Acquisition and Maintenance of XML Data from the Web*

Laurent Mignet[†] Mihai Preda[†] Serge Abiteboul[†] Bernd Amann[‡]
Amélie Marian[§]

ABSTRACT

We consider the acquisition and maintenance of XML data found on the web. More precisely, we study the problem of discovering XML data on the web, i.e., in a world still dominated by HTML, and keeping it up to date with the web as best as possible, under set resources.

We present a distributed architecture that is designed to scale to the billions of pages of the web. In particular, the distributed management of meta-data about HTML and XML pages turns out to be an interesting issue.

The scheduling of the fetching of the page is guided by the importance of pages, their expected change rate, and subscriptions/ publications of users. The importance of XML pages is defined in the standard manner based on the link structure of the web graph. It is computed by a matrix fixpoint computation. HTML pages are of interest for us only in that they lead to XML pages. Thus their importance is defined in a different manner and their computation also involves a fixpoint but on the transposed link matrix this time. The general scheduling problem is stated as an optimization problem that dispatches the resources to various tasks such as (re)reading HTML pages for discovering new XML pages or refreshing already known XML pages.

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 mention first experiments.

[†]I.N.R.I.A. VERSO, BP 105 78153 Le Chesnay Cedex, France, Email: firstname.lastname@inria.fr
[‡]Vertigo - C.N.A.M 292, rue Saint Martin 75141 Paris Cedex 03, France, Email: amann@cnam.fr
[§]Columbia University, New York, USA, Email : amelie@cs.columbia.edu

Keywords

XML, HTML, Warehouse, Web, Refresh Policy, Page Importance, Change Control.

1. INTRODUCTION

The Web is huge and keeps growing at a healthy pace. Most of the data is unstructured, consisting of text (essentially HTML) and images. Some is structured and mostly stored in relational databases. All this data constitutes the largest body of information accessible to any individual in the history of humanity [4]. A major evolution is occurring that will dramatically simplify the task of developing applications with this data, the coming of XML [24, 1]. This new technology provides a major opportunity for changing the face of the web in a fundamental way.

In the present paper, we consider the problem of acquiring XML data from the web, i.e., from a world still dominated by HTML [25], and keeping it up to date with the web as best as possible. This work is part of the Xyleme project [27] aiming at the development of a *dynamic XML warehouse for the web.*

The problems we consider are typical warehousing problems [26]. They take a different flavor when set in the context of the web. Scalability becomes a major issue. First, we need to obtain as many as possible XML pages present on the web. Then, we need to keep them up to date. These two aspects of *acquisition* and *maintenance* of XML data are essential and particular to our context. Clearly, given that network, storage and processing resources are always limited, we will have to be satisfied with possibly getting only part of the data of interest and with maintaining the retrieved data up to date with some level of staleness.

We present here an architecture for the acquisition and maintenance of web data in a large scale repository. Acquisition is guided by the *importance* of pages. Importance is determined by some absolute value derived from the link structure of the web. The intuition [6, 16, 14] is that an important page referring to a page transfer some of its importance to the pointed page. Importance also depends on the particular purpose of the warehouse and specific user requirements. So, for instance, here, we are concerned with an XML repository, so a page leading to many XML resources becomes particularly interesting as a potential source to discover new XML pages.

Once a page has been obtained, it has to be maintained up to date by the system. In contrast to database systems, we are not informed in general when a page changes. So, we have to monitor the web and regularly check whether each page has changed. Influenced by the works of Cho, Haveliwala and Garcia-Molina at Stanford, we will access more frequently important pages. Also, we will attempt to refresh pages that change rarely (e.g. archives) less often than pages that change frequently.

This said, acquisition and maintenance become optimization problems. Given specific resources and most notably the bandwidth of the crawler and the storage space, maximize the quality of the repository, i.e., the relevance of the stored pages and their freshness. We develop the technical tools to estimate the importance of pages and their change rate. Based on that, we propose a cost model and show how an optimal strategy can be obtained.

Novelties and contributions The general architecture is new and was designed to scale to the billions of pages of the web. In contrast to existing search engines that typically discover pages using a breadth first strategy, acquisition is guided by the importance of pages, the domain of interest (here XML pages) and user specific requirements. Similarly, refresh is guided by the same criteria. The estimates for absolute importance and change frequency are improvement over existing techniques. All the ideas presented here have been implemented and tested.

The paper is organized as follows. In Section 2, we present the problem in more details. The architecture is discussed in Section 3. The scheduling of pages is considered in Section 4. The estimation of the importance of pages is the topic of Section 5 and that of change frequency of Section 6. In a last section, we present the status of the work and discuss future directions of research.

2. THE GENERAL PROBLEM

Web data can typically be obtained by two complementary techniques called *pull* and *push*. Pull involves crawling the web to *discover* data of interest. Push implies an active participation of web servers. For instance, in our context, Web masters aware of Xyleme publish available XML resources, i.e., actively informs the system of the existence of such data. When a page of interest is in the repository, we need to keep it up to date with its version on the web, i.e., it is necessary to *refresh* the page periodically.

Thus, there are two essential aspects: (i) the discovery of new pages and (ii) the refreshing of known pages. Observe that the ratio between the two varies in time, e.g., discovery is typically more important in an early phase. Note also that the number of new pages we want to obtain may depend on external conditions such as available storage. Both will be guided by an estimation of *page importance* that we will explain further. Refreshing is also guided by an estimation of *page change frequency.*

The choice of which page to read next is a complex task. Resources are limited and the system can only read a certain amount of pages per time unit, say per day. Since the pool of web pages is very large, a naive strategy of *reading all pages* and/or *refreshing all pages periodically* may result in never obtaining certain pages and refresh cycles of several weeks. The following observations can lead to significant improvements of the quality of the warehouse under fixed resources:

1. we should privilege pages that are requested by users and/or fall within the specific domain of interest [7];
2. we should not waste resources to often read unimportant pages;
3. we should not waste resources to often read a page that changes rarely or never (an archive or a forgotten page); and
4. we should not waste resources to try to keep fresh a page that has a too high change frequency (e.g., stock quotations).

Based on these observations, refreshing becomes an optimization problem, namely minimizing the gap between the web and the warehouse given limited resources (e.g., loading capabilities), for some cost function taking into account the specific interest of users, the importance and change rate of pages.

We distinguish between two kinds of pages, HTML and XML. Together (HTML+XML), these pages and their link structure form the backbone of the web. Essentially, we discover the web by following their links. XML also plays for us the role of the *domain of interest* since we are building an XML repository. Different applications may have different interests, e.g., pages dealing with genomics. All the machinery we develop for XML may be applied to such contexts.

Discovery vs. refresh A main issue is to control the allocation of resources. Let us consider discovery in more detail. We need to crawl new portions of the web to discover new pages. We also need to crawl pages that we have already read (refresh them) because their new versions may lead to new pages of interest.

To allocate resources to discovery, we use three parameters.

The first two, called σ_X, for crawling XML, σ_H HTML respectively, specify the percentage of crawled pages that is allocated to read for the first time XML pages (σ_X) and HTML pages (σ_H).

It is important to observe that, in general, we don't know a priori whether a page is XML or HTML. Typically, we know the type of a page from its MIME type [13] returned by the web server or by analyzing the page. When we discover a new URL, we do not have such information. However, we may consider that a page is an *XML suspect* if, for instance, its URL ends with the suffix ".xml", ".wml", ".rss", etc., or if it is a link with an unknown suffix that we discovered in an XML page. The ratio σ_X is for such pages. Observe that this notion of suspect extends to other applications. Suppose for instance we are interested in pages about genomics. An analysis of a URL together with information about how we obtained this particular URL may give us indications about the likelihood that the page deals with genomics.

This $\sigma_X + \sigma_H$ percent of the pages are pages read for the first time (discovery ratio). The remaining percent, i.e., $(100 - \sigma_X - \sigma_H)$ is used to refresh pages. These parameters are fixed by the system administrator, e.g., $\sigma_X = \sigma_H = 20\%$, and can be changed dynamically.

We fix a *refresh cycle time*, 6 hours in our experiment. This means that a page will never be automatically read more than once every 6 hours. (On the other hand, there is no limitation on the *explicit* refresh requests a user may issue.) This is to avoid allocating too much resources to try to maintain up to date pages that change all the time. Observe that during the first cycle there is no page to refresh and all resources are devoted to discovery. Note also that, as we shall see, this value cannot be chosen arbitrarily small (order of minutes). Thus, in that sense, the system cannot be used to perform quasi-real time monitoring.

The refresh of a page may also eventually lead to discovering new pages. Furthermore, the refresh of an XML page of the repository serves also the purpose of keeping the directory up to date. The system administrator chooses an *importance factor* χ giving the relative importance between HTML and XML. This factor influences our estimation of the importance of pages thereby guiding the frequency of refresh for pages. If we say that the average importance for HTML pages is 1, the average importance for XML pages will be χ. This second parameter therefore guides the refreshing of HTML pages vs. that of XML. A smaller χ will lead to refreshing HTML pages more often. A larger χ will tend to privilege the refreshing of XML pages vs. the refreshing of HTML pages (and possibly the discovery of new XML pages)

Note that the first two parameters σ_X, σ_H are absolute whereas the last χ is relative. This was chosen because it is difficult to select an absolute value for refreshing XML/HTML pages without any knowledge of the number of such pages, a typically changing statistics. The situation would be even more complex in a context where the same system deals with several domains of interest.

Importance of XML pages

The entire acquisition/maintenance for the pages we store in the repository is guided by page importance. In Xyleme, we use a combination of two criteria:

1. the page structure of the web and the standard idea that page importance is defined by the pages pointing to it and their importance [6].
2. some specific interest in some pages specifically expressed by customers in a publication/subscription mechanism [18].

The interest of a customer or a set of customers can easily be captured by extending the web with some *virtual* pages of fixed importance pointing to the pages they are interested in. This yields a definition of importance customized to a specific application.

Based on these criteria, we can compute the importance of pages using a standard fixpoint technique in the style of [6, 14, 16].

Usefulness of HTML pages

We use this standard notion of importance intensively: (i) to guide the discovery of new XML and HTML pages, and (ii) to guide the refresh of already read XML pages. We use a different measure for guiding the refresh of HTML pages. As mentioned above, an HTML page may evolve in time and lead to new XML pages when crawled again. So, to be sure not to ignore new portions of the XML web, we need to read HTML pages more than once, i.e., refresh them. However, the classical importance based on the web structure is not appropriate for such pages. An HTML page may be very important in the web and not lead to any XML page, and thus be useless from a Xyleme viewpoint. For HTML pages, we use a more appropriate measure based on the XML pages it leads to and their importance. To distinguish it from the previous notion, we call it *usefulness*. As we shall see, usefulness leads to a similar fixpoint computation but "inverse" in the sense that the importance of pages propagates backward instead of forward.

To illustrate importance and usefulness, consider the minuscule web in Figure 1. In the graphs, XML pages are in grey and HTML pages in white. Black is used for publication/subscription pages. The graph on the left provides the standard importance of pages based on the link structure of the web. The importance is given by the radius of circles. Note that Page B is very important because many pages point to it including two "important" pages. The graph in the middle uses the same notion for XML pages but for HTML pages, it uses usefulness. Note that Page B is not useful because it does not lead to any XML page. Conversely, Page A is more useful because it points to important XML pages. Finally, observe the impact of publication/subscription (graph on the right). In particular, no web page Xyleme knows of points to page C. Page C is only known because of publication and gets its importance from it.

Guided vs. standard search

We use a *guided strategy* to crawl the web. Classical search engines such as AltaVista [3] typically use a *standard strategy* that consists in, starting from some seeds, following links to discover more and more pages, say in breadth-first search. The standard strategy presents a high level of randomness because pages arrive to the FIFO in an order that depends heavily on the servers response time. (It is clearly not *totally* random because of the choice of a breadth-first traversal.) Standard crawling is conceptually easy and can be very efficiently performed.

Because of its randomness and its efficiency, the standard strategy presents advantages. Indeed, we started first by using a combination of standard and guided crawlings. In our first implementation, it was possible to allocate resources to each of the two kinds of crawling. Based on first experiments, we decided to use only guided crawling for the following reasons:

Randomness A problem of the guided strategy is that it may tend to focus too much discovery on some portions of the web. More precisely, the computation of page importance is biased (in particular at the beginning of the crawl) because we only know a portion of the web. This may tend to limit the crawler to already known portions of the web at the cost of ignoring important other portions. Our first experiments show that the web is connected enough so that this does not happen.

Efficiency Page scheduling with the guided strategy is also very efficient and uses only reasonable resources.

Quality With the guided strategy, we have the means to better select the content of the warehouse (via discovery) and its staleness (via refreshing).

Therefore, although we did implement a standard crawler, we are not using it anymore.

3. ARCHITECTURE

Before starting implementing the system, we evaluated publicly availably crawlers [22, 23]. We found that (at that time) available crawlers were inadequate for our purposes for several reasons. First, many of these crawlers were developed for Intranet crawling and did not scale to the web. But the main reasons for deciding to develop our own crawler were that (i) we wanted the crawler to be fully integrated with the rest of the system and in particular with the XML Loader, and (ii) we wanted to have complete control on the crawling policy (choice of pages to crawl). In this section, we first present the general architecture. We then briefly consider two aspects of acquisition.

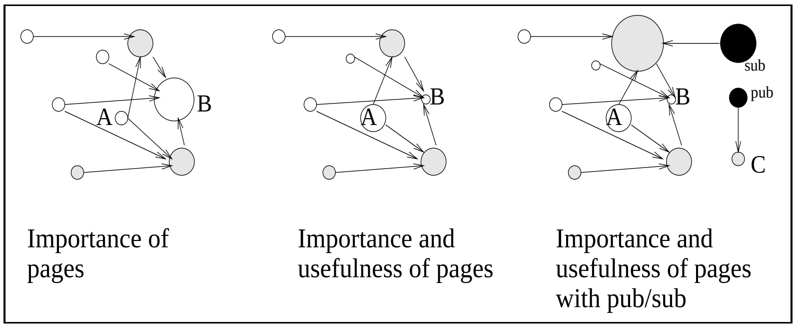

Figure 1: Importance vs. Usefulness

3.1 General architecture

A partial logical view of the Xyleme architecture focusing on acquisition is given in Figure 2. A description of the complete architecture of Xyleme is not within the scope of this paper. Its main modules are as follows:

- The *Web Interface* is the only module that accesses the web. To guarantee fast response time, this module only does very limited processing. More precisely, besides getting data from the web, its unique role is to parse the HTML documents to obtain the URLs in them.
- The *Page Scheduler* decides which pages should be read next using an *Estimator* that provides estimates of the importance of pages and their change frequencies.
- The *Loader* deals with XML pages, i.e., it parses, validates and loads XML documents in a native repository [15]. The versioning of some documents [17] as well as many other issues such as XML query processing [2] will be ignored here.
- The *Metadata Manager* manages metadata about XML and HTML documents that are needed in particular by the Page Scheduler. (See Section 3.3 for more details.) For XML documents, we need specific metadata such as the DTD or whether the document is versioned or not that are handled by a separate module (not shown here).

Xyleme is implemented on a cluster of standard PCs running Linux. The system is written in C++. The communication layer between the different modules is implemented using Corba [10]. All modules and in particular the Metadata Manager are distributed between several machines. This distribution is orthogonal to the functional interfaces provided by the different modules and guarantees scalability.

To conclude this section, we briefly present two modules, namely, Web Interface and Metadata Manager. The Loader, the Repository and the Publication/Subscription modules will not be discussed here. The Page Scheduler and the Estimator will be the topics of two separate sections.

3.2 Web Interface

The Web interface implements most of the features of standard search engines. We discuss some particularities.

This is the only module that fetches pages from the web. This is important in order to guarantee the respect of the *robot laws* [20]. For instance, one of these rules (*no rapid firing*) is that one client should not issue too rapidly many requests to the same site client[1]. When the Web Interface fetches an HTML page, it parses it (on the fly) to discover new URLs in it. The URL of the current page and the list of all its links are sent to the Metadata Manager in order to build the web Link Matrix (see Section 3.3). Similarly, when loading an XML document, the Loader sends a message to the Metadata Manager to update the Link Matrix.

Observe that the Web Interface gets loading requests from the Page Scheduler but also from the Loader. More precisely, when the Page Scheduler decides to load an XML page, the request is sent to the Web Interface. The Loader uses also the Web

[1] Observe the distinction between the possible behavior of a web client that is controlled by a user and one that is driven by a piece of software. A user would not be able to click on thousands of links in a few seconds whereas some software would. Well-known web servers encountered problems, in February 2000, due to a simultaneous firing of requests from robots that were using distribution in an uncontrolled manner.

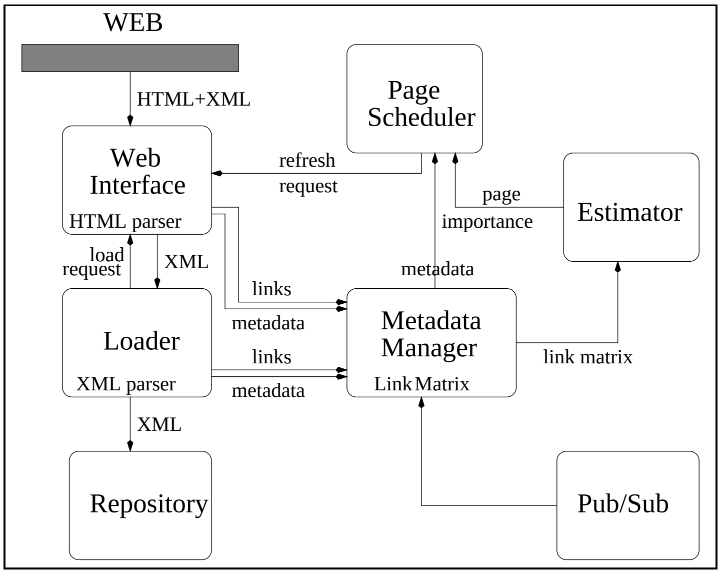

Figure 2: Simplified Functional Architecture for Acquisition/Maintenance

Interface to fetch the different pages, typically the document itself, its DTD and possibly other pages such as external entities. Thus the Loader has to send requests to the Web Interface. Clearly, a request by the Loader is given a higher priority since the Loader may be waiting for a particular resource. Indeed, the URL submission protocol between the Web Interface and the rest of Xyleme supports levels of priority (details omitted).

Metadata provided by the HTTP protocol such as the MIME type [12] together with additional filtering based on file suffixes such as ".jpg" or ".gif" are used to avoid reading pages that are neither XML nor HTML or that did not change.

A particularity of our crawling is that the loading of a particular page is decided by another module, namely, the Page Scheduler. The Web Interface has a buffer of pages that need to be loaded (from previous requests). This buffer is fixed to 100 000 pages in the experiments. We discovered that this is potentially a bottleneck. Pages from a particular site, say *www.popular.com*, may be blocked in the buffer because of the law against rapid firing. The buffer may end up being full of such pages. Thus when a too large number of pages of a site are in the buffer (so that we know they will not be serviced during a specific loading cycle – see further for the definition of these cycles), we simply reject the page. The page scheduler will reissue these requests and in the meantime, they do not slow down the entire process.

The Web Interface runs on a standard PC, with 600 Megahertz mono processor and uses 128 Megabytes of main memory. It fetches up to 4 million pages per day using 300 sockets for communication. If more throughput is needed, several crawlers can be used by distributing the URLs between several crawlers.

Main issues we encountered during the implementation are standard problems for crawlers such as: (i) threads may be blocked for minutes in DNS servers access, (ii) syntax errors are common on the web for HTML/XML files, robot.txt files (the files specifying robot exclusion), and (iii) huge files are sometimes encountered that are bigger that the size of main memory. For these reasons, although it is conceptually simple, it is not an easy task to develop such a module.

3.3 Metadata Management

The task of metadata management is distributed between several Metadata Manager. We first discuss the role of a single such process. We then consider how distribution is handled.

The Metadata Manager is in charge of storing and refreshing meta-informations for each URL. These informations are distributed into several data structures:

- Map-Table: This table maintains a one-to-one mapping between URLs and persistent identifiers that we call URL-ids. We currently use URL-ids of 5 bytes. More precisely, an URLid is a pair $(local, ident)$ where $local$ is an integer identifying the Xyleme machine in charge of this document and $ident$ an integer identifying the particular Url on that particular machine. Typically, the Xyleme machine is either a repository (for XML) or a Metadata Manager (for HTML). Within Xyleme, only the Metadata Manager knows the actual URL attached to a document. All other Xyleme modules deal only with URL-ids.

- Refresh-Table: This relation contains the data needed by the refresh module (see Section 4), and more precisely:
  - temporal data: the last date a change was detected by Xyleme for this page, the first and last date of access by Xyleme, the date of the last change as specified in the document header.
  - the document signature (MD5): as computed by the Web Interface when it last fetched the page. For HTML pages, this is the unique way of detecting changes since HTML pages are not stored in Xyleme.
  - the number of refreshes and number of accesses without refresh since a particular date.
  - the type of the document. For the moment, we use HTML, XML, DTD, ERROR, DEAD, where DEAD means that Xyleme believes this document disappeared from the web.
  - the importance of the page as computed by our Estimator (see Section 5).
  - other data needed, for instance, to gather statistics.
- Link-Matrix: This relation contains, for each page, the list of children[2] of the page, more precisely, the list of the URL-ids of the URLs referenced in that page. This information is used by the Estimator in Section 5.
- Status-Data: This relation contains some information about the temporary status of the loading of a document. This is needed for loading documents that use external entities.

It should be noted, that the work of the Metadata Manager is very data intensive. For each page that is fetched, it needs to update the metadata of the page. The metadata is stored on disk. This is needed because of its size and for recovery. However, this means that we need to perform one disk-write per page. The main difficulty is to update the Link-Matrix. The Link-Matrix uses URL-ids. The links we obtain from parsing a page deal with URLs. So, we need to transform URLs into URL-ids, i.e., we need an average of N random accesses to the Map-Table where N is the average number of links per page (N is of the order of 30). To be able to do that efficiently, we use an in-memory data structure for the Map-Table.

[2]To be precise, we store at most the 65534 first children for each page.

Distribution The Metadata Manager needs intense communications with the Web Interface. They both typically run on the same physical machine and are both designed to be distributed. To use several Web Interfaces, it suffices to split the domain of URLs (strings) into m were m is the number of Web Interfaces, e.g., using some random function from strings to $[1..m]$. One Metadata Manager can support the load of one Web Interface, so, in this case, we use also m Metadata Managers. We still have to perform extensive experiments using several Metadata Managers and Web Interfaces.

4. PAGE SCHEDULER

A most critical issue in Xyleme is to decide when to read (again) a page. This is the role of the Page Scheduler and the topic of the present section.

The quality of the warehouse critically depends on the pages it stores and their freshness. We thus want to minimize the delay between the arrival of an XML page on the web and its discovery by Xyleme. We also want to minimize the delay between the time an XML page changes on the web and the time this change is reflected in Xyleme. So, a main issue is the refreshing of XML pages and also of HTML pages that may lead to new XML pages.

We consider here the refresh of XML pages. The refresh of HTML pages will use similar techniques. The only difference is the definition of the importance of pages. So, its discussion is postponed to Section 5. Recall that this is only part of the larger picture that includes the discovery of new pages (parameters σ_X, σ_H, χ) that is also based on importance and was discussed in Section 2.

We articulate the refresh strategy as an optimization problem. Towards that goal, we use a cost function. (We present a simplified version of this function here.)

The goal of the Page Scheduler is to minimize the obsolescence of the Xyleme Repository under certain constraints. The cost function takes in account:

1. The page importance of the page. We will see in Section 5 how this is defined and computed.
2. The change frequency of a page. We will see in Section 6 how this is defined and computed.
3. The refresh cycle time that determines the shortest interval between two refreshes of the same page. In the experiments, this was fixed to 6 hours.
4. The crawler bandwidth, i.e., the average number of pages that the crawler can refresh in a time unit.

Due to space limitations, the justifications of formulas are not presented in this paper.

Obsolescence

A Xyleme document that is no longer identical to it's source on the web is obsolete. But there are different degrees of obsolescence: as time goes by, the web document can undergo further modifications, and the Xyleme copy while staying out of date, is getting more and more "obsolete". A version of a document that is one day old might be satisfactory to some user even if out of date, while one being a year old might be just unacceptable. We thus distinguish two components of obsolescence: the fact that a document is out of date (at least one update missed), and the fact that it aged (based on the number of updates missed).

As mentioned before, Xyleme does not have complete knowledge of the modifications of a web document. As in [8], we assume that updates to a web page follow a random Poisson process, that is the modifications to a page i are independent and happen at a given frequency λ_i. Certain pages follow a totally different modification pattern, e.g., some pages are updated at regular time intervals such as every Monday. This is easy to handle if such information has been provided to Xyleme by the Webmaster for that page. We thus assume that page updates follow Poisson distributions and we will see in Section 6 how λ_i is estimated for each page i. Observe that the average number of changes in a interval of length t is $\lambda_i t$.

To capture the cost of obsolescence, we need to choose for cost, a function of t and λ. According to the previous discussion, this function should take into account the fact that the page becomes out of date and that it ages (becomes "very" out of date). Roughly speaking, the first component increases until the page is very likely to be out-of-date while the second continues to grow while time goes on. Different applications may assign different weight to each. We selected for cost the function $(\lambda t)^\alpha$. The advantages of this function are two folds:

1. it reasonably approximates what we believe is a "normal" cost function taking into account both factors.
2. it facilitates the search for an optimal schedule.

Although it is conceivable to have α be depending on the document, we still lack the knowledge of the semantics of documents to do so. We therefore use a fixed value for α. The α parameter is used to adjust the two factors of obsolescence mentioned before. By appropriately choosing the α value, the system administrator has the means to influence the strategy, e.g., smaller α values put more emphasis on the out-of-date factor.

We also consider that important documents being out of date have greater (negative) impact than less important obsolete documents. Specifically, the cost of a document being obsolete is also proportional to its importance. Thus, we use the following function giving the *cost of a page i which has not been refreshed for time t*:

$$d_i(t) = w_i(\lambda_i t)^\alpha \tag{1}$$

where w_i is the estimated importance of the page, λ_i the estimated frequency of change of the page, and α is a fixed parameter, 0.95 in your experiments.

REMARK 4.1. *A first observation on the regularity of refresh turns to be very useful. Assume page i has a change frequency of λ_i. Suppose we decide to allocate a certain bandwidth to the page, e.g., m_i accesses every month, i.e., f_i accesses per second. Then one can show that it is optimal to refresh the page uniformly during that interval, i.e., refresh it every $1/f_i$ seconds. (Proof omitted.)* □

Assuming that we use an access frequency of f_i for page i for each i. The average cost of a page i:

$$c_i(f_i) = \frac{\int_0^{1/f_i} d_i(t)dt}{1/f_i} \tag{2}$$

Now, let N be the total number of pages in the repository. We define the *cost of obsolescence of the Xyleme repository* as the sum of the costs of individual pages, i.e.,

$$COST = \sum_{i=1}^{N} c_i(f_i) \quad (3)$$

This cost function represents the average obsolescence of the Xyleme repository. We want to minimize this cost under a constraint, the *bandwidth* of the crawler. Let G be the number of pages the crawler can refresh in a time-unit. This introduces a constraint on the the access frequencies of pages:

$$(\dagger) \quad \sum_{i=1}^{N} f_i - G = 0$$

Thus the problem is to compute $f_i, i = 1...N$, that minimize $COST$ under the constraint ($\dagger$). Using the Lagrange multiplier method, we get

$$f_i = \frac{G}{S} \sqrt[\alpha+1]{w_i \lambda^{\alpha}{}_i} \quad (4)$$

where

$$S = \sum_{i=1}^{N} \sqrt[\alpha+1]{w_i \lambda^{\alpha}{}_i} \quad (5)$$

Formula 4 is the basis of Xyleme Page Scheduler.

The way we compute f_i guarantees that, after a warm-up period, the Page Scheduler will come to an equilibrium point, where the number of pages that need to be crawled in any time-unit is very close to G.

Implementation

The Page Scheduler maintains the global variable S (5). For a page that has just been refreshed, we know the old λ_{old} value (the refresh frequency estimation we had before the refresh), and the new λ_{new} (the estimation done after the refresh). The value of S is updated as follows:

$$S_{new} = S_{old} - \sqrt[\alpha+1]{w\lambda_{old}{}^{\alpha}} + \sqrt[\alpha+1]{w\lambda_{new}{}^{\alpha}} \quad (6)$$

This way, we always know the current value for S.

Let T be the refresh cycle time. Recall that it fixed by the system administrator. The Page Scheduler does a complete scan over the set of pages once per *refresh cycle*. This explains why we forbid too small refresh cycles. When it visits page i (the metadata of page i), the Page Scheduler can easily detect whether page i should be refreshed.

Indeed, f_i is function of the value of S, the importance (ω_i) and the change frequency (λ_i) as shown in equation (4). Then using the last access time (t_i), the page must be refreshed if $(t_i - t_{now}) > 1/f_i$, i.e., if the time elapsed since the last refresh is greater than the refresh period of the page. In that case, the Page Scheduler sends a request to the Web Interface to read the page. The way we computed the access frequency for a page guarantees that we will access exactly[3] G pages per refresh cycle. As the refresh cycle we use is relatively large (6 hours in Xyleme), a scan over all pages per time unit is perfectly acceptable, i.e. is not too expensive.

5. PAGE IMPORTANCE

Page scheduling is guided by the importance of pages. In this section, we see how the importance is estimated for refreshing XML pages (Section 5.1). The same notion is used for the acquisition of new pages. We then consider the refreshing of HTML pages (Section 5.2). We also see how publication/subscription is introduced in this evaluation (Section 5.3). Finally, we describe some aspects of the implementation and possible developments.

5.1 Importance of XML pages

In this section, we recall the technique of [6] to define and compute the importance of pages. This will be used for XML pages.

High quality documents, that contain clear, accurate and useful information, are likely to have many links pointing to them, while low quality documents will get few or no links. Also, links coming from important pages (such as the Yahoo! homepage, for example) weight more than links coming from less important pages.

We first recall the standard technique used to estimate the importance of pages. It uses an *importance vector* $\mathcal{I}$. We use $[1..N]$ as identifiers for the pages. For a page i, let $out(i)$ the set of pages that i points to, and $outdeg(i)$ be the out-degree of i, i.e., $outdeg(i) = |out(i)|$. The vector $\mathcal{I}$ is given by:

$$\mathcal{I}(i) = \sum_{i \in out(j)} \frac{\mathcal{I}(j)}{outdeg(j)} \quad (7)$$

that equally distributes the importance of a page between its successors. Considering the set of equations (7) for every page, we get the following linear equation system:

$$\mathcal{I}_{N\times 1} = M_{N\times N} \times \mathcal{I}_{N\times 1} \quad (8)$$

$$\text{where} \quad M_{ij} = \begin{cases} \frac{1}{outdeg(j)} : i \in out(j) \\ 0 : otherwise \end{cases}$$

[3]Strictly speaking, since the repository changes while we are performing a refresh cycle, we will not refresh "exactly" G pages, but a number of pages very close to G.

The computation of page importance can be viewed equivalently as the resolution of the linear equation system $M \times \mathcal{I} - \mathcal{I} = 0$, as the computation of an eigenvector computation for M, or as a fixpoint computation for φ, where $\varphi(X) = M * X$. This solution can also be interpreted from the perspective of a random walk on a graph. Suppose a user is viewing web-page i and may follow any link on that page with equal probability. Note that $M_{ij} = \frac{1}{outdeg(j)}$ gives the probability she "walks" to j. Let us assume that $\mathcal{I}$ is appropriately normalized, i.e.,

$$\begin{array}{lll} \|\mathcal{I}\|_1 = 1 & \text{where} & \\ \|(v_1, ..., v_n)\|_1 & \text{is defined by} & v_1 + ... + v_n \end{array}$$

With this interpretation, the importance of a page gives the limit probability that a random walk leads to this page.

Let $\|(V_1, ..., V_n)\|_2$ be defined as $\sqrt{V_1^2 + ... + V_n^2}$. The $\mathcal{I}$ vector can be computed using an iterative fixpoint technique, by repeatedly applying M to any non-degenerate start vector:

$\mathcal{I}^0$ a uniform initial vector (all pages have the same importance) or the $\mathcal{I}$ vector from the previous computation

$\mathcal{I}^{i+1} = M \times \mathcal{I}^i$ the vector at the $(i+1)$th iteration

$\|\mathcal{I}^n - \mathcal{I}^{n-1}\|_2 < \epsilon$ the stop condition

The convergence of this algorithm is guaranteed if M is irreducible i.e., the links graph is strongly connected and aperiodic. The latter holds in practice for the web, while the former is true if we add a dampening factor c to the rank propagation:

$$\begin{array}{l} \mathcal{I} = (cM \times \mathcal{I}) + (1 - c) \times E \\ \text{where} \quad c \in]0, 1[\end{array} \tag{9}$$

The c dampening factor diminishes the propagation of $\mathcal{I}$ along long chains, thus diminishing the influence of far pages. In fact, the importance of a page propagated along n nodes will be multiplied by a factor of c^n. Also, a lower c value quickens the convergence of the algorithm. Typical c values are in $[0.7, 0.95]$. For example, the Page-Rank algorithm in [6] uses $c = 0.85$. It turns out that the choice of this factor has little impact on the importance of pages.

The E vector in the equation above is a *personalization vector* [5]. We have $E(i) \in]0, 1[$, $\|E\|_1 = 1$. Assume for now we use $E = \left[\frac{1}{N}\right]_{N \times 1}$. We will see how this is effectively fixed in Section 5.3.

One can easily verify that when M is stochastic, i.e. $\forall i, out(i) \neq \emptyset$, the $\mathcal{I}$ vector is also stochastic: $\|\mathcal{I}^0\|_1 = 1 \Rightarrow \forall i \|\mathcal{I}^i\|_1 = 1$. This nice normalization property of $\mathcal{I}$ is lost when there are nodes without successors, the intuition being that the $\mathcal{I}$ value of these nodes is lost from the system at each iteration. To recover this property, we introduce a normalization factor r such that:

$$\mathcal{I}^{i+1} = r \times (cM \times \mathcal{I}^i + (1 - c) \times E), \tag{10}$$

$$\text{where} \qquad r = \frac{\|\mathcal{I}^i\|_1}{\|cM \times \mathcal{I}^i + (1 - c) \times E_N} \tag{11}$$

Thus we have:

$$\|\mathcal{I}^{i+1}\|_1 = \|\mathcal{I}^i\|_1 \tag{12}$$

We compute $\mathcal{I}$ over the set of XML and HTML pages. Over the already read XML pages, this importance will guide the refreshing of these pages. Over not-yet-read XML and HTML pages, it will help us decide which pages to read next. We will use a different measure for refreshing HTML pages.

5.2 Usefulness for HTML pages

A variant of the algorithm is used for HTML pages. We use the following heuristics: an HTML page that does not point to XML (as we know so far) is less likely to lead (in the future) to new XML pages. Clearly, we are interested also in those pages that point to pages that point to much XML, and so on. That is, a *useful* HTML page points *directly or indirectly* to a lot of useful XML pages. So the problem is not that different. However, while the *importance* for XML pages moves *forward* on links, the *usefulness* for HTML pages moves *backward* on the graph links.

Thus, in order to compute the *usefulness*, we use the same algorithm, but with different *start condition* and on the inverse links graph. The matrix corresponding to this graph is the transposed of the initial graph matrix:

$$M_{ij}^t = \left\{ \begin{array}{ccl} \frac{1}{outdeg(i)} & : & j \in out(i) \\ 0 & : & otherwise \end{array} \right.$$

This leads to compute the following fixpoint:

$\mathcal{J}^0$ the initial vector

$\mathcal{J}^{i+1} = cM^t \times \mathcal{J}^i + (1 - c)E'$

$$E'_i = \left\{ \begin{array}{ccl} \mathcal{I}_i & : & i \text{ corresponds to an XML page} \\ 0 & : & \text{otherwise} \end{array} \right.$$

$\|\mathcal{J}^n - \mathcal{J}^{n-1}\|_2 < \epsilon$ the stop condition

That is, we compute the fixpoint over the inverse graph using E' as personalization vector, which biases the algorithm towards XML pages (taking into account their importance), and with an initial $\mathcal{J}$ value of 0 for HTML pages (also given by E').

5.3 Handling Publication/Subscription

The same framework can be used to take publications/subscriptions of users into account. To capture publication/subscription, we introduce virtual pages and modify the personalization vector. Consider a set of pages a particular user is interested in. We consider that the image of the web we have is extended by a (virtual) page that contains the list of references to these pages. This page cannot be found on the web (it does not exist). However, we use it to capture the interests of users using the personalization vector. Therefore, we can bias the importance of the pages it refers to.

The choice of the importance of such pub/sub pages depends on the application, i.e., the importance one wants to give to pub/sub in general and to this pub/sub in particular. It is difficult to give an absolute value without some global knowledge of the repository that is typically changing, such as the average importance and the standard deviation for importance. We decided to use a relative importance, e.g., a particular pub/sub page may give enough importance to the pages it points to, to promote them to the top 20% most important pages of the repository.

5.4 Implementation

In this section, we briefly discuss some aspects of our efficient and scalable implementation that can deal, on a standard PC with billions of pages of the web. The core of the technique consists in an efficient implementation of sparse-matrix vector multiplication, see [21].

The Naive Algorithm First, observe that the Link Matrix is sparse. We use, for each node i, the list of its successors ($out(i)$). This is also the original representation of the links graph that we get when crawling the web: every page contains the list of its successors. As the matrix metaphor is useful in understanding the algorithm, we will continue to refer to this links representation as the *link matrix*.

The main lines of one iteration of computation of importance are given by:

for j=1 ... N, $dest\mathcal{I}_j = 0$
for i=1 ... N, for each j $\in$ out(i),
$dest\mathcal{I}_i = dest\mathcal{I}_j + \frac{source\mathcal{I}_i}{outdeg(i)}$
$source\mathcal{I} = dest\mathcal{I}$

To compute on the inverse graph (to compute the *usefulness* of HTML pages), we can use the same representation of the links graph with a different algorithm:

for i=1 ... N, $dest\mathcal{I}'_i = 0$
for j=1 ... N, for each j $\in$ out(i),
$dest\mathcal{I}'_i = dest\mathcal{I}'_i + \frac{source\mathcal{I}'_j}{outdeg(j)}$
$source\mathcal{I}' = dest\mathcal{I}'$

In our implementation, we adopted the *Matrix Slicing* technique [14] that we recall briefly. For large N, the link matrix does not fit in (a PC) main memory. It is stored on disk and the trick is to avoid to have to perform random access to this data.

For the *importance* algorithm, we have sequential access over the $source\mathcal{I}$ and the link matrix, but random access on $dest\mathcal{I}$. If the $dest\mathcal{I}$ vector does not fit in memory, it is cut in S slices, each of these small enough to fit in memory. So each slice has β elements ($\beta = \frac{N}{S}$). The $dest\mathcal{I}$ slice s, noted $dest\mathcal{I}_{|s}$, will contain $\mathcal{I}$ values for nodes between $s\beta$ and $(s+1)\beta - 1$. The link matrix is also cut "vertically" in slices, such that each slice of the matrix only references elements of the corresponding $dest\mathcal{I}$ slice. We process the matrix slices one by one. This way, when processing a given slice, we will have random access only on the corresponding $dest\mathcal{I}$ slice that fits in main memory. When we finish computing a given $dest\mathcal{I}$ slice, it is written on disk, and the next slice is prepared in memory.

For the *usefulness* algorithm, we have sequential access over the $dest\mathcal{I}'$ vector, and random access on $source\mathcal{I}'$. So, in order to apply the slicing technique to *inverse I*, we have to slice the $source\mathcal{I}'$. Again, we are able to use the same matrix slices for both algorithms.

Gauss-Seidel Iterations

To compute the fixpoint, we use a technique is similar to the *Gauss-Seidel algorithm* [11] that allows for quicker convergence and also diminishes the disk space requirements.

As we are heading toward a fixpoint, we expect $dest\mathcal{I}$ to be closer to the fixpoint than $source\mathcal{I}$. So, we expect that a newly computed slice of $dest\mathcal{I}$ is closer to the fixpoint than the corresponding $source\mathcal{I}$ slice. When processing a slice s, we have available all the already computed slices $dest\mathcal{I}|t, t < s$. In the subsequent computations, we will use these slices instead of the corresponding $source\mathcal{I}$ parts. Concerning the implementation, we no longer use two vectors, $source\mathcal{I}$ and $dest\mathcal{I}$, but only one, let us call it $\mathcal{I}$. The slice being computed is stored in the main memory; when the computation is done, it is stored in the $\mathcal{I}$ vector, overwriting the old slice, that is no longer needed. The subsequent computations will use the newly computed slices when they are available.

This guarantees a quicker convergence since we compute each slice starting from a better approximation of the fixpoint. Also, we only need one $\mathcal{I}$ vector instead of *source*$\mathcal{I}$ and *dest*$\mathcal{I}$.

Combined computation

We can obtain some (limited) speed-up by computing *importance* and *usefulness* at the same time. The bottleneck of both algorithms is clearly disk I/O since each iteration does a complete scan of the link matrix. By computing *importance* and *usefulness* independently, we perform one matrix scan for each iteration for each of them. We can mix the two and perform one iteration of each while doing a single matrix scan. Of course, we need to store in the main memory the $\mathcal{I}$ slices for both of algorithms; so the memory space available to a slice is halved, and we need twice as much slices when performing the mixed computation. Nevertheless, considering this tread-off, is comes out that it is more efficient to save a matrix scan at the expense of doubling the number of slices.

A difficulty is that the computation of *usefulness* uses the result of *importance*. One can start from the importance as computed by the previous run of the algorithms, then use more precise estimates of importance as they become available.

This technique can be generalized to any computation of multiple independent importance vectors based on the same link matrix.

Benchmark We next present some runtime measurements for the developed algorithms. For the benchmarks we used generated random link graphs, parameterized on the average number of links per node.

We used an Intel Pentium II PC, with 128 Megabytes RAM and a standard IDE disk, running Linux 2.0.36. For the measures, we used randomly generated Link graphs, parameterized on the average number of links per node. In the measures given here, we used graphs with an average of 10 outgoing links per page which is roughly the average on the web. We ran the page ranking algorithm for XML (direct) and HTML (inverse) as well as the algorithm that combines both. Table 1 gives the time used for each iteration.

In the experiments, we get reasonably close to the fixpoint after ten iterations. For a random matrix, this number is a logarithm of the number of pages. So for 10^9 pages, 12 iterations would roughly suffice. We hope that the convergence will be as fast for real data. The main result is that time grows almost linearly in the number of pages thanks to the slicing technique. The use of slicing also explains why the penalty with a much smaller RAM is not so high. The gain of the mixed method (not shown here) is modest. The mixed method turns out to pay really only when the size of the available RAM is much larger.

5.5 Further Development

The algorithm we implemented is designed to run on a single machine. Anyway, as the number of web pages grows very fast, such a solution might soon prove inadequate. We briefly consider next two alternatives, distributed and incremental algorithms:

Distributed Computation

The splitting between slices suggest the following algorithm. We use a number of slices equal to the number of participating machines. In the algorithm initialization phase, each machine is assigned a slice to process, and gets the corresponding slice of the matrix. The computation of each slice of $\mathcal{I}^{i+1}$ takes place in parallel, as these computations are independent. More precisely, in iteration i, each machine C_k computes $\mathcal{I}^{i+1}_k$ (the k^{th} slice of $\mathcal{I}^{i+1}$) using $\mathcal{I}^i$ and M_k (the k^{th} slice of M). Clearly, this algorithm would result in intense network communications since $\mathcal{I}^i$ is distributed over all machines. However, note that $\mathcal{I}^i$ is much smaller smaller than M, so disk I/O still dominates.

One can have all the machine compute synchronously, i.e., all the stations step to iteration $i+1$ at the same moment, when all the slices of $\mathcal{I}^{i+1}$ have been computed. It is more efficient to let each machine compute at its own speed its new $\mathcal{I}_k$ slice obtaining from the other stations l the latest version of $\mathcal{I}_l$ they might have already computed. This allows to converge faster and saves the cost of synchronization.

Incremental Computation

An alternative is an incremental algorithm. Such an algorithm will gradually compute the importance/usefulness, as the link graph is explored and updated (by a web crawler, for example). We are currently considering such an algorithm.

6. ESTIMATING CHANGE FREQUENCY (IN BRIEF)

		80MB RAM	40MB RAM
20 Million Pages	Direct	178s	233s
	Inverse	149s	242s
40 Million Pages	Direct	515s	670s
	Inverse	551s	772s

Table 1: Measures of page ranking

In this section, we briefly mention the estimation of change frequency.

Xyleme does not know when a web page changes.[4] Thus, it is difficult to estimate the change frequency. As shown in [8], the changes of Web pages are well-described by a random Poisson process. So, by default, i.e., in absence of additional information, we model the page update by a Poisson process. The only parameter in a Poisson process is the average frequency λ of the random event. Every page is characterized by such a λ. The λ of a page is not known. This section describes how it is estimated.

The frequency (λ) for a given page can be estimated from a number of observations:

Visible changes When Xyleme refreshes the page, it can see if the page has changed since the last access (typically by comparing some signatures). An issue is that we only know if a page has changed or not between two accesses. We ignore how many times it did.

Last date of change Typically, Web servers provide, together with the requested page, the date when the page was last modified, called *last date of change*. This is a more precise information and it will be preferred when available.

For the estimators, we use improvements over [9] that are presented in [19].

7. CONCLUSION

All modules discussed in this paper have been implemented.

We have spent the previous few months tuning the crawler and recently started crawling the web for obtaining large volumes of XML documents. We need to continue testing the system. We started gathering a number of statistics such as the ratio between XML and HTML pages, the average size of XML pages, the number of tags, the depth of DTD, the distribution between DTDs, the change frequency of XML pages, the average quantity of changes, etc. We plan also to gather system measures such as network bandwidth actually used by the crawler or the weight of Corba communications. These are measures that we need to design and tune the general system.

A number of improvements need to be considered. In particular, we would like to be able to introduce some semantic data analysis of documents and DTDs to guide more accurately the search. We mentioned distributed and incremental versions of the computation of importance/usefulness that need to be implemented and compared to the current algorithm we use. The change frequency estimator needs to be improved.

Acknowledgments We would like to thank a number of people for participating in meetings where we discussed this work, and in particular, Zhen Liu, Luc Segoufin, Guy Ferran, Bruno Koechlin, Genevieve Jomier, François Llirbat, Sophie Cluet and V. Vianu. Finally we thank the AFS Team at INRIA Rocquencourt for theirs valuable help in the system architecture.

8. REFERENCES

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publisher, October 1999.

[2] Vincent Aguiléra, Sophie Cluet, Pierangelo Veltri, Dan Vodislav, and Fanny Wattez. Querying xml documents in xyleme. In *to appear in the proceedings of the ACM-SIGIR 2000 Workshop on XML and Information Retrieval*, Athens, Greece, july 2000.

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

[4] The internet archive: Building an 'internet library'. `http://www.archive.org/`.

[5] Sergey Brin, Rajeev Motwani, Lawrence Page, and Terry Winograd. What can you do with a web in your pocket. *Bulletin of the*

[4] A web master may notify Xyleme of a change via the publication API but this is the easy case ignored here.

IEEE Computer Society Technical Committe on Data Engineering, 1998.

[6] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *7th WWW*, 1998.

[7] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused crawling: A new approach to topic-specific web resource discovery. *Computer Networks*, 31(11-16):1623–1640, 1999.

[8] Junghoo Cho and Hector Garcia-Molina. The Evolution of the Web and Implications for an Incremental Crawler. Technical report, Stanford University, 1999. http://dbpubs.stanford.edu:8090/pub/1999-22/.

[9] Junghoo Cho and Hector Garcia-Molina. Estimating Frequency of Change. Technical report, Stanford University, 2000. http://dbpubs.stanford.edu:8090/pub/1999-22/.

[10] Common Object Request Broker Architecture (O.M.G). `http://www.corba.org/`.

[11] M. Gondran and M. Minoux. *Graphes et Algorithmes*. Eyrolles, 1995.

[12] Network Working Group. MIME (Multipurpose Internet Mail Extensions part one : Mechanism for specifying and describing the format of internet message bodies. RFC 1521.

[13] Network Working Group. XML Media Types. RFC 2376.

[14] T. H. Haveliwala. Efficient computation of pagerank, 1999. Stanford Database Group.

[15] Carl-Christian Kanne and Guido Moerkotte. Efficient storage of XML data. Technical Report 8/99, University of Mannheim, 1999. available at `http://pi3.informatik.uni-mannheim.de/publications/techrep899.ps`.

[16] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. In *Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[17] Amélie Marian, Serge Abiteboul, and Laurent Mignet. Change-centric management of versions in an XML warehouse, October 2000. proceedings of Base de Données Avancées conference.

[18] B. Nguyen, S. Abiteboul, G. Cobena, and L. Mignet. Query subscription in an xml webhouse, 2000. DELOS - to appear.

[19] M. Preda. Data acquisition for an xml warehouse. Technical report, Univ. Paris VII, 2000.

[20] The Robots Exclusion Protocol. can be obtained from `http://info.webcrawler.com/`.

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

[22] ht://Dig WWW Search Engine Software.

[23] Wget. `http://www.gnu.org/software/wget/wget.html`.

[24] World Wide Web Consortium. eXtensible Markup Language (XML) 1.0. `http://www.w3.org/TR/REC-xml/`.

[25] World Wide Web Consortium. HyperText Markup Language (HTML) 4.1. `http://www.w3.org/MarkUp/`.

[26] J. Widom. Research problems in data warehousing. In *CIKM '95 (invited paper)*, 1995.

[27] Xyleme Project. `http://www-rocq.inria.fr/verso/Xyleme.html`.