

# Acquiring XML pages for a WebHouse

Laurent Mignet<sup>†</sup>      Mihai Preda<sup>†</sup>      Serge Abiteboul<sup>†</sup>  
Sébastien Ailleret<sup>†</sup>      Bernd Amann<sup>‡</sup>      Amélie Marian<sup>†</sup>

July 23, 2000

## Abstract

Xyleme is a dynamic warehouse for the XML data of the web supporting change control and data integration. Major issues are the acquisition of XML data and keeping data up to date with the web as best as possible. This is the topic of the present paper.

## Résumé

Xyleme est un entrepôt pour toutes les données XML de la toile offrant des fonctionnalités de contrôle des changements et d'intégration de données. Des problèmes importants soulevés sont l'acquisition et la maintenance à jour de ces données. C'est le sujet du présent article.

**Keywords:** XML, Warehouse, WebHouse, Refresh Policy, Page Rank, Change Control.

## 1 Introduction

The current development of the web and the generalization of XML [21] technology provides a major opportunity for changing the face of the web in a fundamental way. This work is part of the Xyleme project [25] aiming at the development of a *dynamic XML warehouse for the web*. In the present paper, we consider the problem of acquiring XML data from the web, i.e., from a world still dominated by HTML [22], and keeping it up to date with the web as best as possible.

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 [1].

We are studying and building Xyleme, a *dynamic World Wide XML warehouse* capable of storing all the XML data available on the planet. Therefore the problems we consider are typical warehousing problems [23]. First, we need to obtain as many as possible XML

---

<sup>†</sup>I.N.R.I.A. VERSO, BP 105 78153 Le Chesnay Cedex, France, Email: [firstname.lastname@inria.fr](mailto:firstname.lastname@inria.fr)

<sup>‡</sup>Vertigo - C.N.A.M 292, rue Saint Martin 75141 Paris Cedex 03, France, Email: [amann@cnam.fr](mailto:amann@cnam.fr)

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 very essential and particular to our context. Clearly, given that network, storage and processing resources are always limited, we will have to be satisfied with getting only a part of all interesting XML data and maintaining the retrieved data up to date with some level of staleness.

A major issue in this context is *scalability*. XML is still in its infancy and finding as much XML data as possible as well as minimizing the portion of stale data in our warehouse is not a very big challenge yet<sup>1</sup>, but we believe that XML will grow up and these problems will soon become important.

**Crawling** Web data can typically be obtained by two complementary technologies called *pull* and *push*. Pull involves crawling the web and fetching data. Push implies an active participation of the web server. More precisely, a site master aware of Xyleme publishes some available XML resources, i.e., actively warns our system of the existence of this data. Crawling is the more standard technique used by HTML search engines such as Alta Vista [3], Yahoo! [26] or Google [12]. Starting from some seeds, the crawler follows links to discover more and more pages. Crawling is a conceptually easy task. The system gets new HTML or XML pages by recursively following existing links in already discovered pages. Observe that although we are primarily interested in the XML web, i.e. the portion of the web corresponding to XML pages, HTML pages have to be considered as well in our crawling process, since they might contain links to XML document. This frontier between the XML web and the classical HTML web leads to interesting issues in the refresh policy and page ranking that we study.

**Refreshing XML pages** The problem of refreshing XML pages in order to keep the XML warehouse up to date is a complex task. As mentioned before, resources are limited and our system can only read a certain amount of pages per time unit, say per day. Since the pool of existing pages is much wider, the naive strategy of refreshing *all* pages periodically may result in refresh cycles of several weeks and staleness for rapidly evolving data. Indeed, this is the situation for some of the best HTML search engines that visit pages so rarely that their index is largely obsolete with negative impacts on the search quality. The following observations allow to improve the quality of the warehouse under fixed resources:

1. we should not waste resources to often read unimportant pages;
2. we should not waste resources to often read a page that changes rarely or never (an archive or a forgotten page); and
3. 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 (loading capabilities), for some cost function taking into account the importance and change rate of pages.

---

<sup>1</sup>The number of accessible XML pages is marginal for the moment, less that 1% of the number of HTML pages. However, we believe that many companies already switched to XML and that the number of XML pages will explode as soon as XML will be supported by most installed browsers. Within a year?

We shall see that estimating change rates of pages is not very easy because of our limited knowledge of the changes that occur for each page. Determining the importance of a page is also far from trivial. To do it, we use a page ranking technique in the style of Google [6, 13]. Note that page ranking is a useful notion in itself since it also allows to rank pages returned by a query as done in Google for HTML pages. Indeed, we use here a page ranking à la Google. The intuition is that if an important page *points to* a page *p*, it transfers to *p* some of its importance. To compute the Xyleme-importance ranking of pages, we therefore run a fixpoint computation on all the HTML and XML pages the system knows. We need to consider both since an XML page may get its importance from the fact that it is referenced by important HTML pages. We then use the importance of pages to guide refreshing and to order documents in query answers.

**Refreshing HTML pages** Observe that an HTML page may evolve in time and lead to new XML pages when crawled again. So, to be sure not to ignore important portions of the XML web, we need to read HTML pages more than once, i.e., refresh them too. However, the Xyleme-importance as obtained by page ranking à la Google is not the good criteria to guide the refreshing of HTML pages. An HTML page may be ranked very high for importance and not lead to any XML page, and thus be useless from a Xyleme viewpoint. A better estimation of the importance of an HTML page *p* for refreshing can be obtained by transferring to *p* the importance of the XML pages that can be *reached from p*. So, for HTML pages, we use a different ranking algorithm (related to likelihood to bring to XML pages) to guide refreshing.

**Novelties and contributions** Some of the ideas described here are not new. However, we believe that the entire work is worth describing since (i) the general framework is new and (ii) the understanding of some background (such as Google page ranking algorithm) is needed to understand our contributions. The originality of the crawler is mostly in its integration with the remaining of the system and in particular the loader (for XML pages), the Metadata Manager (for XML/HTML pages) and with the Acquisition Module (see Figure 1). The page ranking we use is more original in its handling of XML pages and in novel optimization techniques we developed. The general strategy for controlling the acquisition of new pages is new.

To conclude this section, we want to stress that a major issue here is the size of the web. Many academic and industrial projects are concerned with warehousing XML data (e.g., Google found more than 10 000 pages for the search “xml warehouse” in July 2000). Clearly, these projects share objectives with ours. However, most of the problems we consider here would be relatively easy at the level of an Intranet. They are not in the context of the web. The number of HTML pages on the web is above 1 000 millions and it should be expected that a reasonable portion of it will soon be XML. To illustrate the scaling problems, page ranking leads to handling the Link matrix of the web ( $B(i, j) = 1$  if page *i* points to page *j*). Stored as a matrix of bits, it would be of the order of  $10^{18}$  bits or 1 000 000 Terabytes ! Hopefully, it is sparse but still requires the order 10 Gigabytes for 200 millions pages. In [7, 15], experiments are achieved with this matrix in a main memory of 12 Gigabytes. We do our computation on a standard PC with reasonable memory and disks.

The Xyleme project is functioning as an open, loosely coupled network of researchers.

The Verso Group at INRIA was at the origin of the project. Groups in Mannheim U., U. Paris XI (Orsay) and CNAM-Paris rapidly joined as well as individual researchers from other places.

The paper is organized as follows. The architecture is discussed in Section 2. The crawler is considered in Section 3, metadata management in Section 4, page ranking in Section 5 and the refresh policy in Section 6. In a last section, we present the status of the work and discuss future directions of research.

## 2 Architecture

A partial logical view of the Xyleme architecture focusing on acquisition is given in Figure 1. A description of the complete architecture of Xyleme is not within the scope of this paper. The *Crawler*, called Xyro (for Xyleme robot), is the only module that accesses the web, i.e., it serves as the unique interface to 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 it obtains and extracts the URLs in them. The *Loader* module deals with XML pages, i.e., it parses, validates and loads XML documents in the repository developed at Mannheim University [14]. (The versioning of some documents [16] 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 for page ranking and acquisition. (See Section 4 for more details.)

For XML documents, we need some more metadata (concerning information such as the DTD or whether the document is versioned or not) that are handled by other modules not described in this paper. The *Page Ranking* module implements the page ranking algorithm and returns its result to the *Refresh Module*. Observe that the crawler acting as a web interface gets loading requests from the XML loader (e.g. for loading XML entities and DTDs) and from the acquisition module (for refreshing XML and HTML documents).

Xyleme is implemented on a cluster of standard PCs running Linux. The entire system is written in C++. The communication layer between the different modules is implemented by 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 some degree of scalability.

## 3 Web Interface and Crawler

Before implementing our own crawler, called Xyro, we evaluated publicly available crawlers [19, 20]. 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 for the web (e.g., non respect of robot exclusion files). 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, the Metadata Manager and the Acquisition Module and (ii) we wanted to have complete control on the crawling policy (e.g., depth of crawled portion on some specific site and selective crawling by visiting more interesting sites first). In this section, we briefly describe the present status of the

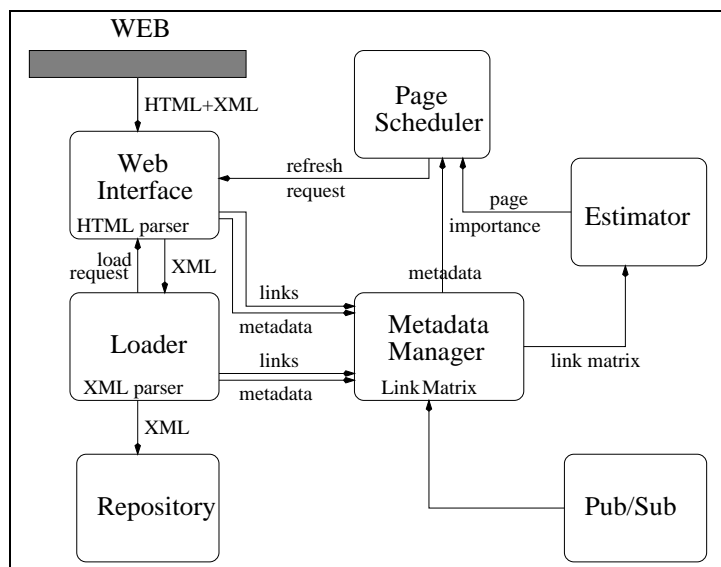


Figure 1: Simplified Logical Acquisition Architecture

crawler.

The aim of Xyro is to crawl the whole web in order to find XML. As a side effect, Xyro plays the role of an abstract web interface and is the only module that is allowed to fetch pages from the web in order to guarantee important rules that have to be respected by any software accessing a web server, i.e., some standard robot laws [18]. For instance, one of these rules is that one client should not issue too rapidly many requests to the same site client<sup>2</sup>.

Starting from a set of pages (seeds), Xyro follows links recursively to discover new pages. More precisely, the behavior of Xyro w.r.t. HTML and XML pages is the following:

1. After an HTML page is fetched, it is parsed (this parsing is done on the fly during fetching) and all URLs contained in the page are added to the FIFO of URLs to be fetched. The URL of the current page and the list of all its links are sent to the Metadata Module in order to maintain the web Link matrix (see Section 4).
2. XML pages are sent to the Xyleme Loader. XML parsing is under the responsibility of the loader module, which thereby discovers new pages to be fetched.

Metadata provided by the HTTP protocol such as mime type 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.

Note that since Xyro does not parse XML documents, Xyro cannot follow links in XML documents [24] by itself. Therefore, other modules such as the loader module can submit

<sup>2</sup>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, due to a simultaneous firing of requests from robots that were using distribution in an uncontrolled manner.

URLs to be fetched to Xyro. This functionality is also useful for refreshing pages already stored in the repository. If the acquisition module decides to refresh a particular page, it sends a request to Xyro to that effect. The URL submission protocol between Xyro and the rest of Xyleme supports five levels of priority (details omitted). Priority is essential because the completion of some tasks may depend critically on the loading of a document. For instance, while loading an XML document, one loaders may need other documents such as a DTD and external entities. These loading subtasks must be given high priority since this particular loader may be blocked until they are performed.

To support maximal throughput during crawling, Xyro has to check very rapidly if a given URL  $u$  has already been fetched or not. This check is implemented using a hash function  $h$  over URLs and a table  $T$  such that  $T(h(u))$  is 1 if  $u$  has already been fetched. We use a hash table reasonably bigger than the number of URLs we expect to fetch, so that we can assume without a too large chance of error that for some  $u$ , if  $T(h(u)) = 1$  then  $u$  has already been read. Errors will occur because the URL of a page may hash to the same integer as that of a page Xyleme already read. Such errors are acceptable since even a page first wrongly omitted by Xyro will eventually be read later when the Refresh Module decides to order so.

Xyro is already running. It works on a standard PC with 600 Megahertz mono processor and uses 128 Mo of main memory. It fetches about 2.5 million pages per day using 400 sockets for communication and obeys the robots standards such as robots exclusion or preventing rapid fire on a web site. Main issues we encountered during implementation are standard problems for crawlers such as: (i) threads may block for minutes in DNS servers access, (ii) erroneous syntax is common on the web for HTML/XML files, robot.text files (the files specifying robot exclusion), and (iii) huge files are sometimes encountered that are bigger than the size of main memory. For these reasons and others, it is not an easy task to develop a robust crawler.

In the first version of Xyro, the discovery of new pages was basically obtained (as done by standard search engines to our knowledge) by following the links to new pages up to a certain depth. We now have an alternative, more refined way of “choosing” the pages to read that will be described in Section 6.

## 4 Metadata Management

The Metadata manager is in charge of storing and refreshing meta-informations for each URL. This informations are distributed into several logical modules:

- Map-Data: This module is in charge of maintaining a one-to-one mapping between URLs and integers that are persistent identifiers that we call URL-Ids. Within Xyleme, only the Metadata Manager knows the actual URL attached to a document. Other Xyleme modules deal only with URL-Ids. (The mapping from URL-Ids to URLs is stored in a table. The inverse mapping is implemented using hashing.)
- Refresh-Data: This module is in charge of managing the data needed by the refresh module (see Section 6), and more precisely:

- some dates: the last date a change was detected by Xyleme, 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 Xyro 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.
  - number of refreshes and number of accesses without refresh since a last *base line*.
  - 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 the page ranking algorithm (see Section 5).
- Link-Data: This module maintains, for each page, the list of children<sup>3</sup> of the page, i.e., the list of URLs referenced in that page. This will be the basis for page ranking in Section 5.
  - Status-Data: This module maintains some information about the status of the loading of a document. This is needed for loading documents that use external entities.

The Metadata Manager is implemented. It stores data directly in the file system. It should be noted, that the work of the metadata manager is very data intensive. For each URL processed by Xyro, it needs to perform some random access to the store to update the information about that URL and in particular, its list of children. For the moment, we use a single metadata manager and this is slowing down the crawler by a factor of at least 50%. The metadata manager is designed to be distributed. We still have to experiment using several metadata managers to support the throughput of a single crawler without slowing it down.

## 5 Page Ranking

In this section, we consider the various aspects of page ranking. For ranking globally XML pages we use the page ranking algorithm of [6] on the set of all XML and HTML pages. We recall this algorithm in Section 5.1. For ranking HTML pages we use a novel algorithm that we present in Section 5.2. We describe the status of the implementation together with some experiments in Section 5.3 and possible developments in a last section.

### 5.1 Page ranking

To rank pages, our algorithm uses the web link structure to compute the importance of pages. This is described next.

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,

---

<sup>3</sup>To be precise, we store at most the 65534 first children for each page.

links coming from important pages (such as the Yahoo! homepage, for example) value more than links coming from less important pages. So, the connectivity or pattern of linkages between pages does contain a lot of implicit information about the relative importance of pages. The algorithm extracts (makes explicit) this information in the form of the PageRank vector, assigning to each page a value that correlates with an informal notion of *importance*.

For a page  $i$ , we note with  $Out(i)$  the set of pages pointed to by  $i$ , and  $OutDeg(i)$ , the out-degree of  $i$  ( $OutDeg(i) = |Out(i)|$ ). Let  $N$  be the total number of pages.

We define the PageRank for a page  $i$  as:

$$PR(i) = \sum_{j \in Out(j)} \frac{PR(j)}{OutDeg(j)} \quad (1)$$

We see that the importance of a page is equally distributed between all its successors. If we consider the set of equations (1) for every page, we get the following linear equation system:

$$PR_{N \times 1} = M_{N \times N} \times PR_{N \times 1} \quad \text{where} \quad M_{ij} = \begin{cases} \frac{1}{OutDeg(j)} & i \in Out(j) \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

We can (equivalently) view the computation of the PageRank vector either as an eigenvector computation for  $M$ , as the resolution of the linear equation system  $(M - I) * PR = 0$  or as a fixpoint computation for  $f$ , where  $f(X) = M * X$ .

We can also interpret the algorithm from the perspective of a random walk on a graph. We suppose that when a user is viewing a web-page, she can follow any of the links on that page with equal probability. Thus,  $M_{ij} = \frac{1}{OutDeg(j)}$  when there is a link from  $j$  to  $i$  and  $M_{ij} = 0$  if not, gives us the probability to walk from page  $j$  to page  $i$ . We also have:

$$\forall j, Out(j) \neq \emptyset \Rightarrow \sum_i M_{ij} = 1$$

With this interpretation, the PageRank vector will give the limit probability that the random walk will be at that node, and we have  $\|PR\|_1 = 1$ .

We compute the  $PR$  vector seen above using an iterative fixpoint technique, by repeatedly applying  $M$  to any non-degenerate start vector. We note  $PR^0$  the initial vector, and  $PR^i$  the vector after the iteration  $i$ .

$$PR^{i+1} = M \times PR^i \quad (3)$$

We iterate until a given *stop condition* is satisfied, for example

$$\|PR^n - PR^{n-1}\|_2 < \epsilon \quad (4)$$

The resulting  $PR = PR^n$  is an approximation of the dominant eigenvector of  $M$ . The convergence of this algorithm is guaranteed if  $M$  is irreducible (i.e., the Link 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.

$$PR = cM \times PR + (1 - c) \times E_{N \times 1} \quad \text{where} \quad c \in ]0, 1[ \quad (5)$$

The  $c$  dampening factor diminishes the propagation of PR along long chains, thus diminishing the influence of far pages. In fact, the PR 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 chosen within  $[0.7, 0.95]$ . For example, the page ranking algorithm employed by the *Google* [6] search engine uses  $c = 0.85$ .

The  $E$  vector in the equation above is a *personalization vector* [5]. We have  $E(i) \in ]0, 1[$ ,  $\|E\|_1 = 1$ . If we do not need personalization, we use  $E = \left[\frac{1}{N}\right]_{N \times 1}$ .

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

$$PR^{i+1} = r \times (cM \times PR^i + (1 - c) \times E), \quad (6)$$

$$\text{where } r = \frac{\|PR^i\|_1}{\|cM \times PR^i + (1 - c) \times E\|_1} \quad (7)$$

Thus we have:

$$\|PR^{i+1}\|_1 = \|PR^i\|_1 \quad (8)$$

We will see this in more detail when discussing the implementation of the algorithm (see Section 5.3).

## 5.2 Xyleme page ranking

Xyleme, as a XML repository, is mainly concerned with XML files. Nevertheless, we will have links information for both HTML and XML files. A page ranking computation is done on the whole graph (HTML and XML pages) in order to determine the importance of the XML pages. That is, we only need the resulting  $PR$  values for XML pages,  $PR_{XML}$ , but we have to do the computation for all the pages. The  $PR_{XML}$  will be used by the Acquisition Module of Xyleme when determining which XML pages will be crawled, based on the principle that more important XML pages (higher  $PR$ ) will be refreshed more often.

A different flavor of the page ranking algorithm is used to give a notion of *interest* for HTML pages. Xyleme is interested by XML pages. The only use it makes of HTML pages, is to use them as pointers to new XML (and to construct the Link matrix, of course). So, we are interested by those HTML pages that will lead us to new XML. We use this heuristics: an HTML page that has nothing to do with XML (i.e does not point at all to XML) is less likely to lead (in the future) to new XML, as compared to a HTML page that already is deep involved with XML. We see here a notion of *interest* for HTML: pages that point to many XML pages are more interesting. We are interested not only by pages that point directly to XML, but also by those pages that point to pages that point to much XML. That is, an interesting HTML page points *directly or indirectly* to a lot of XML. Do we have here something resembling the page ranking algorithm we seen before?

In fact yes, but of a different flavor: while the *importance* moves *forward* on links, the *interest* of this algorithm moves *backward* on the Link graph. That is, a page is *important*

if it is pointed to by many *important* pages, while it is *interesting* if it points to many *interesting* pages.

So, in order to compute the *interest*, we use a similar fixpoint algorithm, but with different *input data*: we will run it on the inverse Link graph, that is the graph in which the direction of edges is inverted. On this graph we have  $j \in Out(i)$  when we had  $i \in Out(j)$  on the initial graph). Note that the matrix corresponding to this graph is the transposed of the initial graph matrix:

$$M^{Out} = M^t$$

where

$$M_{ij}^{out} = \begin{cases} \frac{1}{OutDeg(i)} & : j \in Out(i) \\ 0 & : otherwise \end{cases}$$

A second aspect is that while in the first algorithm all pages (HTML or XML) were treated the same, when computing the *interest* we are only interested by XML, so we want to bias the algorithm towards the XML pages. This can be done by using an appropriate *personalization vector*  $E$ .

We note with  $PR_{XML}$  the results of the direct page ranking, giving the importance of XML files, that we already explained, and with  $PR'_{HTML}$  the inverse PageRank we are considering now, giving the *interest* of HTML files. Let  $N_{XML}$  be the total number of XML pages ( $N_{XML} < N$ ).

We note  $\mathcal{PR}(matrix, personalization, initial)$  the iterative algorithm given by Equation 6, in which we have abstracted the input data: *matrix* representing the graph matrix ( $M$ ), *personalization* the personalization vector ( $E$ ), and *initial* the initial value for the PR vector ( $PR^0$ ). As explained in the previous section, the *importance* of XML pages is obtained by:

$$PR_{XML} = \mathcal{PR}(M, E, E) \quad \text{where} \quad E = \left[ \frac{1}{N} \right]_{N \times 1}$$

The *interest* of HTML pages is given by:

$$PR'_{HTML} = \mathcal{PR}(M^{Out}, E', E')$$

where

$$E'_i = \begin{cases} \frac{1}{N_{XML}} & : i \text{ corresponds to a XML page} \\ 0 & : otherwise \end{cases}$$

That is, we compute the page ranking on the inverse graph (we use  $M^{Out}$ ), using  $E'$  as personalization vector, which biases the algorithm towards XML pages, and with an initial  $PR$  value of 0 for HTML pages (also given by  $E'$ ).

Alternatively we can consider

$$PR''_{HTML} = \mathcal{PR}(M^{Out}, PR_{XML}, PR_{XML})$$

thus taking into consideration the importance of XML pages when computing the interest of HTML pages.

Observe that the technique we use for ranking XML pages and HTML pages leading to them can be used as well for focusing on some pages of specific interest, e.g. genomic resources.

### 5.3 Implementation

In this section, we discuss various aspects of our efficient and scalable implementation of page ranking in Xyleme. As the number of pages on the web is estimated to an order of  $10^9$  *now*, the ranking algorithm will have to cope with large values of  $N$ . Also we want the algorithm to be efficient, because a small lapse multiplied by  $N$  might mean hours of extra running time. We present the key-points that allow us to compute the *importance* and *interest* for up to  $10^9$  pages on a personal computer in reasonable time. A distributed variant of the algorithm is briefly mentioned in Section 5.4.

**The Naive Algorithm** One of the first things to observe is that the Link matrix is sparse. So we do not represent the Link graph as a square  $N \times N$  matrix, instead we use, for each node  $i$ , the list of its successors ( $Out(i)$ ). This is also the original representation of the Link 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 sometimes refer to this representation of links as the *Link matrix*.

The main lines of one iteration of the naive algorithm are given by:

$$\begin{aligned} &\text{for } j=1 \dots N, \text{ } destPR_j = 0 \\ &\text{for } i=1 \dots N, \text{ for each } j \in Out(i), \text{ } destPR_j = destPR_j + \frac{sourcePR_i}{OutDeg(i)} \\ &sourcePR = destPR \end{aligned}$$

Remember that we also want to compute the ranking on the inverse graph (to compute the *interest* of HTML pages). Luckily, we are able to do so using the same representation of the Link graph, and a different algorithm:

$$\begin{aligned} &\text{for } i=1 \dots N, \text{ } destPR'_i = 0 \\ &\text{for } i=1 \dots N, \text{ for each } j \in Out(i), \text{ } destPR'_i = destPR'_i + \frac{sourcePR'_j}{OutDeg(j)} \\ &sourcePR' = destPR' \end{aligned}$$

**Matrix Slicing** Here, we present the *Matrix Slicing* technique [13], which allows for efficient ranking computation for large  $N$ .

For large  $N$ , the Link matrix (i.e. the set of  $Out(i), \forall i$ ) or  $destPR$  and  $sourcePR$  would not fit in active main memory, so, we have to store them on disk. This is not such a big problem as long as our algorithm uses only on sequential access over data stored on disk. But we want to avoid having to use random access over such data on disk to avoid a too big performance penalty.

In the first (direct ranking) algorithm, we have sequential access over the  $sourcePR$  and the Link matrix, but random access on  $destPR$ . The  $destPR$  will be cut in  $S$  slices, each of these small enough to fit in memory. So each slice has  $\beta$  elements ( $\beta = \frac{N}{S}$ ). The  $destPR$  slice  $s$ , noted  $destPR_{|s}$ , will contain  $PR$  values for nodes in  $[s\beta$  and  $(s+1)\beta]$ . The Link matrix will also be cut 'vertically' in slices, such that each slice of the matrix will only reference elements of the corresponding  $destPR$  slice. (In matrix slice  $s$ , we will store  $\forall i, Out(i) \cap [s\beta, (s+1)\beta]$ ). We will process the matrix slices one by one. This way, when processing a given slice, we will have random access only on the corresponding  $destPR$  slice,

which fits in main memory. When we finish computing a given *destPR* slice, it is written on disk, and the next slice is prepared in memory.

In the case of *inverse PR* (the *interest* of HTML pages) we have sequential access over the *destPR'* vector, and random access on *sourcePR'*. So, in order to apply the slicing technique to *inverse PR*, we have to slice the *sourcePR'*. Again, we are able to use the same matrix slices for both algorithms.

**Gauss-Seidel Iterations** We present a further optimization in the spirit of Gauss-Seidel [11] that can be applied when using the slicing technique presented above. This optimization allows for quicker convergence, while diminishing the disk space requirements, and this, at no cost.

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

So, using this simple optimization, we have quicker convergence because we compute each slice starting from a better approximation of the fixpoint. This has been verified experimentally. Also, we only need one *PR* vector instead of *sourcePR* and *destPR*.

**Combined computation of multiple ranking** Here we present a method to further speed up the algorithm when we need to compute both direct and inverse ranking (as is the case with Xyleme).

The bottleneck of the ranking algorithm is not the CPU but the disk. Every ranking iteration does a complete scan of the Link matrix, which generates a lot of disk I/O. When computing independently a direct and an inverse ranking, we will perform a matrix scan for each of them. Instead, we can mix the two computations, and perform them both while doing a single matrix scan. Of course, we need to store in the main memory the *PR* 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 combined computation. Nevertheless, considering this trade-off, it turns out that it is more efficient to save a matrix scan even at the cost of doubling the number of slices.

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

**Measures** We next present some preliminary measurements of the ranking algorithm with simulated data. We plan measurements with real data in the near future.

We used an Intel Pentium II PC, with 128Mo 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. The times that are given are the time used by 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. See Table 5.3. 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.

|                  |         | 80Mo RAM | 40Mo 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

## 5.4 Distribution

The algorithm we presented in Section 5.3 is designed to run on a single machine. As the number of web pages grows, it may become necessary to distribute the processing between more than one machines. This is considered next,

We have seen that, while the computation of a slice of  $PR^{i+1}$  vector needs the whole  $PR^i$  vector, it only depends on the corresponding slice of the matrix. That is, the computation of a slice of  $PR^{i+1}$  can be done independently of the other slices, using the corresponding slice of the matrix and the whole  $PR^i$  vector. This property is very useful for implementing a distributed algorithm: we use a number of slices equal to the number of participating computers. In the algorithm initialization phase, every computer is assigned a slice to process, and gets the corresponding slice of the matrix. In this way, the main memory, disk space and CPU requirements are evenly distributed between all the stations. The computation of each slice of  $PR^{i+1}$  can take place in parallel, as these computations are independent. More precisely: in iteration  $i$ , every computer  $C_k$  computes  $PR_{||k}^{i+1}$  (the  $k^{th}$  slice of  $PR^{i+1}$ ); for this it needs  $PR^i$  and  $M_{||k}$  (the  $k^{th}$  slice of  $M$ ). The problem comes when passing from iteration  $i$  to iteration  $i+1$ , because at this moment all the stations must get the whole  $PR^{i+1}$  (which has just been distributively computed). So now, every  $C_k$  must get  $PR_{||l}^{i+1}, l \neq k$  from  $C_l$ . This implies quite an intense network traffic, a potential bottleneck. However, recall  $PR^i$  is of the order of ten times smaller than  $M$ , i.e., for each machine, the quantity of data involved in disk I/O is much larger than that sent over the net.

In the previous algorithm, machines operate synchronously, i.e. all machines work on iteration  $i$  at the same time. We next sketch an improved distributed algorithm, that does not

require such synchronization and provides quicker convergence. The basic idea is to abandon the notion of *iterations* as seen above, which was inherited from the non-distributed algorithm. Instead, each station  $k$  computes (as fast as it can) its new  $PR_{||k}$  slice, demanding to the other stations  $l$  the latest version of  $PR_{||l}$  they might have already computed. Meanwhile, it will answer requests from the other stations for  $PR_{||k}$  with the newest version that it has already computed. The intuition for the resulting quicker convergence is that every newly computed slice is closer to the fixpoint, and so we should use it as early as we can for the subsequent computations. This kind of fixpoint computation is known as *chaotic iterations*.

**Remark 5.1** An alternative to distributed ranking is an incremental algorithm that would gradually compute the ranking as the Link graph is explored and updated. The development of an incremental ranking algorithm is currently under consideration.  $\square$

## 6 Refresh policy

A most critical issue in Xyleme is when to decide to read (again) a page. The page may be an XML page that is already stored by Xyleme or an HTML page that we read simply in the hope that it leads to new XML pages. Indeed, it may also well be a page that Xyleme knows of (it is referenced by a known page) but has never been loaded yet. So, refresh basically involves maintenance of known pages and *discovery* of new pages. This is the topic of the present section.

The quality of the warehouse will critically depend on the pages it stores and their freshness. Thus, we want to minimize the delay between the time an XML page changes on the web and the time this change is detected by Xyleme. We also want to minimize the delay between the arrival of a new XML page on the web and its discovery by Xyleme. So, a main issue is the refreshing of XML pages and also of HTML pages (that may lead to new XML pages). A subtlety is that the robot only crawls to a certain depth, so it will not load certain pages although they have already been discovered. The Refresh Module may decide to read such pages, so in fact, the discovery of the web is split in two: a superficial very efficient scouting by Xyro, and a more in depth search by the Acquisition Module.

Our refresh policy is guided by a cost function. We present a simplified version of this function here. In particular, we will mostly ignore the following three aspects:

*Query subscription* Users may have asked to be notified weekly of the changes of particular documents, and to service that query subscription (see, e.g., MindIt [17]), Xyleme should refresh these documents before sending the notification.

*Publication* The web master of a web catalog may use Xyleme publication interface to let the system know that a new document or the new version is out. The Publication module could also tell us about some cluster of documents (e.g., a web catalog). We may want to see such a cluster as a single document and refresh all its pages *relatively* simultaneously to maintain some kind of coherence. In the complete setting, the cost function takes also into account the cost incurred if a publication order or a query subscription is not going to be serviced in time.

*Allocating resources* Xyleme administrators are offered the means to influence the general refresh strategy. For instance, the administrator may decide that it is becoming more important to refresh known pages that invest in exploring new branches of the Web (e.g., exploring deeply an important site). The cost function is parameterized in such a way that it is possible to do so by simply adjusting parameters. It is conceivable – although we did not address the issue yet – to have these parameters automatically adjusted by the system.

The desiderata of Xyleme refresh policy is to minimize the average obsolescence of the Xyleme Repository under constraint. The cost function takes in account:

1. Frequency rate: As shown in [8], the changes of Web pages are well described by a random Poisson Process. (Although updates are in general well described by this model, 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 published for the webmaster. An issue is to detect it automatically. We will ignore this issue here.)
2. Page Importance (as introduced in Section 5).
3. Time Unit, i.e., a parameter that fixes the shortest interval between two refreshes of the same page. In the current setting, the time unit is 24 hours.
4. Crawler Bandwidth: the bandwidth of the crawler is the average number of pages that the crawler can refresh in a time unit. It is fixed to 2.4 million pages in the current setting.

In this section, we present a simplified view of the cost function. We describe first the frequency rate (Section 6.1) and introduce the notion of page obsolescence (Section 6.2). Based on that we first consider a cost function that ignores page importance (Section 6.3) and then one that takes it into account (Section 6.4). In Section 6.5, we consider the trade off between acquiring new pages and maintaining the pages we already know. Due to space limitations, the justifications of formulas are not presented in this paper. At the end, we mention the status of the implementation (Section 6.6).

## 6.1 Estimating the frequency rate

The update of a page is modeled by a Poisson process, so every page is characterized by the average frequency  $\lambda$  of changes. If this change frequency is not specified by publication, it is estimated from observations, i.e., from detecting whether the page changed when attempting to read it. Note that the observation tells us whether the page changed and not how many times it did between two accesses. Web servers might also provide the date when the page was last modified. This information is called *last date of change*. This turns out to provide more informations than the *observations of change*. Thus, a priori, we use *last date of change* when available unless (i) it is not available or (ii) Xyleme detects that the web server provides an unreliable last date of change.

We present the estimation of  $\lambda$  in both cases considering first the use of *change observation* and then *last date of change*.

### 6.1.1 Using change observations

In this estimation, Xyleme only knows if a page has changed or not since the last access.

This is formalized as follows. Consider a page that is accessed  $N + 1$  times, at time  $t_i, i = 0 \dots N$ . The  $i^{\text{th}}$  access interval is  $[t_{i-1}, t_i], i = 1 \dots N$ . If Xyleme observed that the page changed during this interval, it is called a ‘change’ interval, otherwise it is a ‘no-change’ interval. We note  $C$  (respectively  $V$ ) the number of change (respectively no-change) intervals. We note  $c_i, i = 1 \dots C$  the length of the change intervals, and  $v_i, i = 1 \dots V$  the length of no-change intervals. When  $C > 0$  and  $V > 0$ , we get the estimated  $\lambda$  as the point that maximizes the function<sup>4</sup>:

$$f(x) = \left( e^{-x \sum_{i=1}^V v_i} \right) \sum_{j=1}^C (1 - e^{-c_j x}), \quad x \geq 0 \quad (9)$$

The particular case of *existence of change* with regular access ( $\forall i, t_{i+1} - t_i = \text{constant}$ ) is discussed in [9]. However, we believe that the assumption of regular access is too biased. Therefore, we use a generalization of the result of [9] that does not assume regularity of accesses.

### 6.1.2 Last date of change

In this estimation, Xyleme knows when the page changed for the last time. The estimation of  $\lambda$  based on this information is more accurate than the last one, and so this will be preferred method when the last date of change is available and seems reliable.

In addition to the notations introduced in the previous section, we use  $d_i, i = 0 \dots N$  to designate the last date of change reported at the  $i^{\text{th}}$  access. We have a ‘change’ interval if  $t_{i-1} < d_i \leq t_i, i = 1 \dots N$ . The first access will always represent a ‘change’ interval<sup>5</sup> (always  $d_0 < t_0$ ). For every change interval  $i$ , we note  $l_i = t_i - d_i$ .

With the above notations, the estimated value of  $\lambda$  is

$$\lambda = \frac{C}{\sum_{i=1}^C l_i} \quad (10)$$

A similar result is presented in [9]. A (small) difference with [9] is that we do not assume that the first access always represents a change.

## 6.2 Obsolescence

In this section we introduce the notion of page and repository obsolescence.

### 6.2.1 Page Obsolescence

As shown in the previous section, the page change is modeled by a Poisson process. Then, in a time interval  $t$ , the expected number of changes of a page is  $\lambda t$ . In other words, the

<sup>4</sup>In the given conditions the function is maximized in a unique finite point.

<sup>5</sup>Strictly speaking, this is not an *interval* as defined above, but will count as one.

number of changes of a page is proportional with the rate of change ( $\lambda$ ) and with the elapsed time ( $t$ ).

We define the *obsolescence* at time  $t$ , of a page with a estimated rate of change  $\lambda$ , which was last refreshed at time  $t_0$ , to be:

$$O_{\lambda,t_0}(t) = \lambda(t - t_0) \quad (11)$$

For a page  $i$ , we also use  $O_i(t)$  to denote  $O_{\lambda_i,t_{0_i}}(t)$ .

We can see the obsolescence of a page as the ‘time’ elapsed since the page was last refreshed. But the ‘time’ does not run the same way for all pages: for a page which changes often (high  $\lambda$ ), the ‘time’ goes quickly, while for a ‘slow’ (low  $\lambda$ ) page, the ‘time’ lingers. So came the saying “a day is like an year for the eager, and like a second for the idle”. Now we have an intuitive view of the  $\lambda(t - t_0)$  expression for the page obsolescence.

To simplify, we will consider here that the penalty incurred for a given page is given by the page obsolescence. In Xyleme, we consider a more complex penalty that is better described by:

$$\alpha\lambda(t - t_0) + \beta\mu(t, t_0, \lambda)$$

where  $\mu$  gives the probability that the page is stale. The intuition is that you are penalized if the page is stale, and you are also penalized if the page is old. The parameters  $\alpha$  and  $\beta$  can be tuned by the administrator based on user needs.

## 6.2.2 Xyleme Repository Obsolescence

In the following,  $N$  is the total number of pages in the Xyleme repository. We define the *obsolescence of the Xyleme repository* as the sum of the obsolescence of individual pages:

$$\Omega(t) = \sum_{i=1}^N O_i(t) \quad (12)$$

## 6.3 Simplified Cost Function

The cost function has to distribute a certain resource (the crawler bandwidth) by a time unit to the whole Xyleme repository. Say a page has been allocated, or a given time interval, a certain quantity of bandwidth, which will allow  $n$  refreshes for this page during the interval. We consider that the page is up to date at the beginning of the interval. The question is how are the refreshes distributed inside the interval. We prove that, in order to minimize the total obsolescence of this page during the interval, the refreshes have to be uniformly (equidistantly) distributed inside the interval.

We note  $t_i$  the moment of the  $i^{th}$  refresh ( $i = 1..n$ ).  $t_0$  is the beginning of the interval, and  $t_{n+1}$  the end of the interval.  $L = t_{n+1} - t_0$  is length of the interval. We have  $t_i \geq t_{i-1}$ ,  $i = 1 \dots n + 1$ . We note  $d_i = t_i - t_{i-1}$ ,  $i = 1 \dots n + 1$ . We have

$$d_i \geq 0, \quad i = 1 \dots n + 1 \quad \text{and} \quad \sum_{i=1}^{n+1} d_i = L \quad (13)$$

The obsolescence of the page at some point  $t$ ,  $t \in [t_{i-1}, t_i]$  is  $O_{\lambda, t_{i-1}}(t) = \lambda(t - t_{i-1})$ . The total obsolescence between  $t_{i-1}$  and  $t_i$  is

$$\int_{t_{i-1}}^{t_i} O_{\lambda, t_{i-1}}(t) dt = \int_{t_{i-1}}^{t_i} \lambda(t - t_{i-1}) dt = \lambda \frac{(t_i - t_{i-1})^2}{2}$$

The total obsolescence in the whole interval is

$$\sum_{i=1}^{n+1} \lambda(t_i - t_{i-1})^2 / 2 = \lambda \sum_{i=1}^{n+1} \frac{d_i^2}{2}$$

We define the function  $f$  giving the total obsolescence:

$$f(d_1, \dots, d_{n+1}) = \lambda \sum_{i=1}^{n+1} \frac{d_i^2}{2} \quad (14)$$

We seek the minimum total obsolescence, that is the minimum of  $f(d_1, \dots, d_{n+1})$  under the constraint given by equation 13. By applying the Lagrange multiplier method, we obtain that the minimum is reached when

$$d_i = d_j, \quad i, j = 1 \dots n + 1$$

i.e. when the refreshes are uniformly distributed, q.e.d.

The conclusion is: a page with a given  $\lambda$ , in a given Xyleme context, has to be refreshed at regular intervals in order to insure minimum average page obsolescence.

**The refresh interval of a page** Based on the above conclusion, we associate with each page  $i$  a new parameter,  $t_i$ , which gives the refresh interval for that page (as a number of time-units).

### 6.3.1 Constraints

In order to analyze the relation between the refresh of pages and the crawler bandwidth, noted  $G$ , we see the crawler as a limited resource: each page refresh using a given quantity of this resource. We associate with each page  $i$  a new parameter  $t_i$ , which gives the refresh interval for that page (as a number of time-units).

Each refresh consumes one unit of bandwidth. A page which is refreshed once every  $t$  time units, will consume one unit of bandwidth over  $t$  time units, that is  $\frac{1}{t}$  bandwidth per time unit. So, the set of all the pages will use  $\sum_{i=1}^N \frac{1}{t_i}$  per time unit. The relation between the set of refresh intervals  $(t_i, i = 1..N)$  and the crawler is:

$$G = \sum_{i=1}^N \frac{1}{t_i} \quad (15)$$

### 6.3.2 Average obsolescence

The average obsolescence of a page  $i$  refreshed with a period  $t_i$  is:

$$\int_{t_{access}}^{t_{access}+t_i} O_{\lambda_i, t_{access}}(t) dt = \int_0^{t_i} \lambda_i t dt = \lambda_i \frac{t_i^2}{2} \quad (16)$$

So, the average obsolescence of the Xyleme repository is

$$\Omega(t_1, \dots, t_N) = \sum_{i=1}^N \lambda_i \frac{t_i^2}{2} \quad (17)$$

### 6.3.3 When to refresh

Our goal is to minimize  $\Omega(t_1, \dots, t_N)$  under the constraint given by equation 15.

We apply again the Lagrange multiplier method, and get that the minimum average obsolescence is reached when:

$$t_i = \frac{1}{\sqrt{\lambda_i}} \frac{\sum_{j=1}^N \sqrt{\lambda_j}}{G} \quad (18)$$

This formula is the core result of Xyleme refresh. In the following, we note:

$$K = \frac{\sum_{i=1}^N \sqrt{\lambda_i}}{G} \quad (19)$$

A page will be refreshed when the time elapsed since the last refresh reach

$$t_i = K / \sqrt{\lambda_i} \quad (20)$$

The way we compute  $t_i$  ensures that, after a 'warm-up' period, the refresh will come to an equilibrium point, when the number of pages that reach the 'refresh point'  $t_i$  in any time-unit will be very close to  $G$ .

Based only on  $\lambda_i, i = 1..N$  and  $G$ , we can compute the average obsolescence of the Xyleme repository:

$$\Omega(G) = \frac{(\sum_{i=1}^N \sqrt{\lambda_i})^2}{2G}$$

## 6.4 Dealing with Page Importance

In the previous discussion, the decision concerning when to refresh a page was based only on the (estimated) rate of change of the page, and the time elapsed since the last refresh. We now introduce the notion of page importance (see Section 5).

To illustrate the impact of importance, consider two pages  $A$  and  $B$  (of about the same size) with  $A$  10 times more important than  $B$ . The intuition is that  $A$  is visited 10 times more than  $B$ . Thus at the same cost (loading the page), the benefit is much higher when we read  $A$  than  $B$ . Thus, pages with higher importance will be refreshed more often than pages with lower importance, all other parameters being equal. In practice, every page ( $i$ ) will have associated its importance  $r_i$ , a real value centered on 1 (i.e.,  $\sum_{i=1}^N r_i = N$ ).

The importance is taken in consideration when computing the obsolescence of a page, the equation (11) becoming:

$$O_{\lambda,r,t_0}(t) = r * \lambda * (t - t_0) \quad (21)$$

Thus, a page with a higher importance becomes obsolete quicker, and so is refreshed more often.

The modification of the obsolescence equation is then propagated through the equations presented in the previous section, and we get the final result which takes into consideration the importance:

$$t_i = \frac{1}{\sqrt{r_i \lambda_i}} \frac{\sum_{j=1}^N \sqrt{r_j \lambda_j}}{G} \quad (22)$$

Note that pages with importance  $r_i = 0$  will never be refreshed, while pages with importance very close to 0 will be seldom refreshed. This might be the intended behavior. Indeed, one might want to decide of a threshold below which any page will neither be read nor refreshed. If the policy is to read/refresh pages no matter what, the PageRank algorithm has to be modified to provide a minimum importance for all pages.

## 6.5 Discovery vs. maintenance

First observe that the crawler (like standard crawlers) has all the machinery needed to discover the web by itself. However, this discovery is simplistic in the sense that it is not guided by the importance of pages. This is used when Xyleme is started but soon, the refresh module starts feeling the crawler queue of request and the decisions to read pages mostly come from the refresh module.

An issue that we ignored here is that of maintenance (refresh) vs. discovery (read for the first time). The process we described uses the same cost function to decide whether to read a new page or to re-read an already known page. In practice, we want to provide the means to tune the system to favor discovery vs. maintenance or vice versa. Indeed, at different stages, different policies may be chosen, e.g., more resources for discovery in a first phase then slowly moving to more resources for maintenance. It is easy to do so. We fix a priori some parameter  $\gamma$  between 0 and 100. At each step, we devote  $\gamma$  per cent of the crawler bandwidth to discover new pages and the rest for refresh. This forces us at each step, to compute a different obsolescence for known pages (pages already read) and unknown ones (page we heard of at some time  $t$  but never read).

## 6.6 Implementation

We use a global variable  $S = \sum_{i=1}^N \sqrt{r_i \lambda_i}$ . For a page that has just been refreshed, we know the old  $r_i \lambda_{i_{old}}$  value (the refresh rate estimation we had before the refresh), and the new  $r_i \lambda_{i_{new}}$  (the estimation done after the refresh<sup>6</sup>). The value of  $S$  is updated as follows:

$$S_{new} = S_{old} - \sqrt{r_i \lambda_{i_{old}}} + \sqrt{r_i \lambda_{i_{new}}}$$

This way, we always know the up to date value for  $\sum_{i=1}^N \sqrt{r_i \lambda_i}$ .

---

<sup>6</sup>We do the same calculus after the Page Rank process.

The refresh criteria presented (equation (20)) has the nice property of allowing us to know when a page is to be refreshed, only by looking at the importance ( $r$ ), rate of change ( $\lambda$ ) and last access time for that page.

We will do a complete scan over the set of pages once per time-unit; each time it finds a page that has reached the ‘refresh point’ (i.e., the time elapsed since the last refresh is  $\geq$  than the corresponding refresh period ( $t_i$ )), it will send the page to the crawler. The way we computed the refresh period for a page insures that, this way, we will refresh exactly<sup>7</sup>  $G$  pages per time-unit. As the time-unit we use is relatively large (e.g., 24 hours), a scan over all pages per time unit is perfectly acceptable (i.e. not too expensive).

Nevertheless, shall such a scan prove impractical, another implementation solution can be used: the pages are organized in folders, a folder corresponding to a time-unit and containing the pages to be refreshed in that time-unit. When the folder for the actual time-unit is processed, all the pages it contains<sup>8</sup> are refreshed and distributed to new folders, based on their parameters ( $r, \lambda$ ). When all the pages from the current folder are processed, the folder is destroyed, and we start processing the next folder.

The advantages of this method are that it only needs to scan  $G$  pages per time unit. The disadvantage is the more complex data structures, and the additional storage space needed. So, we have here a CPU — storage tradeoff. Also, the error in the estimation of the moment of refresh for a page is bigger when using the second method<sup>9</sup>.

## 7 Conclusion

As mentioned earlier, this is a report on an on-going research. All modules discussed in this paper have been implemented and (to some extent) integrated. They are currently being tested.

We started only recently crawling the web in a significant way for obtaining XML documents. We need to continue testing the system. For instance, Page Ranking has mostly been tested so far with synthetic data. We would like to test it with real data and see whether this affects key factors such as the convergence rate. We also want to gather 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 rate 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.

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

---

<sup>7</sup>Strictly speaking, not ‘exactly’  $G$ , but a very close value.

<sup>8</sup>We expect it to contain about  $G$  pages.

<sup>9</sup>Because the refresh period ( $t_i$ ) estimation uses  $K$  (see equation (19), (20)), which might slowly change in time, more recent  $K$  values provide a more reliable estimation.

## 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 Watez. 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 IEEE Computer Society Technical Committee 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] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. In *9th WWW*, 2000.
- [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] Google incorporation. <http://www.google.com/>.
- [13] T. H. Haveliwala. Efficient computation of pagerank, 1999. Stanford Database Group.
- [14] 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>.
- [15] S. Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D. Sivakumar, Andrew Tomkins, and Eli Upfal. The web as a graph. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-17, 2000, Dallas, Texas, USA*, pages 1–10. ACM, 2000.
- [16] Amélie Marian, Serge Abiteboul, and Laurent Mignet. Change-centric management of versions in an XML warehouse, October 2000. BDA'00 to appear, [www.xyleme.com/DOCS/version.ps](http://www.xyleme.com/DOCS/version.ps).

- [17] Mindit. <http://mindit.netmind.com/>, 2000.
- [18] The Robots Exclusion Protocol.  
<http://info.webcrawler.com/mak/projects/robots/exclusion.html>.
- [19] ht://Dig WWW Search Engine Software.
- [20] Wget. <http://www.gnu.org/software/wget/wget.html>.
- [21] World Wide Web Consortium. eXtensible Markup Language (XML) 1.0.  
<http://www.w3.org/TR/REC-xml/>.
- [22] World Wide Web Consortium. HyperText Markup Language (HTML) 4.1.  
<http://www.w3.org/MarkUp/>.
- [23] J. Widom. Research problems in data warehousing. In *CIKM '95 (invited paper)*, 1995.
- [24] XML Linking Language (XLink), February 2000.
- [25] Xyleme Project. <http://www-rocq.inria.fr/verso/Xyleme.html>.
- [26] Yahoo! <http://www.yahoo.com/>.