

Data Acquisition for an XML Warehouse
DEA Internship, Project Verso, INRIA¹

Mihai Preda
DEA Sémantique Preuve Programmation
ENS Cachan

September 12, 2000

¹Institut National de Recherche en Informatique et Automatique

Abstract

Ce travail, qui conclut le stage de DEA de six mois, a été effectué à l'Institut National de Recherche en Informatique et Automatique (INRIA) à Rocquencourt, au sein du projet Verso, sous la direction de Serge Abiteboul.

Le stage a été accompli dans le cadre du projet *Xyleme*, un *entrepôt dynamique de données XML du web*. Xyleme a pour but d'intégrer l'ensemble des données XML qui se trouvent sur le Web (*acquisition*), de stocker et versionner ces documents (*repository*), de réaliser un groupement sémantique et de permettre une interrogation facile et complexe (*query*).

Mon stage a concerné la partie *Acquisition* de Xyleme, qui s'occupe de trouver et acquérir les documents XML du web (*Découverte*), et de maintenir ces données à jour (*Rafraîchissement*). Le rafraîchissement est nécessaire parce que les documents qui se trouvent sur le web sont modifiés continuellement et l'information que détient Xyleme devient périmée.

Pour réaliser la découverte et le rafraîchissement des documents XML d'une manière efficace et économique, on tient compte des informations telles que leur *frequence de changement* et *l'importance* des pages. Un document qui change fréquemment ou qui est important sera rafraîchi plus souvent que celui qui n'est jamais modifié ou qui est peu important. En ce qui concerne la découverte, il faut acquérir les documents importants en priorité.

Le travail accompli pendant le stage comporte:

- La conception de module *PageRank*, inspiré par l'algorithme similaire utilisé par Google, qui calcule l'importance des pages à partir du graphe des liens entre ces pages. Les parties originales sont: la mise au point de *Inverse PageRank*, un algorithme adapté à Xyleme pour calculer l'importance des pages HTML dans le cadre spécifique de Xyleme; une amélioration de l'algorithme pour augmenter la vitesse de convergence; la présentation de deux manières de calcul de PageRank distribuées.
- La mise au point de l'algorithme pour l'estimation de taux de changement des pages à partir d'un ensemble d'observations de l'état des pages. Une extension des méthodes présentées en [12] a été réalisée.

- La conception originale de l'algorithme pour le contrôle du rafraîchissement et de la découverte des pages. Cet algorithme est optimal dans le sens qu'il assure une obsolescence moyenne minimale de l'ensemble des pages de Xyleme. L'algorithme a été étendu pour permettre le contrôle de plusieurs catégories de pages et adapté à une exécution distribuée. Il est incorporé, dans le module *PageScheduler*.

Le stage a compris aussi l'implémentation des deux modules, PageRank et PageScheduler, ce qui a offert la possibilité de réaliser des analyses de performance pour le Page Rank et de valider le Scheduler.

Le travail effectué est présenté dans le contexte de Xyleme, mais les principaux résultats peuvent être utilisés dans une large catégorie d'applications, par exemple des moteurs de recherche sur Internet ou des serveurs proxy faisant du caching.

L'acquisition de données dans ces applications est encore faite de manière très primitive. Les techniques que nous présentons ici permettent (à ressources constantes) d'améliorer la qualité des informations fournies de manière importante.

Contents

1	Introduction	4
1.1	Context	4
1.2	Our work	4
1.2.1	Discovery and Refresh	5
1.2.2	Change rate	6
1.2.3	Importance	6
1.3	State of the art	6
2	Page Ranking	7
2.1	Page Ranking	7
2.2	Xyleme page ranking	9
2.3	Implementation	11
2.3.1	The Naive Algorithm	12
2.3.2	Matrix Slicing	12
2.3.3	Gauss-Seidel Iterations	13
2.3.4	Combined computation of multiple ranking	14
2.4	Performance Measures	14
2.5	Distribution	15
2.5.1	Incremental ranking	16
3	Page Scheduler	17
3.1	Freshness and Obsolescence	18
3.2	Cost of Obsolescence	18
3.3	Refresh Pattern of a page	19
3.4	Obsolescence of the Xyleme repository	21
3.5	Bandwidth constraint	21
3.6	Implementation	22

<i>CONTENTS</i>	2
4 Change rate estimation	24
4.1 The Model	24
4.2 Estimation of the frequency of change	24
4.2.1 Existence of change	25
4.2.2 Last date of change	26
5 The Global Picture	27
5.1 Publication	27
6 Conclusion	29

Thanks

I would like to thank Serge Abiteboul, Directeur de Recherches for guiding my work with attention and interest, for correcting this report and for providing support. I also thank Benjamin and Laurent with whom I worked closely, for all their help.

Chapter 1

Introduction

1.1 Context

The web is huge and keeps growing at a healthy pace. Most of the data is unstructured, consisting mainly of HTML and other media such as video, images, sound. 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. A major evolution is occurring that will dramatically simplify the task of developing applications with this data, the coming of XML [6, 4, 21, 18]. The eXtensive Mark-up Language is a semi-structured data language, as opposed to the HyperText Mark-up Language, which is more of a display language. The generalization of XML adds a new dimension to data on the web today.

This work is part of the Xyleme project [19, 5, 1, 20] aiming at the development of a *dynamic XML warehouse for the web*.

1.2 Our work

We are studying and building Xyleme, a *dynamic World Wide XML warehouse* capable of storing all the XML data available. In this report we focus on the Acquisition Module of Xyleme, whose task is to discover, acquire and maintain up to date the XML documents of the web. We analyze the situation created by the constraint of limited resources (mainly network bandwidth).

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 the system of the existence of this data. Crawling is the more standard technique used by HTML search engines such as Alta Vista [7], Yahoo! [22] or Google [15]. Starting from some seeds, the crawler follows links to discover more and more pages. 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 documents. 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.

1.2.1 Discovery and Refresh

We call *discovery* the operation of fetching for the first time a page. We call *refresh* the subsequent fetching operations for a page already discovered.

Refreshing XML pages is needed to keep the XML warehouse up to date. As mentioned before, resources are limited and the 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);

Based on these observations, refreshing becomes an optimization problem, namely minimizing the gap between the web and the warehouse given limited resources (fetching capability), for some cost function taking into account the *importance* and *change rate* of pages.

The refresh is also important for HTML pages, because a modified HTML page could lead to new XML pages.

Similar problems exist for discovery, as we want to discover important pages first.

1.2.2 Change rate

The change rate of a page is estimated using the information available when fetching the page. Each time we refresh a page, we can tell if it has changed or not with respect to the last version we own. Some web servers make available additional information, such as the last date of change of the page. From a series of such observations, we can estimate the change rate.

1.2.3 Importance

The importance of a page is based on a page ranking technique in the style of Google [9, 16]. We compute a PageRank score for each page using the links graph. The intuition is that a page pointed by many important pages is important. This results in a fix-point computation on the whole graph of links between pages.

1.3 State of the art

A general analyze of the web links graph is presented in [10]. A variant of the PageRank algorithm, using the notions of *hubs* and *authorities* is described in [17]. The classical PageRank was popularized by its use in Google [15] to rank search results, and is described in [8, 9]. A technique for efficient main-memory PageRank computation is presented in [16].

Few work has been done on crawler scheduling policies, as shown by the fact that major search engines still use naive scheduling algorithms for their robots [3, 2]. Also, there is little transparency on this topic from the industry, the information publicly available being very reduced. Some basic results are shown in [13].

Experimental data showing that the page modification process follows a Poisson law is given in [11], as well as a simple method for estimating the page change frequency. A taxonomy of the change rate estimation is presented in [12], together with more complex estimators.

Chapter 2

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 [9] on the set of all XML and HTML pages. We recall this algorithm in Section 2.1. For ranking HTML pages we use a novel algorithm that we present in Section 2.2. We describe the status of the implementation together with some experiments in Section 2.3 and possible developments in a last section.

2.1 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, 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:

$$PR(i) = \sum_{j \in Out(j)} \frac{PR(j)}{OutDeg(j)} \quad (2.1)$$

We see that the importance of a page is equally distributed between all its successors. If we consider the set of equations (2.1) for every page, we get the following linear equation system:

$$PR_{N \times 1} = M_{N \times N} \times PR_{N \times 1} \quad \text{where} \quad M_{ij} = \begin{cases} \frac{1}{OutDeg(j)} & i \in Out(j) \\ 0 & \text{otherwise} \end{cases} \quad (2.2)$$

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 fix-point 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 fix-point 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 .

$$PR^{i+1} = M \times PR^i \quad (2.3)$$

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

$$\|PR^n - PR^{n-1}\|_2 < \epsilon \quad (2.4)$$

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.

$$PR = cM \times PR + (1 - c) \times E_{N \times 1} \quad \text{where} \quad c \in]0, 1[\quad (2.5)$$

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* [9] search engine uses $c = 0.85$.

The E vector in the equation above is a *personalization vector* [8]. 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. $\forall i, \text{Out}(i) \neq \emptyset$, 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:

$$PR^{i+1} = r \times (cM \times PR^i + (1 - c) \times E), \quad (2.6)$$

$$\text{where} \quad r = \frac{\|PR^i\|_1}{\|cM \times PR^i + (1 - c) \times E\|_1} \quad (2.7)$$

Thus we have:

$$\|PR^{i+1}\|_1 = \|PR^i\|_1 \quad (2.8)$$

We will see this in more detail when discussing the implementation of the algorithm (see Section 2.3).

2.2 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 fix-point 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 2.6, 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

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

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.

2.3 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 $2 * 10^9$ 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 billions of pages on a personal computer in reasonable time. A distributed variant of the algorithm is presented in Section 2.5.

2.3.1 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:

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

2.3.2 Matrix Slicing

Here, we present the *Matrix Slicing* technique [16], 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 sequential access over this 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 β 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.

2.3.3 Gauss-Seidel Iterations

We present a further optimization in the spirit of Gauss-Seidel [14] 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 fix-point, we expect *destPR* to be closer to the fix-point than *sourcePR*. So, we expect that a newly computed slice of *destPR* is closer to the fix-point 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 fix-point. This has been verified experimentally. Also, we only need one *PR* vector instead of *sourcePR* and *destPR*.

2.3.4 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.

2.4 Performance 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 fix-point 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 2.4. 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 2.1: Measures of page ranking

2.5 Distribution

The algorithm we presented in Section 2.3 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 $PR_{\parallel k}^{i+1}$ (the k^{th} slice of PR^{i+1}); for this it needs PR^i and $M_{\parallel k}$ (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 fix-point, and so we should use it as early as we can for the subsequent computations. This kind of fix-point computation is known as *chaotic iterations*.

2.5.1 Incremental ranking

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.

Chapter 3

Page Scheduler

We describe here the general design and implementation of the PageScheduler module in Xyleme. This module decides which pages have to be fetched at a given moment, hence it implements a *crawler scheduling policy*.

Although the discussion is based on Xyleme, the same ideas can be applied to any Web information system, e.g. any Web Search Engine, Web Mirroring or Monitoring System, or a caching proxy server.

Such a system contains (at least) a fetching module, a refresh module, and a processing module. The fetcher has the role of retrieving pages from the web. The processing module uses the retrieved pages for a specific goal. For example, the Processing module in Xyleme scans the HTML pages for links, and stores the XML pages, for further processing. The Processing module of a Search Engine indexes the retrieved pages, while that of a Monitoring System analyzes the pages in a specific manner.

The processing module can use only the information retrieved by the fetcher. It does not have access to the current version of the page (as it is on the web), it only sees the page as it was when the crawler fetched it.

As the web pages undergo modifications, the information used by the Processing module becomes obsolete. To avoid this, the pages need to be crawled again (*refreshed*), in order for the Processing module to obtain up to date information.

So, ideally, every time a page is modified, it will be immediately refreshed. This way, the Processing module will not have any obsolete information. Unfortunately, this is not always possible, for different reasons:

- The page modification is an event extern to the Web system; The

system cannot control it, and is not informed of the modification. The typical case is that of web pages modified by human beings.

- The crawler is a limited resource. That is, it has a limited capacity, it can only refresh a certain number of pages in a time interval.

The role of the Scheduler module is to deal with the situation presented above. It must decide which pages have to be crawled and when, in order to minimize the obsolescence of information, while respecting the existing constraints.

From an architectural point of view, the Scheduler module has close interactions with the Crawler module: the Crawler will provide the Scheduler with information concerning the pages being retrieved, while the Scheduler will “drive” the Crawler, by deciding which pages have to be crawled.

3.1 Freshness and Obsolescence

A Xyleme document becomes *obsolete* when it gets out of date with respect to the corresponding web document. When a web page changes, the Xyleme copy of this page is obsolete until the page is crawled again (refreshed). This applies directly to XML documents, which are stored by Xyleme, but also to HTML pages which, even if they are not stored, are used to discover new pages by following links. When a HTML page changes on the web, the links information Xyleme has for this page is no longer accurate (up to date), and the page needs refreshing.

We model the obsolescence by considering the negative impact it has on quality of data in Xyleme.

3.2 Cost of Obsolescence

A Xyleme document that is no longer identical to its 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’.

Imagine I have a Monday newspaper. It is out of date Tuesday, but even more Friday. 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 ‘too obsolete’.

We discriminate two components of obsolescence: the fact that a document is out of date, and the *distance* it is from the up to date version.

As mentioned before, Xyleme does not have complete knowledge of the modifications of a web document. As shown in [11], the modification process of a web page is well modeled by a random Poisson process, that is the modifications to a page are independent and happen at a given rate. In Section 4 we show how the rate of change can be estimated using the limited information Xyleme gets when retrieving pages.

As the changes occur accordingly to a Poisson process, the average number of changes of a page with change rate λ during a period t is λt . The distance of a document not refreshed for a time t from its up to date version is estimated to λt , if the document has change rate λ .

To capture the cost of obsolescence, we need to chose a function $f(t, \lambda)$. According to the previous discussion, this function should take into account the fact that the page became out of date and that it aged (is “very” out of date). We selected for cost the function $(\lambda t)^\alpha$. With $\alpha = 0.95$ (chosen in Xyleme) this function approximates what we believe is a “normal” cost. $(\lambda t)^\alpha$. The α parameter allows to some control on the two components of obsolescence mentioned before, the out of date factor and the age factor. A α value smaller than 1 favors the out of date factor, while α values greater than 1 favor the age 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 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* :

$$d_i(t) = w_i(\lambda_i t)^\alpha \quad (3.1)$$

where w_i is the importance of the page, and λ_i the estimated rate of change.

3.3 Refresh Pattern of a page

Now we focus at the page level, and analyze the refresh pattern of an individual page.

Say a page has been allocated, for a given time period, a certain quantity of bandwidth, which will allow n refreshes for this page during the period.

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 $l_i = t_i - t_{i-1}$, $i = 1 \dots n + 1$. We have

$$l_i \geq 0, \quad i = 1 \dots n + 1 \quad \text{and} \quad \sum_{i=1}^{n+1} l_i = L \quad (3.2)$$

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

$$\int_{t_{i-1}}^{t_i} d(t - t_{i-1}) dt = w\lambda^\alpha \frac{(t_i - t_{i-1})^{\alpha+1}}{\alpha + 1}$$

The total obsolescence in the whole interval is

$$\sum_{i=1}^{n+1} w\lambda^\alpha \frac{(t_i - t_{i-1})^{\alpha+1}}{\alpha + 1} = w \frac{\lambda^\alpha}{\alpha + 1} \sum_{i=1}^{n+1} l_i^{\alpha+1}$$

We define the function f giving the total obsolescence:

$$f(l_1, \dots, l_{n+1}) = w \frac{\lambda^\alpha}{\alpha + 1} \sum_{i=1}^{n+1} l_i^{\alpha+1} \quad (3.3)$$

We seek the minimum total obsolescence, that is the minimum of $f(l_1, \dots, l_{n+1})$ under the constraint given by equation 3.2. By applying the Lagrange multiplier method, we obtain that the minimum is reached when

$$l_i = l_j, \quad i, j = 1 \dots n + 1$$

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

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

3.4 Obsolescence of the Xyleme repository

We have seen that, in order to minimize the obsolescence cost of a page, we have to access it regularly. Now we can speak of the *access frequency* f_i of a page i (a page is accessed every $1/f_i$ time units).

The average cost of a page i refreshed with frequency f_i is

$$c_i(f_i) = \frac{\int_0^{1/f_i} d_i(t) dt}{1/f_i} \quad (3.4)$$

In the following, N is 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:

The *cost of obsolescence of the repository* is

$$C(f_1, \dots, f_N) = \sum_{i=1}^N c_i(f_i) \quad (3.5)$$

This cost functions represents the average obsolescence of the Xyleme repository, and is this cost that we want to minimize.

3.5 Bandwidth constraint

The *bandwidth* of the crawler is the number of pages the crawler can refresh in a time-unit. The crawler bandwidth is noted G . For example, the Xyleme crawler bandwidth is about 3,000,000 pages/day, or 30 pages/second.

The bandwidth constraint is the relation between the crawler bandwidth and the access frequencies of pages:

$$\sum_{i=1}^N f_i = G$$

We define the function $D(f_1, \dots, f_N)$ expressing the above constraint:

$$D(f_1, \dots, f_N) = \left(\sum_{i=1}^N f_i \right) - G$$

$$D(f_1, \dots, f_N) = 0 \quad (3.6)$$

We compute $f_i, i = 1 \dots N$ as to minimize C (3.5) under the constraint given by D (3.6). Using the Lagrange multiplier method, we get

$$f_i = \frac{G}{S} \sqrt[\alpha+1]{w_i \lambda^{\alpha_i}} \quad (3.7)$$

where

$$S = \sum_{i=1}^N \sqrt[\alpha+1]{w_i \lambda^{\alpha_i}} \quad (3.8)$$

The formulae 3.7 is the main result of Xyleme PageScheduler.

The way we compute f_i ensures that, after a 'warm-up' period, the 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 .

3.6 Implementation

The Scheduler manages the global variable S (3.8). For a page that has just been refreshed, we know the old λ_{old} value (the refresh rate 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:

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

This way, we always know the up to date value for S .

The refresh criteria presented (equation (3.7)) 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 (λ) and last access time for that page.

The Refresh will do a complete scan over the set of pages once per *acquisition cycle*; 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 ($1/f_i$)), it will send the page to the crawler. The way we computed the access frequency for a page insures that, this way, we will access exactly¹ G pages per acquisition cycle. As the acquisition cycle we use is relatively large (24 hours in Xyleme), a scan over all pages per time unit is perfectly acceptable (i.e. not too expensive).

¹strictly speaking, not 'exactly' G , but a very close approximation of G .

Nevertheless, shall such a scan prove impractical, another implementation solution can be used: the pages are organized in folders, a folder corresponding to a acquisition cycle and containing the pages to be refreshed in that cycle. When the folder for the actual cycle is processed, all the pages it contains² are refreshed and distributed to new folders, based on their parameters (r, λ) . 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 acquisition cycle. 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 larger when using the second method³.

²We expect it to contain about G pages

³Because the access frequency (f_i) estimation uses S (see Equation 3.8, 3.7), that slowly changes in time, more recent S values provide a more reliable estimation.

Chapter 4

Change rate estimation

4.1 The Model

In the general case, Xyleme does not have knowledge of the underlying process (a webmaster, for example) controlling page update. Nevertheless, in order to deal with this process, we need to model it. As shown in [11], the changes of Web pages are well described by a random Poisson process. So, by default¹ we model the page update by a Poisson process. But even if in general² the page update is well described by a Poisson process, there might be certain pages that follow a different modification pattern (for example, pages that change at a regular interval). How information about specific update patterns is used by the PageScheduler is described in Chapter 5.

The only parameter of a Poisson process is the process rate, λ , which gives the average frequency of the random event, in our case the average rate of page update, for a specific page. Every page is characterized by a such λ ; but the λ of a page is not known, and will have to be estimated by Xyleme. How this is done is the described in this section.

4.2 Estimation of the frequency of change

The update of a page is modeled by a Poisson process, so every page is characterized by the average rate (frequency) of change. Even if Xyleme

¹‘by default’ means when we don’t have any additional information available concerning the update of given page

²‘in general’ meaning when we consider the whole set of web pages.

does not know the frequency of change (λ) for a given page, it can estimate it from a number of observations. When Xyleme refreshes the page, it can see if the page has changed since the last access³.

The information available to Xyleme by such an observation is called *existence of change*, because Xyleme only knows if a page has changed or not between the accesses, but does not know when or how many times.

The Web servers might also provide, together with the requested page, the date when the page was last modified, called *last date of change*. Of course, *last date of change* provides more information than *existence of change*.

From a series of such observations, Xyleme can estimate of the actual rate of change (λ) of a page.

4.2.1 Existence of change

Now we present the estimation of λ in the situation where Xyleme knows, for each page access (refresh) if the page has changed or not since the last access.

We formalize the situation like this: The page is accessed $N + 1$ times, at time points $t_i, i = 0 \dots N$. The i^{th} access interval is given by $[t_{i-1}, t_i], i = 1 \dots N$. If Xyleme observed that the page changed during this interval, we call it 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 λ as the point that maximizes the function⁴:

$$f(x) = \left(e^{-x \sum_{i=1}^V v_i} \right) \prod_{j=1}^C (1 - e^{-c_j x}), \quad x \geq 0 \quad (4.1)$$

Note that in the general case the value that maximizes the function (4.1) cannot be obtained analytically; in this case, we can use approximations or numeric algorithms.

The particular case of *existence of change* with regular access ($\forall i, t_{i+1} - t_i = \text{constant}$) is discussed in [12]. Anyway, such a constraint as regular

³This can be done by comparing the new version with the old one, if the page is stored, or by comparing some signature (a message digest, for example) of the new and old version, if the page is not stored.

⁴In the given conditions the function is maximized in a unique finite point

access is not acceptable in the Xyleme case. The estimation method works for arbitrary access points, and is a generalization of the result presented in [12].

4.2.2 Last date of change

Most Web servers provide, together with the requested page, the date of the last modification. Here we present the estimation of λ in the situation where Xyleme knows, for each access, the *last date of change*.

The estimation of λ based on this information is much more accurate than the *existence of change* estimation, and so this will be the preferred method when the last date of change is available.

In addition to the notations introduced in Section 4.2.1, we use $d_i, i = 0 \dots N$ to designate the last date of change reported at the i^{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' interval⁵ (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 λ is

$$\frac{C}{\sum_{i=1}^C l_i} \quad (4.2)$$

A similar result is presented in [12]. The difference is that, in our case, the first access always represents a change, which is not the case in the mentioned article. This is the cause of the bias of the estimator given in [12].

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

Chapter 5

The Global Picture

We present here how the algorithms we seen before are integrated in Xyleme.

Xyleme deals with multiple categories of pages, such as XML, HTML or DTD. All need to be kept up to date, but with different priorities. We control how the crawler resources are allocating by associating an *importance factor* to each category of pages. These factors are used by the PageScheduler to weight the importance of pages of the corresponding categories. This way we can easily tune the acquisition at run-time, e.g. to refresh with priority XML pages, or to stop refreshing HTML, etc.

In a similar way we are able to specify the percentage of the crawler bandwidth that is used for *discovery* (i.e. fetching a page for the first time) or for *refreshing* pages.

5.1 Publication

Xyleme offers the possibility for the users to publish various information concerning web pages. It is possible to specify the change rate of a page, overriding the estimation done by Xyleme. Users can also demand special treatment for certain pages, by indicating for example that a page should never be refreshed, or by asking to crawl a page immediately.

Other modules of Xyleme (e.g. Page Monitoring [20]) might need to indicate a specific refresh frequency for a page. Note that this is not the same as specifying the change rate. PageScheduler handles this by allocating the necessary part of crawler bandwidth for “fixed refresh” pages.

We see that the specific needs of the system may demand extensions or

adaptations of the basic algorithm.

Chapter 6

Conclusion

A method for computing page importance from the links graph has been presented, which extends the classical PageRank algorithm for specific use by Xyleme. The algorithm has been implemented and benchmark data is shown.

Estimation of the frequency of change is done similarly with the methods presented in [12]. An algorithm for change rate estimation for the case of existence of change with random access is developed which is new with respect to [12].

An original optimal scheduling policy using a parameterizable cost function is presented. This algorithm together with the page importance computation using PageRank and with the change rate estimation represents a *comprehensive robot scheduling policy*.

These results are presented in the context of Xyleme, but they have a large are of direct applicability, ranging from improving existing web search engines to developing advanced caching servers.

Bibliography

- [1] A dynamic warehouse for the xml data of the web. Document can be found at: <http://www-rocq.inria.fr/xyleme/DOCS/general.ps>.
- [2] Search engine watch. <http://www.searchenginewatch.com/>.
- [3] Search tools for web sites, intranets and portals. <http://www.searchtools.com/>.
- [4] World wide web consortium page on xml. <http://www.w3c.org/TR/REC-XML>.
- [5] Xyleme home page. <http://www.xyleme.com/>.
- [6] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web*. Morgan Kaufmann, California, 2000.
- [7] Altavista. <http://www.altavista.com/>.
- [8] 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.
- [9] Sergey Brin and Lawrence Page. The anatomy of a large-scale hyper-textual web search engine. In *7th WWW*, 1998.
- [10] 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.
- [11] 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/>.

- [12] Junghoo Cho and Hector Garcia-Molina. Estimating Frequency of Change. Technical report, Stanford University, 2000. <http://dbpubs.stanford.edu:8090/pub/1999-22/>.
- [13] Jr. E.G. Coffman, Zhen Liu, and Richard R. Weber. Optimal Robot Scheduling for Web Search Engines. Technical report, INRIA - Sophia Antipolis, 1997.
- [14] M. Gondran and M. Minoux. *Graphes et Algorithmes*. Eyrolles, 1995.
- [15] Google incorporation. <http://www.google.com/>.
- [16] T. H. Haveliwala. Efficient computation of pagerank, 1999. Stanford Database Group.
- [17] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. In *Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1998.
- [18] Alain Michard. *XML, langage et applications*. Eyrolles, Paris, 1999.
- [19] Laurent Mignet, Serge Abiteboul, Mihai Preda, Sébastien Ailleret, Bernd Amann, and Amélie Marian. Acquiring xml pages for a web-house, October 2000. To appear in BDA'00.
- [20] Benjamin Nguyen. Temporal queries and monitoring of a data warehouse, June 2000. DEA Internship.
- [21] W3C. *eXtensible Markup Language (XML) 1.0*, february 1998.
- [22] Yahoo! <http://www.yahoo.com/>.