundred), whereas the *Alerters* depend on the **total** number of people that have subscriptions (millions). The *Subscription Manager* runs on a single machine.

4. MONITORING QUERY PROCESSOR

In this section, we consider the *Monitoring Query Processor*. In particular, we introduce a novel algorithm to perform the detection of subscriptions that are matched by each document. The main difficulty is the heavy rate of arrival of documents and the number of subscriptions we want to process. We now present the problem, our algorithm, then a brief analysis.

4.1 Overview

For each document d_i that is fetched by the system, the alerters detect the set of atomic events that are raised by this document. The role of the Monitoring Query Processor is to decide whether this set of events contains all the events in a complex event associated to a particular monitoring query. In such a case, a notification is triggered.

The main difficulty of the problem is that we need to support a rate of millions of documents per day. So, for instance, we cannot afford one disk I/O access per event detected. Before focusing on the algorithm, we briefly consider without going into details other important aspects of this module:

- Subscriptions keep being added, removed and updated while the system is running. Thus the data structure we use has to be updatable dynamically. Although our system *does* support such updates, we will ignore this issue here.
- Persistency and recovery are handled here by the *Subscription Manager* that pilots the MySQL database.
- The *Monitoring Query Processor* has no semantic knowledge of the data associated to the atomic or complex events it handles. Such additional information is passed in XML format, from the Alerters to the Reporter in a transparent manner.

We now present the algorithm. Let $\{d_i\}$ be the list of documents that is being filtered. Let $\mathcal{A}$ denote the set of all possible *atomic events* where an atomic event in our setting corresponds to an atomic condition in the *where* clause of a monitoring query (see Section 5). A *complex event* C is a finite subset of $\mathcal{A}$. The core notification process can be abstracted as follows:

- Let $\mathcal{C} = \{C_j \mid 1 \leq j \leq M\}$ represent 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 Monitoring Query Processor must determine for each i, the set $\{j \mid C_j \subseteq A_i\}$.

It is convenient to assume some ordering on the atomic events, i.e., $A = \{a_1, ..., a_N\}$. Thus, each A_i and C_j can be viewed as an ordered subset of A.

Typically, we are considering the case where we have about 1 to 10 million atomic events ($N \approx 10^6$) and 1 to 10 million complex events ($M \approx 10^6$). Note that the problem can be stated as a finite state automata problem. For each i, we need to find the words in $\{C_1, ..., C_M\}$ "contained" in the word A_i. In principle, we could detect this using a finite state automaton in linear time (in the cardinality of A_i) and in constant time in the other inputs to the problem. Unfortunately, because of the size of the problem, the number of states of the automaton would be prohibitive.

We next present the algorithm. It should be noted that before selecting this particular algorithm, we considered alternatives. We found out that the choice of one over the other depended on the conditions of use of the system. A critical factor is the number of atomic conditions in a complex event. . Another critical factor is the number of complex events interested in a specific atomic condition. An interesting candidate algorithm we considered turned out to be exponential in that factor. The cardinality of the set of atomic events detected on a document is also important. The algorithm we introduce next presents (as we shall see) a nice behavior with respect to these three aspects as well as others.

4.2 Algorithm: "Atomic Event Sets"

We use a recursive algorithm. The main idea of the algorithm is to take advantage of a data structure that enables immediate testing of sets of atomic events. The algorithm presents the advantage of reducing the dependence on the fact that many complex events may be interested in the same atomic event. The data structure is represented in Figure 4. The entry is made of a large table H of all atomic events in A. Each other table corresponds to a prefix of atomic events, e.g., $H_{1,5}$ to the complex events $a_1 a_5$. A mark in the corner of a cell indicates that the prefix indeed corresponds to a complex event C_j. More precisely, a mark C_k in an a_i cell of H indicates that the k-th complex event is a_i. Similarly, a mark C_k in an a_i cell of table $H_{i_1 ... i_k}$ indicates that the k-th complex event is $a_{i_1} ... a_{i_k} a_i$.

In the implementation, the H and H_ω tables are stored as hash tables.

Let us consider how the algorithm runs using an example. Suppose the document that is being processed has triggered the ordered atomic event set $S = \{a_1, a_3, a_5\}$. We first enter the data structure by $H[a_1]$, i.e., the first event of S. (Since the box is not marked, a_1 alone is not a complex event.) We proceed to H_1. We find a_3. Since the box is marked, we detected the complex event $C_{10} = \{a_1, a_3\}$. Then we proceed to table $H_{1,3}$. We see that the cell containing a_5 is marked, this means we also detect the complex event $C_3 = \{a_1, a_3, a_5\}$. Back in table H_1 we find a_5, but the cell is not marked. Since we have no more atomic events left to process in this sub table, we exit it. We now reenter the data structure with a_3, to find $C_{15} = \{a_3\}$. Since this cell does not point to a sub table, we stop its processing here. Then we enter cell a_5 and detect the complex event $C_4 = \{a_5\}$. We have no more atomic events to process, so our task is over. The document triggered 4 complex events: C_{10}, C_3, C_{15} and C_4.

Observe the algorithm is recursive, so potentially very costly in the length ℓ of the atomic event set S detected (naïvely $O(\ell!)$). Nonetheless our experimentation shows that in practise its behavior is much better. See the analysis further in Section 4.2.

Let us consider the algorithm more precisely. Observe Figure 4. The main components of the data structure are tables. For each table T and each atomic event a_k, we denote by $T[a_k]$ the a_k entry of the table. Starting from a document d_i and the *ordered set* A_i of atomic events detected on document d_i, we produce the set of notifications using the following algorithm:

Algorithm For each ordered set S of atomic events, the result set is *Notif*(H, S) where the function *Notif* is defined as follows:

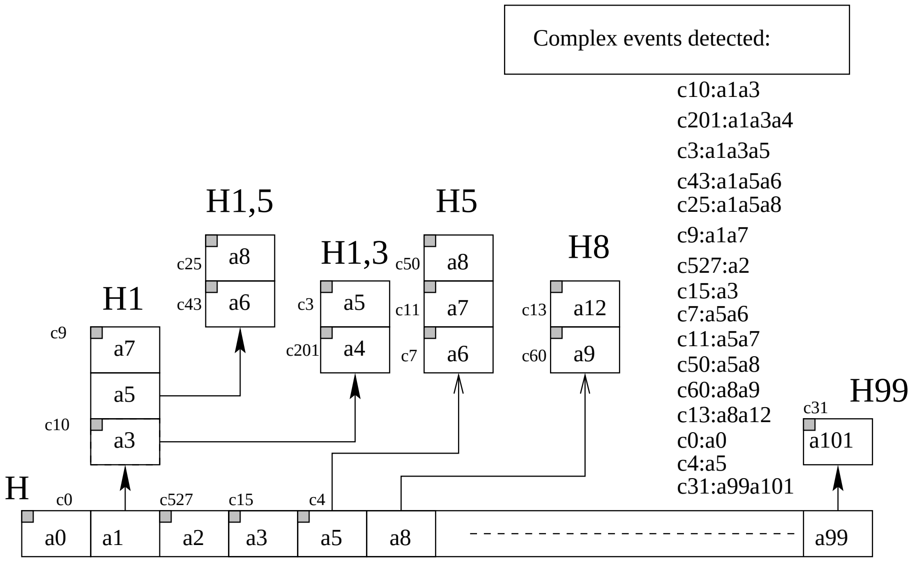

Figure 4: Data structure using a chain of atomic events

T is the H table or an H_ω table

$a_{i_1}...a_{i_n}$ is an ordered set of atomic events

Function *Notif*(T,$a_{i_1}...a_{i_n}$)

1. $Result = \emptyset$
2. For each k in $[1..n]$ do
   (a) if $T[a_{i_k}]$ is marked, add its mark to $Result$
   (b) if $T[a_{i_k}]$ points to some table T', add *Notif*($T', a_{i_{k+1}}...a_{i_n}$) to $Result$

Example Let us run the algorithm on a different input, $S = a_1a_2a_3$. The following sequence of operations occur for *Notif*($H, a_1a_2a_3$):

- $H(a_1)$ is considered without success.
  - *Notif*(H_1, a_2a_3).
  - a_2 is not in H_1.
  - $H_1(a_3)$ is considered and C_{10} is found.
    - H_{13} is not considered since there are no more events to process.
- $H(a_2)$ is considered and C_{527} is found.
  - H_{12} does not exist.
- $H(a_3)$ is considered and C_{15} is found.
  - H_{13} does not exist.

Analysis in brief

We next present a brief analysis. From an algorithmic viewpoint, the performance critically depends on the following parameters:

- the average number h of atomic events in a complex event. A complex event may be, for instance, composed of a URL pattern (e.g., URL="www.cd.com/*"), a status for the document (e.g., UPDATED), and a condition on an element (e.g., category = "toy"). We will see more examples of events when we study the subscription language. Parameter h will typically be small, say between 3 and 4. In the worst case, it is unlikely in our context to exceed 7 or 8.
- the average number ℓ of atomic events triggered by a document. This represents the length of S. Clearly, this parameter depends on the number of subscriptions that are checked in, and on their nature. For instance, if we are monitoring extremely rare atomic events, ℓ will tend to be quite small whereas if we are monitoring very common events (e.g., the word *the* is in the document), it will have a tendency to be rather large. We will assume in our analysis that this parameter ranges from 10 to 100.
- the number of atomic events $Card(A)$ and of complex events $Card(\mathcal{C})$. The system is designed to work with millions of complex events. It is a bit harder to estimate the number of atomic events. We believe that $Card(A)$ will be smaller than $Card(\mathcal{C})$, between 2 to 100 times smaller.
- the average number k of complex events interested in the same atomic event. This is possibly the hardest parameter to estimate. When the number of users of the system grows, this number k could have a tendency to grow. Also, there may be thousands of complex events that will involve the url of Amazon's whereas only very few will be concerned with the url of John Doe's home page. We will estimate k based on the other parameters.

We performed experimentations to evaluate the impact of the various parameters. A major difficulty is that it is very difficult to isolate one parameter. In our experimentation,

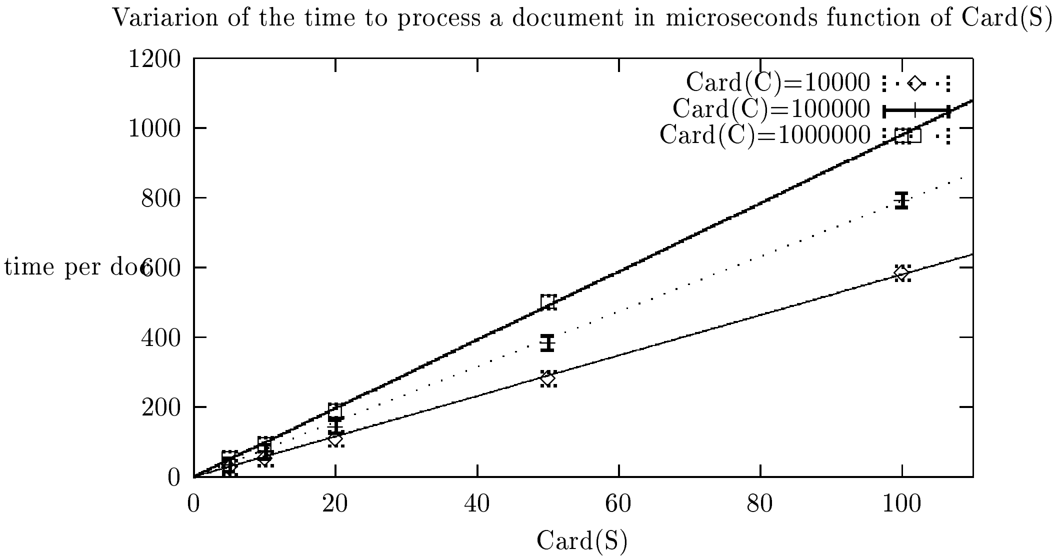

Figure 5: Influence of the ℓ parameter

we completely controlled $Card(\mathcal{C})$, ℓ and h. For $Card(A)$, we fix an upper bound. Then to produce the test set, atomic events are randomly drawn in the set $\{a_1, ..., a_{Card(A)}\}$ with no guarantee that they will all be taken. Finally, to obtain k, we use the fact that[2] $\frac{h}{k}$ can be estimated as $\frac{Card(A)}{Card(C)}$. One would expect that the behavior of the system depends on the values of some of these parameters. It turns out that it also depends in a complex manner on relationships between them such as $\ell >> h$ or $\ell << h$. Thus, a complete analysis of the algorithm is complex and is beyond the scope of this paper. We briefly present here results on the experimentation that highlight the good performance of the algorithm in our context. In all our experiments, we evaluate the average time for processing one document.

First, we fixed all parameters and let ℓ vary. Our tests showed that the processing time (at least in our context) is linear in ℓ. Figure 5 shows the time to process one document in milliseconds as a function of ℓ. The different lines are plotted with different values of $Card(A)$ and $Card(\mathcal{C})$, ranging from 10000 to 1 million. One can note that even for $\ell \approx 100$ the time to process one document is only about 1 millisecond.

Then, we turned to the dependence on h. We considered only the realistic case where $\ell >> h$. Our tests showed that the complexity is independent of h for h ranging from 2 to 10.

Perhaps the most interesting dependence is that on k. To study the dependence on k, we ran our benchmark with $\ell = 20$, $Card(A) = 10000$ and $h = 4$. We controlled the variation of k by varying $Card(\mathcal{C})$ from 10000 to 1 million. Thus k varies from h to $100 \times h$. Figure 6 shows that the dependency is $O(\log k)$.

We next give some intuition to explain complexities that we observed and that may be somewhat unexpected. Consider an atomic event a_i and the substructure consisting of H_{a_i} and all its subtables $H_{a_i \omega}$. Observe that, by definition of k, the atomic event a_i is in k different complex events. Thus the substructure contains at most k cells that are marked, so contains $O(k)$ cells. From this, one can roughly estimate that the processing of the substructure would be in time $O(k)$, so the processing of a set of ℓ atomic events would only be in $O(\ell \times k)$. Furthermore, a more careful analysis shows that the substructure contains on average much less than $O(k)$ cells. Indeed, experimentation shows that the algorithm runs in $O(\ell \times \log k)$.

Measures show that the algorithm can process several thousand sets of atomic events per second on a standard PC. This should be compared to the rate of our crawler. Currently, one Xyleme crawler [19] is able to fetch about 4 million pages per day, that is approximately 50 per second. Thus the *Monitoring Query Processor* we described here can support the load of about 100 crawlers. The data structures we use require about 500MB of memory for $Card(A) = 10^6, Card(C) = 10^7$ and $h = 10$. Like the other modules, the Monitoring Query Processor was designed so that its task could be distributed on several machines. Typically, one can use distribution along two directions:

1. Processing speed: we can split the flow of documents into several partitions and assign a Monitoring Query Processor to each block of the partition.

2. Memory: we can split the subscriptions into several partitions and assign a Monitoring Query Processor to each block. This results in smaller data structures for each processor.

Based on these two kinds of distributions, we obtain a very scalable system.

[2]Its demonstration is simple and omitted here.

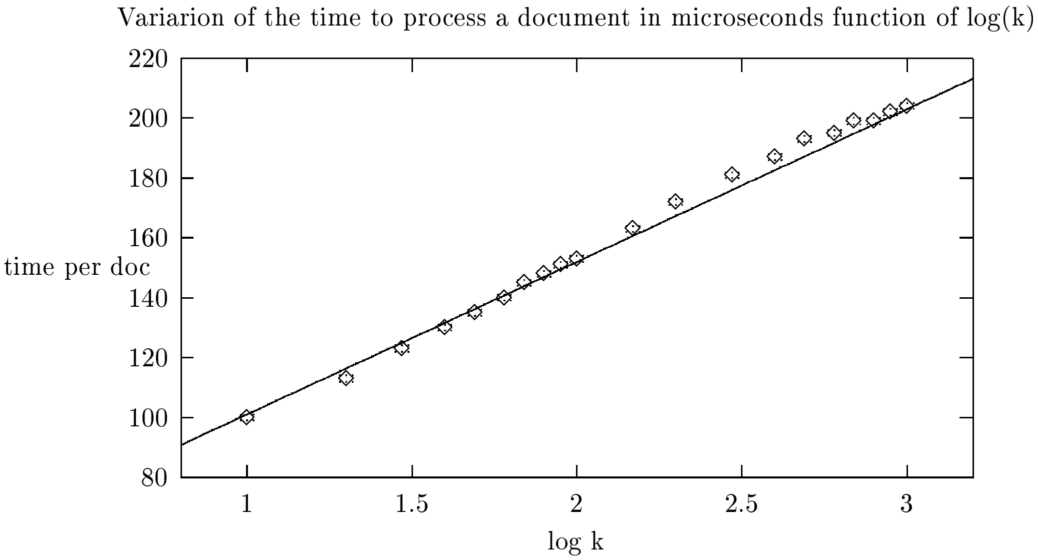

Figure 6: Influence of the k parameter

The Monitoring Query Processor is generic in that it may be used for a very wide range of applications. We next turn to parts of the system that are more specific to monitoring a flow of Web pages. In the next section, we consider the subscription language; and in the following one, the specific alerters we use.

5. THE SUBSCRIPTION LANGUAGE

In this section, we briefly describe the subscription language.

As we have seen in Figure 2, a subscription consists of the following parts: (i) monitoring queries, (ii) continuous queries, (iii) report specification, and (iv) refresh statements, and is of the following form:

```
subscription name
  monitoring...        % (i)
  continuous...        % (ii)
  report when...       % (iii)
  refresh...           % (iv)
```

We next consider (i,ii,iii) in turn; (iv) will be ignored in this paper. Then we consider the issue of controlling subscriptions to block requests that would require too many resources. To conclude this section, we briefly consider the relationship between monitoring and continuous queries.

5.1 Monitoring Query

A monitoring query has the general form:

```
select  result
(from   from-clause)
where   condition
```

The *from* clause This clause may be omitted because we know the data that is being filtered, i.e., the document that is currently being processed. This current document is denoted *self* in the query. A *from* clause may be used to attach variables to elements of the current document.

The *select* clause This clause describes the data the resulting notification should contain based on constants, $self$ or variables defined in the *from* clause. The notification itself is an XML element. For the moment, we have not implemented this part of the system. Notifications simply return the URL of the document that triggered the monitoring query and basic informations about the document.

The where clause This clause is a condition that consists of a *conjunction* of *atomic conditions*. An atomic condition may be of one of the following:

URL **extends** string	URL = string
DTDID = integer	DTD = string
DOCID = integer	domain = string
filename = string	

where *domain* is one of the semantic domains that Xyleme uses to classify documents (e.g., biology), *DOCID, DTDID* are internal identifiers, and *filename* is the tail of an URL (e.g., index.html) . Other atomic conditions deal with information about documents that either is maintained by Xyleme or can be obtained by the alerters when the document is fetched: e.g.,

- *LastAccessed* $\langle$comparator$\rangle$ date
- *LastUpdate* $\langle$comparator$\rangle$ date
- *self contains* string
- $\langle$*status*$\rangle$ *self*

where ⟨*status*⟩ is defined as one of: *new, unchanged, updated, deleted*[3].

The most interesting atomic conditions deal with element values inside a document and are meaningful only for XML documents. Their syntax is as follows[4]:

(⟨*change*⟩) ⟨*element-name*⟩ (*contains string*)

The ⟨*change*⟩ keyword means some change pattern of the elements of a given name (tag) will be monitored, e.g., we are only interested in documents containing a new element with the tag *product*. We may also impose that the element contains a given string, e.g., we are only interested in document with an element containing the word *electronic* and under the tag *category*. The semantic of *contains* is that this string occurs within the element. We also support *strict contains string* that specifies that the string must be present directly in the text of the element.

A *where* clause is a conjunction of atomic conditions, e.g.:

- *new self*
  and URL extends "http://theuniv.xyz/mygroup/"
  (new documents with a certain URL pattern)
- *new Product*
  and URL extends "http://www.thecom.xyz/catalog/"
  (documents in a particular catalog containing a new product)
- *updated Product contains "camera"*
  and DTD="http://www.thecom.xyz/dtd/catalog.dtd"
  (documents with a particular DTD containing an updated product containing the word camera).

Each atomic condition is mapped to an atomic event. Note that it is likely that each document we read will raise one atomic event involved in at least one subscription, i.e., one in *new, unchanged, updated*. So, if we are not careful we would have to raise one alert for each document, that is, we would have to send a set of atomic events to the Monitoring Query Processor for each document. To avoid this, we distinguish between *weak* events (*new*, *modified*, *unchanged*) and *strong* events (all other atomic events). We disallow *where* clauses composed solely of a weak atomic condition. Thus, a document is detected as potentially interesting if at least a strong atomic event of interest for a subscription is detected. In this case only, an *alert*, consisting of the set of atomic events detected plus some extra data (defined by the *select* clause) is sent to the *Monitoring Query Processor*.

5.2 Continuous queries

As mentioned in Section 2, a subscription may also include one or more continuous queries that will participate in the notification stream. A continuous query consists of a standard Xyleme query [2] plus a condition that specifies when to apply the query. Typically, this condition involves a frequency (e.g., every week). The continuous query may also be triggered by a notification sent by the subscription processor.

Consider the continuous query:

```
continuous delta AmsterdamPaintings
  select p/title
  from   culture/museum m, m/painting p
  where  m/address contains "Amsterdam"
  when   biweekly
```

that asks for the names of all paintings found in an Amsterdam museum. Here *culture* is an *abstract* domain that provides an integrated view of museum resources. We ask the system to evaluate the query twice a week. The report will therefore contain a list of results:

```
<AmsterdamPaintings> ... </AmsterdamPaintings>
<AmsterdamPaintings> ... </AmsterdamPaintings>
```

The use of the keyword *delta* specifies that we are interested by changes to the result and not by the result *per se*, and that we are interested in storing the delta of this document [17]. So the first time the query is evaluated, we get its answer, but later, we only receive the modifications of the result such as:

```
<AmsterdamPaintings-delta> ...
<inserted ID="556" parent="556" position="4">
     <title> ... </title>
     <title> ... </title>
<updated ID="332" note="not available
  -- visiting MOMA">
</AmsterdamPaintings-delta>
```

Identifiers such as 556 are used here as the foundation of a naming scheme for XML documents that makes the specification of changes easier. *Deltas* based on XIDs provide a compact naming of the elements of the documents that is the basis of the versioning mechanism of the system. In particular, the new version of a document can be constructed based on an old version and the delta. We also provide a practical change editor for the visualization of changes in XML documents or query results in the spirit of change editors as found, for instance, in MS-Word. A key component for defining these changes is the *diff* package for XML that we developed. A detailed presentation of these mechanisms can be found in [17].

In the previous example, the continuous query is asked with some time frequency. It is also possible to trigger the evaluation of a continuous query with notifications from a monitoring query as in:

```
subscription XylemeCompetitors

monitoring
    select <ChangeInMyProducts/>
    where URL = ``www.xyleme.com/products.xml''
          and modified self
```

[3]We will not discuss deletions in this paper. It is not an obvious notion since deletion is rarely explicit on the web.
[4]Parenthesis denote an optional parameter

```
continuous MyCompetitors
    select ...
    when   XylemeCompetitors.ChangeInMyProducts
```

This requests the system to reevaluate the query whenever it detects a change in the *products.xml* page.

5.3 Reporting

The report part of a subscription has the following form:

```
select     ...              % report query
when       ...              % reporting condition
(atmost)   ...              % limiting conditions
(archive)  ...              % archiving information
```

The report query is a standard Xyleme query that takes as input the current set of notifications, i.e., an XML document, and produces another XML document. The *when* clause tells when to fill in a new report. It follows the syntax:

$\langle WhenClause\rangle$	:=	*immediate* |
		$\langle Event\rangle$ (*or* $\langle Event\rangle$)$*$
$\langle Event\rangle$	:=	$\langle frequency\rangle$ | *each* $\langle$*date-pattern*$\rangle$ |
		$\langle count\rangle$ $>$ *integer* |
		$\langle notification\_name\rangle$
$\langle frequency\rangle$	:=	*dayly* | *weekly* | *biweekly* | *monthly*
$\langle count\rangle$	:=	*count* |
		count "(" $\langle notification\_name\rangle$ ")"
$\langle$*date-pattern*$\rangle$	:=	*monday* | *tuesday...* |
		$January1st$ | $January2nd...$

where $notification\_name$ is the name of a monitoring query (e.g., UpdatedPage). The semantics of these report conditions with a few exceptions should be clear. A condition of the form $count > 500$ means that a report is generated whenever 500 notifications have arrived. A condition $count(UpdatedPage) > 10$ means that the report is generated when 10 *UpdatedPage* notifications have arrived. Immediate means that as soon as something is added to this subscription, a report is generated. The disjunction of several conditions means that a report is generated whenever one of the reporting conditions holds. The generation of a report for a subscription empties the global buffer of notification answers.

The *when* clause is compulsory whereas the last two clauses are optional. The *atmost* clause sets a limit to the reporting query. For instance, *atmost 500* means that after 500 notifications, we will stop registering the new notifications until the next report. Also, *atmost weekly* means that, we do not send a report more frequently than once a week even if the *when* condition "triggers" more often.

Finally the *archive* clause requests the results of this particular subscription to be archived for some period of time: For instance, *archive monthly* requests to archive the reports for this particular subscription for a month before garbage collecting them. We refer to [14] for more details on the *reporter*.

5.4 Controlling subscriptions

It should be noted that the cost of some monitoring or continuous queries may be quite prohibitive. This is the reason why we only allow the condition *extend URL*, and not the matching of an arbitrary pattern. This also indicates that we should prevent the user from using *extends http:* as a subscription condition. Similarly, one would like to prevent the use of *contains* conditions on too common a word such as "*the*" or attempting to refresh every day all documents in too wide a domain such as *biology*. As a last example, we do not want to trigger a continuous query with too frequent an event, e.g., an event that would occur every minute. To control this, we could use a cost model to estimate *a priori* the cost of a subscription and to restrict the right of specifying expensive subscriptions to users with appropriate privileges. Perhaps a simpler solution would be to allow arbitrary subscriptions, but inhibit them *a posteriori*, if the system finds out they require too much resources.

To the same end of avoiding the waste of resources, we mentioned in the introduction the possibility of using techniques as in [13, 8] to factorize the work in monitoring and continuous queries. Although we have not introduced such optimizations yet, we do however let some subscriptions share monitoring and continuous queries from other subscriptions. These are called *virtual subscriptions*. For instance, a user may request the following subscription (purely virtual):

```
subscription MyVirtualXyleme
virtual MyXyleme.Member
```

A user specifying such a subscription is simply registering to a subscription owned by another user. In other words, we distinguish here between the possibility of creating monitoring and continuous queries (expensive for Xyleme) with that of subscribing to them (that only puts stress on the *Reporter*).

5.5 Monitoring vs. continuous queries

To conclude this section, we briefly examine the relationship between monitoring and continuous queries.

We have already seen a first interaction, i.e., the control of the triggering of a continuous query by the monitoring query processor. It should be noted that there is also an overlap in functionalities between the two features. For instance, consider the monitoring of all documents of a certain DTD discovered by Xyleme during one week:

- One can obtain this information easily using monitoring.
- One can also obtain it using a query that would recover all documents of that DTD and select those with a (Xyleme) creation date of less than a week.

In some cases, only one may be feasible:

monitoring only Suppose that every week we want to obtain all documents of a certain DTD where a new *product* element has been inserted. It is not possible to obtain this information using a continuous query, since the old version of the data will not be available a week later when the query is run. We can use versioning [17], but it is costly in space, and introduces additional processing.

continuous only Our monitoring is performed on the fly with no access to the repository. Thus, it is not possible to support queries spanning multiple documents. For instance, if we are interested in new documents of a particular DTD with *product*s that are also supplied by, say IBM, only the continuous query solution is feasible[5].

When both monitoring and continuous queries are feasible, the choice of one over the other is an interesting research topic that will not be addressed here. In particular, it would be interesting to introduce optimization techniques that would make possible the implementation of a monitoring query using a continuous one and vice versa. Observe that the difficulty is in evaluating the cost of each approach, since they stress very distinct portions of the system, namely the acquisition module [19] for monitoring and the query processors [2] for continuous queries.

6. ALERTERS

Alerters are the first step of the notification chain. When a document is being retrieved, it is first handled by the URL manager. If it is an XML document, it is managed by the XML loader. (For HTML documents, the story is a bit different but similar and will not be considered in detail here.) Alerters are in charge of detecting atomic events and sending them to the *Monitoring Query Processor*. This is the topic of this section. We next consider the architecture, then two kinds of alerters, the URL and the XML Alerters.

6.1 Architecture

To avoid unnecessary network traffic, alerters have to be set as close as possible to the modules they monitor. For instance, the URL Alerter must be placed next to the URL manager (that is gathering metadata about documents), and the XML Alerter next to the XML Loader (that is in charge of loading 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. Then it communicates to other modules (see Figure 7):

1. an *Alerter* sends its alerts and their associated information to the *Monitoring Query Processor* that is in charge of handling them.

2. an *Alerter* may send the atomic events it detected on a document to another *Alerter* for further investigation onto some data.

For performance reasons, each alerter uses different threads for input and output.

[5]Observe furthermore that if we are interested by the documents with new products such that these products are supplied by IBM, then none of the approaches will work. In such cases, we would have to perform serious query processing at the time the document is discovered or use versioning of the documents of interest.

An essential aspect of this process is that we collect all the atomic events of interest on a given document before sending them to the *Monitoring Query Processor*, and thus the Monitoring Query Processor receives simultaneously all the atomic events concerning a document. For instance, for an XML document, some atomic events may be detected by the URL Alerter, then sent to the XML alerter that may detects more atomic events. A single alert is then sent to the *Monitoring Query Processor*. Based on the content of these alerts, the system may decide to do more processing, e.g., send a notification to the Reporter or to the Trigger Engine. Thus we use an approach that is typically based on a *document flow*, vs. workflow.

6.2 URL Alerters and pattern detection

In this section, we briefly consider the URL Alerter and, in particular, its main task, namely, pattern detection.

The role of the URL Alerter is to construct, for a document, the sequence of atomic events that have been detected on a document. It must produce a sorted sequence since, as we have seen, the Monitoring Query Processor takes advantage of the ordering. We expect such a sequence to be reasonably small (at most a few thousand).

The URL Alerter must be able to manage millions of atomic events, and detect such events on the incoming data (here some metadata about the page that is being fetched) without slowing down the rest of the system. The main datastructure of the URL manager is devoted to this detection. (Again, we will ignore here the issues of the runtime update of the data structure.) To be more precise, we use several datastructures depending on the nature of the conditions, e.g. one for *URL extends string* and a different one for *domain = string*. Our data structures are standard and essentially use hash tables and extensible arrays implemented with STL [26].

We next focus on the detection of URL patterns that ais by far the most critical in terms of performance. We handle three kinds of URL pattern detections:

1. URL extends string (e.g., pattern http://www.xyleme.com/*), and

2. filename = string (e.g., pattern */Xyleme2000.xml).

3. URL = string (e.g., pattern http://www.xyleme.com/index.html)

Each one of them is handled using a separate data structure. To handle *extends*, given the URL of the document that is being fetched, we look up each of its prefixes to see if it matches the 'URL*' pattern of some atomic event of interest. The dominating cost is the look-up in the million-records hash table. To obtain a linear lookup cost, we tried using a dictionary structure. This improved the speed by about 30 percent. But in terms of memory size, the overhead was too high.

Observe that the URL Alerter is working concurrently with the URL Manager. Typically, alerters should not slow down the process they monitor by holding up resources. Thus, in particular, it is important that the alerters be able to handle many documents concurrently.

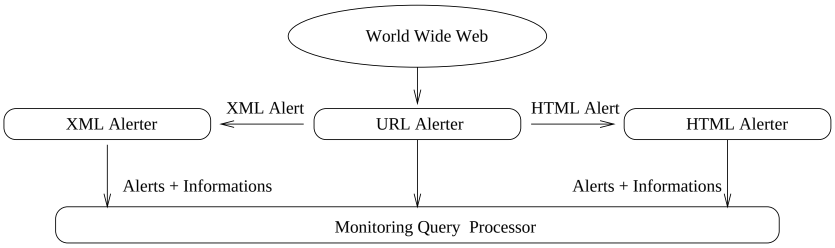

Figure 7: Overview of Alerters architecture

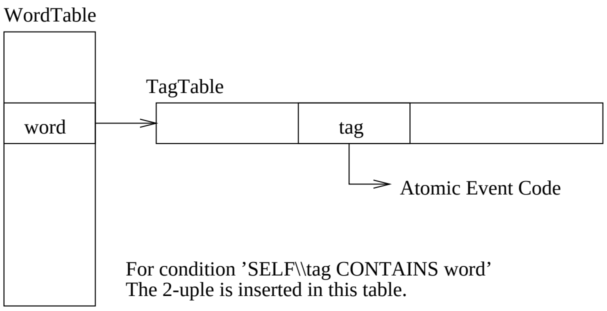

Figure 8: Registering (Tag, Word) conditions

6.3 XML Alerter

The main purpose of this alerter is to detect atomic events of the form:

$(\langle change \rangle)$ $\langle$*element-name*$\rangle$ ((*strict*) *contains string*)

For the detection of changes (elements inserted, updated, etc.), we compute the *delta* between the document that is being loaded and its previous version (if available) [17]. We will ignore this aspect here and will focus on the detection of words of interest in the document:

$\langle$*element-name*$\rangle$ (*strict*) *contains string*

Recall that *contains* means that the word with a particular tag must be found anywhere in the subtree. On the other hand, *strict contains* means that the element with this tag directly contains this word, i.e., in DOM terms [11], a node with this particular tag has a data child containing this particular word.

Our algorithm relies on the postfix traversal of the DOM tree. For each node n in the tree, let $p(n)$ be the pair: $(level, content)$ where:

1. *level* is the level of the node in the tree;
2. *content* is either the tag if the node is an element node or the data if the node is a data node.

Let $\{n_i\} = \{n_0, n_1, n_2, ...\}$ be the list of the nodes in a postorder traversal of the tree. To handle *contains*, we use the flow of $\{p(n_i)\}$. While processing $p(n_i)$ in the flow $\{p(n_i)\}$, it is easy to have the list of words in the subtree rooted at n_i and those directly below n_i. To do that, two different data structures are used (See Figure 8):

1. For each "interesting" word w, we use a hash table called $TagTable[w]$. For each tag l such that (l, w) is an atomic event of interest, the table provides its code. The *TagTable*'s can be accessed from another hash table named *WordTable*. Due to the different nature of condition *contains* and *strict contains*, we need two data structures, one for each.
2. Let us first consider *contains*. For the document that is being processed, we use a data structure that provides for the node being processed, the list of words of the tree rooted at this node, and this at no cost. This is where we benefit from the postordering in the input to the algorithm. We can essentially handle the words in a stack of lists of words. Note that we can save some space and processing by keeping in this structure only words that are interesting, i.e., are entries in the contains *WordTable*. Now let us consider *strict contains*. It is somewhat similar since two data children of the node may be separated by an element node, so we have to process the trees rooted at child element nodes before processing the node itself.

Observe that this second data structure may contain at some point in the worst case all words of a document, i.e. be roughly of the document's size. With respect to time, we may have to perform one lookup for each word of the document at each level of the document, which leads in the worst case to $Size \times Depth$, where *Size* is the number of words in the documents, and *Depth* is the maximum depth of the tree structure of the document. For XML documents found on the web, it turns out that the depth of the document is rather small, so on average, this is an acceptable cost.

In our experiments, the *Alerters* could easily support the rate of fetching documents on the web imposed by the crawlers and URL managers.

7. CONCLUSION

The work described here has been implemented and integrated to the Xyleme system. We already mentioned some limitations such as: the refresh clause, the (full) select clause, and the HTML alerters are not yet implemented. All the rest is running.

Future developments include:

- We started a formal study of the *Monitoring Query Processor*'s algorithm. This turns out to be nontrivial and quite interesting from an algorithmic analysis viewpoint.
- We are finishing the implementation of the subscription system, e.g., *Xyleme Select* module.
- Little work has been invested on the *Trigger Engine* and the continuous query module. We intend to work on this next. We mentioned the use of optimization techniques.
- It is interesting to also consider other practical uses of the Trigger Engine, e.g., for the analysis of certain documents, their automatic classification, versioning, etc.
- One might also consider using a combination of the *Monitoring Query Processor* and the *Trigger Engine* to monitor complex events that would include *disjunctions* of atomic conditions.

Acknowledgments We would like to thank Jérémie Jouglet and David Leniniven for implementing the reporter, INRIA postmaster, and finaly the members of Verso group and Xyleme SA. for the valuable comments they provided.

8. REFERENCES

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web*. Morgan Kaufmann, California, 2000.

[2] Vincent Aguilera, Sophie Cluet, Pierangelo Veltri, and Fanny Watez. Querying xml documents in xyleme. *ACM SIGIR Workshop on XML and information retrieval*, 2000. To appear.

[3] Apache web server. http://www.apache.org/.

[4] The internet archive. http://www.archive.org/.

[5] Kevin Atkinson. Mysql++ a c++ api for mysql, 2000. http://www.mysql.com/documentation/.

[6] S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. *Proceedings of the IEEE International Conference on Data Engineering*, pages 4–13, 1998.

[7] S. Chawathe, S. Abiteboul, and J. Widom. Managing historical semistructured data. *Theory and practice of object systems*, 5(3):143–162, August 1999.

[8] Jianjun Chen, David DeWitt, Fend Tian, and Yuan Wang. Niagaracq: A scalable continous query system for the internet databases. *ACM SIGMOD*, page 379, 2000.

[9] Webcq, opencq webpage. http://www.cc.gatech.edu/projects/disl/WebCQ/.

[10] Corba web page. http://www.omg.org/.

[11] Document object model (DOM) level 1 specification version 1.0, October 1998.

[12] F. Fabret, F. Llirbat, J. Pereira, and D. Shasha. Publish/subscribe on the web at extreme speed. submitted to publication, 2000.

[13] Eric Hanson, Chris Carnes, Lan Huang, Mohan Konyala, Lloyd Noronha, Sashi Parthasarathy, J.B. Park, and Albert Vernon. Scalable trigger processing. *Proceedings of the 15th International Conference on Data Engineering*, pages 266–275, 1999.

[14] Jérémy Jouglet. Souscription de requêtes dans un entrpôt de données xml. Stage d'option scientifique de l'École Polytechnique, 2000.

[15] Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *IEEE TKDE*, 11(4):610, 1999.

[16] Ling Liu, Calton Pu, Wei Tang, and Wei Han. Conquer: A continual query system for update monitoring in the www. *International Journal of Computer Systems, Science and Engineering*, 2000.

[17] Amélie Marian, Serge Abiteboul, and Laurent Mignet. Change-centric management of versions in an xml warehouse, October 2000. BDA'00.

[18] Alain Michard. *XML, langage et applications.* Eyrolles, Paris, 1999.

[19] Laurent Mignet, Serge Abiteboul, Sébastien Ailleret, Bernd Amann, Amélie Marian, and Mihai Preda. Acquiring xml pages for a webhouse, October 2000. BDA'00.

[20] Mind-it web page. http://mindit.netmind.com/.

[21] Guido Moerkotte. The aodb relational system. U. Mannheim, personal communication, 1999.

[22] Niagara webpage. http://www.cs.wisc.edu/niagara/.

[23] Northern light news search. http://www.northernlight.com/news.html.

[24] Information on clusters of pcs. http://www.alinka.com/fr/index.htm.

[25] R.T. Snodgrass, editor. *The TSQL2 temporal query language*. Kluwer Press, 1995.

[26] Bjarne Stroustrup. *The C++ programming language.* Addison-Wesley, Reading, Massachusetts, special edition, 2000.

[27] W3C. *eXtensible Markup Language (XML) 1.0*, february 1998.

[28] World Wide Web consortium page on XML. http://www.w3c.org/TR/REC-XML.

[29] J. Widom and S. Ceri. *Active database systems: Triggers and rules for advanced prcessiong.* Morgan-Kaufmann, California, 1995.

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

[31] Xyleme home page. http://www.xyleme.com/.