The Evolution of the Web and Implications for an Incremental Crawler

Junghoo Cho, Hector Garcia-Molina
Department of Computer Science
Stanford, CA 94305
{cho, hector}@cs.stanford.edu

Abstract

In this paper we study how to build an effective incremental crawler. The crawler selectively and incrementally updates its index and/or local collection of web pages, instead of periodically refreshing the collection in batch mode. The incremental crawler can improve the "freshness" of the collection significantly and bring in new pages in a more timely manner. We first present results from an experiment conducted on more than half million web pages over 4 months, to estimate how web pages evolve over time. Based on these experimental results, we compare various design choices for an incremental crawler and discuss their trade-offs. We propose an architecture for the incremental crawler, which combines the best design choices.

1 Introduction

A crawler is a program that automatically collects Web pages to create a local index and/or a local collection of web pages. Roughly, a crawler starts off with an initial set of URLs, called *seed URLs*. It first retrieves the pages identified by the seed URLs, extracts any URLs in the pages, and adds the new URLs to a queue of URLs to be scanned. Then the crawler gets URLs from the queue (in some order), and repeats the process.

In general, the crawler can update its index and/or local collection in two different ways. Traditionally, the crawler visits the web until the collection has a desirable number of pages, and stops visiting pages. Then when it is necessary to refresh the collection, the crawler builds a brand new collection using the same process described above, and then replaces the old collection with this brand new one. We refer to this type of crawler as a *periodic crawler*. Alternatively, the crawler may keep visiting pages after the collection reaches its target size, to *incrementally* update/refresh the local collection. By this incremental update, the crawler refreshes existing pages and replaces "less-important" pages with new and "more-important" pages. When the crawler operates in this mode, we call it an *incremental crawler*.

In principle, the incremental crawler can be more effective than the periodic one. For instance, if the crawler can estimate how often pages change, the incremental crawler may revisit only the pages that have changed (with high probability), instead of refreshing the entire collection altogether. This optimization may result in substantial savings in network bandwidth and significant improvement in the "freshness" of the collection. Also, the incremental crawler may index/collect a new page in a more timely manner than the periodic crawler does. That is, the periodic crawler can index a new page only after the next crawling cycle starts, but the incremental crawler may immediately index the new page, right after it is found. Given the importance of web search engines (and thus web crawlers), even minor improvement in these areas may enhance the users' experience quite significantly.

Clearly, the effectiveness of crawling techniques heavily depends on how web pages change over time. If most web pages change at similar frequencies, the periodic and the incremental crawlers may be equally effective, because both crawlers in fact revisit all pages at the *same* frequencies. Also, if the web is quite static and only a small number of pages appear/disappear every month, the issue of how fast new pages are brought in may be of negligible importance to most users.

In this paper we will study how we can construct an effective incremental crawler. To that end, we first study how the web evolves over time, through an experiment conducted on more than half million web pages for more than 4 months. Based on these results, we then compare various design choices for a crawler, discussing how these choices affect the crawler's effectiveness. Through this discussion, we will also compare relative advantages/disadvantages of a periodic and an incremental crawler. Finally, we propose an architecture for an incremental crawler, which combines the best design choices.

In summary, our paper makes the following contribution:

- We study how web pages evolve over time, by an experiment conducted on 720,000 web pages for multiple months (Sections 2 and 3). We use our operational WebBase crawler for this experiment. (An earlier version of this crawler was used for the Google search engine [PB98].)
- We identify various design choices for an incremental crawler, and using our experimental data, we *quantify* the impact of various choices (Section 4). Our results let us make more informed decisions on the structure of a crawler.
- Based on our observations, we propose an architecture for an incremental crawler, which maintains only "important" pages and adjusts revisit frequency for pages depending on how often they change (Section 5). This architecture is driving the new version of our WebBase crawler.

2 Experimental setup

Our initial experiment tries to answer the following questions about the evolving web:

- How often does a web page change?
- What is the lifespan of a page?
- How long does it take for 50% of the web to change?
- Can we describe changes of web pages by a mathematical model?

Note that an incremental crawler itself also has to answer some of these questions. For instance, the crawler has to estimate how often a page changes, in order to decide how often to revisit the page. The techniques used for our experiment will shed a light on how an incremental crawler should operate and which statistics-gathering mechanisms it should adopt.

To answer our questions, we crawled around 720,000 pages from 270 sites every day, from February 17th through June 24th, 1999. This was done with the Stanford WebBase crawler, a system designed to create and maintain large web repositories (currently 290GB of HTML is stored). In this section we briefly discuss how the particular sites and pages were selected.

2.1 Monitoring technique

For our experiment, we adopted an *active crawling* approach with a *page window*. With active crawling, a crawler visits pages of interest periodically to see if they have changed. This is in contrast to a passive scheme, where say a proxy server tracks the fraction of new pages it sees, driven by the demand of its local users. A passive scheme is less obtrusive, since no additional load is placed on web servers beyond what would naturally be placed. However, we use active crawling because it lets us collect much better statistics, i.e., we can determine what pages to check and how frequently.

The pages to actively crawl are determined as follows. We start with a list of root pages for sites of interest. We periodically revisit these pages, and visit some predetermined number of pages that are reachable, breadth first, from that root. This gives us a *window of pages* at each site, whose contents may vary from visit to visit. Pages may leave the window if they are deleted or moved deeper within the site. Pages may also enter the window, as they are created or moved closer to the root. Thus, this scheme is superior to one that simply tracks a fixed set of pages, since such a scheme would not capture new pages.

We considered a variation of the page window scheme, where pages that disappeared from the window would still be tracked, if they still exist elsewhere in the site. This scheme could yield slightly better statistics on the lifetime of pages. However, we did not adopt this variation because it forces us to crawl a growing number of pages at each site. As we discuss in more detail below, we very much wanted to bound the load placed on web servers throughout our experiment.

2.2 Site selection

To select the actual sites for our experiment, we used the snapshot of 25 million web pages in our WebBase repository. Based on this snapshot, we identified top 400 "popular" sites as the candidate sites (The definition of the "popular" site is given below.). Then, we contacted the webmasters of all candidate sites to get their permission for our experiment. After this step, 270 sites remained, including sites such as Yahoo (`http://yahoo.com`), Microsoft (`http://microsoft.com`), and Stanford (`http://www.stanford.edu`). Obviously, focusing on the "popular" sites biases our results to a certain degree, but we believe this bias is toward what most people are interested in.

domain	number of sites
`com`	132
`edu`	78
`netorg`	30 (`org`: 19, `net`: 11)
`gov`	30 (`gov`: 28, `mil`: 2)

Table 1: Number of sites within a domain

To measure the popularity of a site, we used modified PageRank metric [PB98]. Informally, the PageRank metric considers a page "popular" if it is linked to by many other web pages. More precisely, the PageRank of page P, $PR(P)$, is defined by

$$PR(P) = d + (1-d)[PR(P_1)/c_1 + ... + PR(P_n)/c_n]$$

where $P_1,\ldots,P_n$ are the pages pointing to P, and $c_1,\ldots,c_n$ are the number of links going out from pages $P_1,\ldots,P_n$, and d is a damping factor, which was 0.9 in our experiment. This leads to one equation per web page, with an equal number of unknowns. The equations can be solved for the PR values iteratively, starting with all PR values equal to 1. At each step, the new $PR(P)$ value is computed from the old $PR(P_i)$ values (using the equation above), until the values converge. Intuitively, $PR(P)$ value gives us the probability that "random web surfer" is at P at a given time.

Since the PageRank computes the popularity of web *pages* not web *sites*, we need to slightly modify the definition of the PageRank for web sites. To do that, we first construct a hypergraph, where the nodes correspond to the web *sites* and the edges correspond to the links between the *sites*. Then for this hypergraph, we can define PR value for each node (site) using the same formula above. The value for a site then gives us the measure of the popularity of the web site.

In Table 1, we show how many sites in our list are from which domain. In our site list, 132 sites belong to `com` and 78 sites to `edu`. The sites ending with ".net" and ".org" are classified as `netorg` and the sites ending with ".gov" and ".mil" as `gov`.

2.3 Number of pages at each site

After selecting the web sites to monitor, we still need to decide the window of pages to crawl from each site. In our experiment, we crawled 3,000 pages at each site. That is, starting from the root pages of the selected sites we followed links in a breadth-first search, up to 3,000 pages per site. This "3,000 page window" was decided for practical reasons. In order to minimize the load on a site, we ran the crawler only at night (9PM through 6AM PST), waiting at least 10 seconds between requests to a single site. Within these constraints, we could crawl at most 3,000 pages from a site every day.

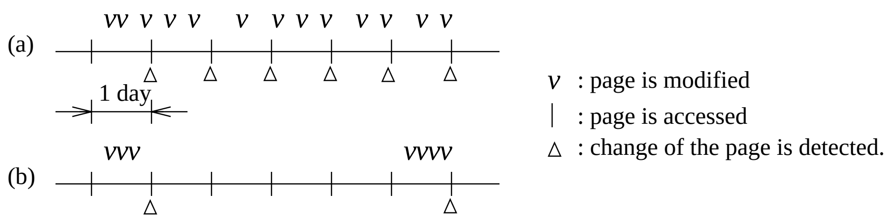


Figure 1: The cases when the estimated change interval is lower than the real value

3 Results

From the experiment described in the previous section, we collected statistics on how often pages change and how long they stay on the web, and we report the result in this section.

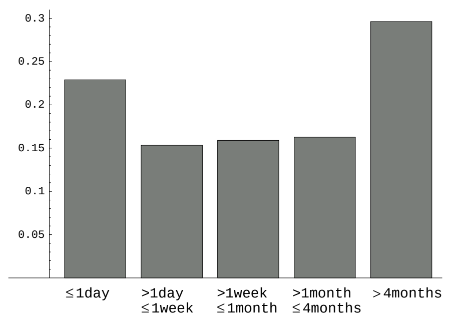
3.1 How often does a page change?

Based on the data that we collected, we can analyze how long it takes for a web page to change. For example, if a page existed within our window for 50 days, and if the page changed 5 times in that period, we can estimate the *average change interval* of the page to be 50 days/5 = 10 days. Note that the granularity of the estimated change interval is one day, because we can detect at most one change per day, even if the page changes more often (Figure 1(a)). Also, if a page changes several times a day and then remains unchanged, say, for a week (Figure 1(b)), the estimated interval might be much longer than the true value. In this case, however, we can interpret our estimation as the interval between the *batches of changes*, which might be more meaningful than the average interval of change.

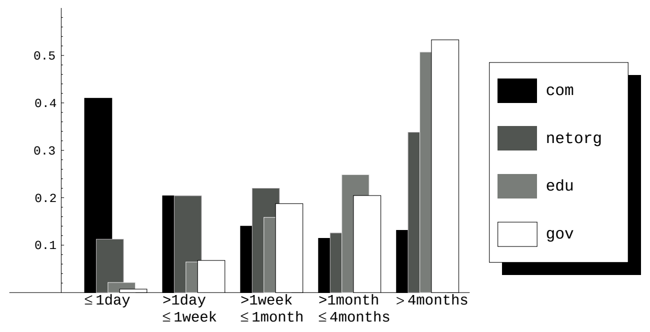
In Figure 2 we summarize the result of this analysis. In the figure, the horizontal axis represents the average change interval of pages, and the vertical axis shows the fraction of pages changed at the given average interval. Figure 2(a) shows the statistics collected over all domains, and Figure 2(b) shows the statistics broken down to each domain. For instance, from the second bar of Figure 2(a) we can see that 15% of the pages have a change interval longer than a day and shorter than a week.

From the first bar of Figure 2(a), we can observe that a surprisingly large number of pages change at very high frequencies: More than 20% of pages had changed whenever we visited them! As we can see from Figure 2(b), these frequently updated pages are mainly from the `com` domain. More than 40% of pages in the `com` domain changed every day, while less than 10% of the pages in other domains changed at that frequency (Figure 2(b) first bars). In particular, the pages in `edu` and `gov` domain are very static. More than 50% of pages in those domains did not change at all for 4 months (Figure 2(b) fifth bars). Clearly, pages at commercial sites, maintained by professionals, are updated frequently to provide timely information and attract more users.

Note that it is not easy to estimate the *average* change interval over all web page, because we conducted the experiment for a limited period. While we know how often a page changes if its change interval is longer than one day and shorter than 4 months, we do not know exactly



(a) Over all domains



(b) For each domain

Figure 2: Fraction of pages with given average interval of change

how often a page changes, when its change interval is out of this range (the pages corresponding to the first or the fifth bar of Figure 2(a)). However, if we assume that the pages in the first bar change every day and the pages in the fifth bar change every year (as a crude approximation), the overall average change interval of a web page is about 4 months.

In summary, web pages change rapidly overall, and the actual rates vary dramatically from site to site. Thus, a good crawler that is able to effectively track all these changes will be able to provide much better data than one that is not sensitive to changing data.

3.2 What is the lifespan of a page?

In this subsection we study how long we can access a particular page, once it appears on the web. To address this question, we investigated how long we could detect each page during our experiment. That is, for every page that we crawled, we checked how many days the page was accessible within our window (regardless of whether the page content had changed), and used that number as the *visible lifespan* of the page. Note that the *visible* lifespan of a page is not the same as its *actual* lifespan, because we measure how long the page was visible *within* our window. However, we believe the visible lifespan is a close approximation to the lifespan of a page *conceived by users* of the web. That is, when a user looks for an information from a particular site, she often starts from its root page and follows links. Since the user cannot infinitely follow links, she concludes the page of interest does not exist or has disappeared, if the page is not reachable within a few links from the root page. Therefore, many users often look at only a *window* of pages from a site, not the entire site.

Because our experiment was conducted in a limited time period, measuring the visible lifespan of a page is not as straightforward as we just described. Figure 3 illustrates the problem in detail. For the pages that appeared *and* disappeared during our experiment (Figure 3(b)), we can measure how long the page stayed in our window precisely. However, for the pages that existed from the beginning (Figure 3(a) and (d)) or at the end of our experiment (Figure 3(c) and (d)), we do not know exactly how long the page was in our window, because we do not know when the page appeared/disappeared. To take this error into account, we estimated the visible

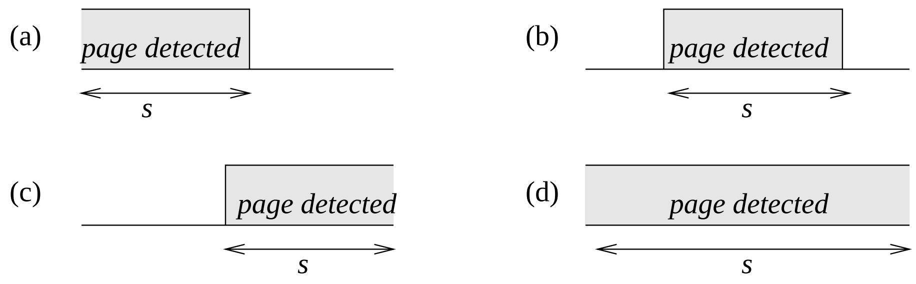


Figure 3: Issues in estimating the lifespan of a page

lifespan in two different ways. First, we used the length s in Figure 3 as the lifespan of a page (Method 1), and second, we assumed that the lifespan is $2s$ for pages corresponding to (a), (c) and (d) (Method 2). Clearly, the lifespan of (a), (c) and (d) pages can be anywhere between s and infinity, but we believe $2s$ is a reasonable guess, which gives an *approximate* range for the lifespan of pages.

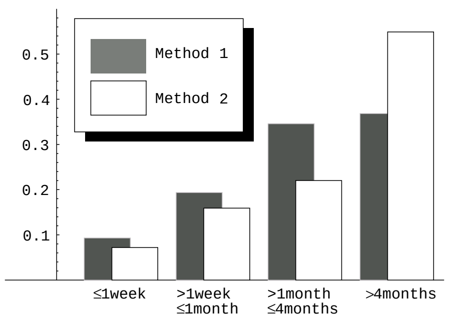
Figure 4(a) shows the result estimated by the two methods. In the figure, the horizontal axis shows the visible lifespan and the vertical axis shows the fraction of pages with given lifespan. For instance, from the second bar of Figure 4(a), we can see that Method 1 estimates that around 19% of the pages have a lifespan of longer than one week and shorter than 1 month, and Method 2 estimates that the fraction of the corresponding pages is around 16%. Note that Methods 1 and 2 give us similar numbers for the pages with a short lifespan (the first and the second bar), but their estimates are very different for longer lifespan pages (the third and fourth bar). This result is because the pages with a longer lifespan have higher probability of spanning over the beginning or the end of our experiment and their estimates can be different by a factor of 2 for Method 1 and 2. In Figure 4(b), we show the lifespan of pages for different domains. To avoid cluttering the graph, we only show the histogram obtained by Method 1.

Interestingly, we can see that a significant number of pages are accessible for a relatively long period. More than 70% of the pages over all domains remained in our window for more than one month (Figure 4(a), the third and the fourth bars), and more than 50% of the pages in the `edu` and `gov` domain stayed for more than 4 months (Figure 4(b), fourth bar). As expected, the pages in the `com` domain were the shortest lived, and the pages in the `edu` and `gov` domain lived the longest.

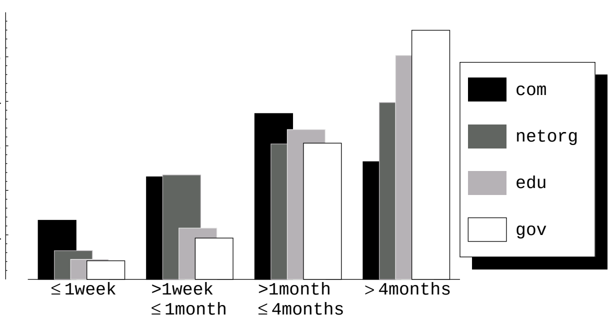
3.3 How long does it take for 50% of the web to change?

In the previous subsections, we mainly focused on how an *individual* web page evolves over time. For instance, we studied how often a page changes, and how long it stays within our window. Now we slightly change our perspective and study how the *web as a whole* evolves over time. That is, we investigate how long it takes for p% of the pages within our window to change.

To get this information, we traced how many pages in our window remained unchanged after a certain period, and the result is shown in Figure 5. In the figure, the horizontal axis shows the number of days from the beginning of the experiment and the vertical axis shows the fraction

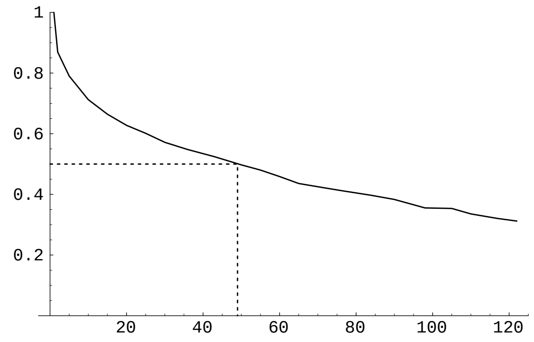


(a) Over all domains

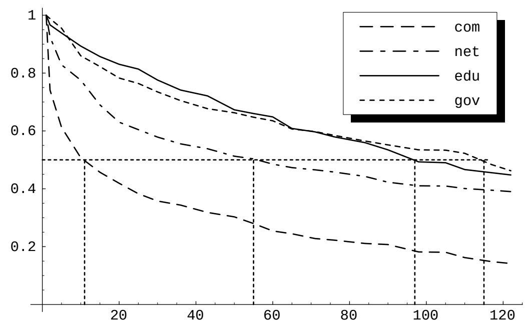


(b) For each domain

Figure 4: Percentage of pages with given visible lifespan



(a) Over all domains



(b) For each domain

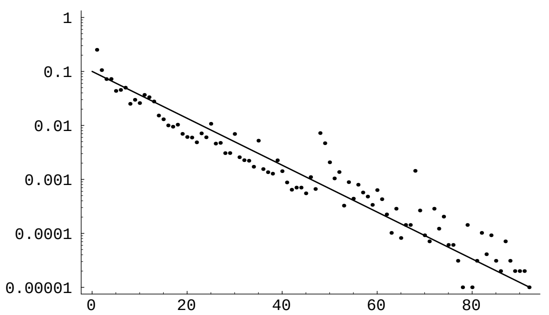
Figure 5: Fraction of pages that did not change or disappear until given date.

of pages that were unchanged by the given day.

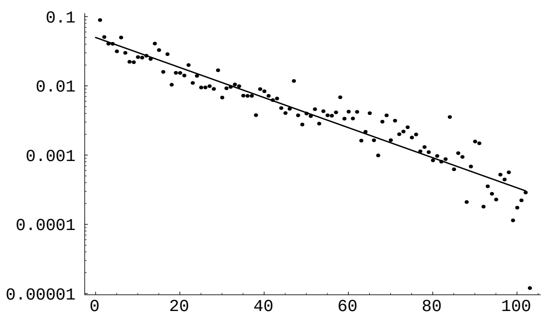
From Figure 5(a), we can see that it takes about 50 days for 50% of the web to change or to be replaced by new pages. From Figure 5(b), we can confirm that different domains evolve at highly different rates. For instance, it took only 11 days for 50% of the `com` domain to change, while the same amount of change took almost 4 months for the `gov` domain (Figure 5(b)). Similarly to the previous results, the `com` domain is the most dynamic, followed by the `netorg` domain. The `edu` and the `gov` domains are the most static. Again, our results highlight the need for a crawler that can track these massive but skewed changes effectively.

3.4 Can we describe changes of a page by a mathematical model?

Now we study whether we can describe changes of web pages by a mathematical model. In particular, we study whether changes of web pages follow a *Poisson process*. Building a change model of the web is very important, in order to compare how effective different crawling policies are. For instance, if we want to compare how "fresh" crawlers maintain their local collections, we



(a) For the pages that change every 10 days on average



(b) For the pages that change every 20 days on average

Figure 6: Change intervals of pages

need to compare how many pages in the collection are maintained up-to-date, and this number is hard to get without a proper change model for the web.

A Poisson process is often used to model a sequence of *random* events that happen *independently* with *fixed rate* over time. For instance, occurrences of fatal auto accidents, arrivals of customers at a service center, telephone calls originating in a region, etc., are usually modeled by a Poisson process. We believe a Poisson process is a good model for changes of web pages, because many web pages have the properties that we just mentioned. For instance, pages in the CNN web site change at the *average* rate of once a day, but the change of a particular page is quite random, because update of the page depends on how the news related to that page develops over time.

Under a Poisson process, we can compute the time between two events. To compute this interval, let us assume that the first event happened at time 0, and let T be the time when the next event occurs. Then the probability density function of T is given by the following theorem [TK98].

Theorem 1 *If T is the time to the occurrence of the next event in a* Poisson process *with rate λ, the probability density function for T is*

$$f_T(t) = \begin{cases} \lambda e^{-\lambda t} & \text{for } t > 0 \\ 0 & \text{for } t \leq 0 \end{cases} \qquad \square$$

We can use Theorem 1 to verify whether web changes follow a Poisson process. That is, if changes to a page follow a Poisson process of rate λ, its change intervals should follow the distribution $\lambda e^{-\lambda t}$. To compare this prediction to our experimental data, we assume that each page p_i on the web has an *average* rate of change λ_i, where λ_i may differ from page to page. Then we select only the pages whose *average change intervals* are, say, 10 days and plot the distribution of their change intervals. If the pages indeed follow a Poisson process, this graph should be distributed exponentially. In Figure 6, we show some of the graphs plotted this way. Figure 6(a) is the graph for the pages with 10 day change interval, and Figure 6(b) is for the pages with 20 day change interval. The horizontal axis represents the interval between successive changes, and the vertical axis shows the fraction of changes with that interval. The vertical axis in the graph is logarithmic to emphasize that the distribution is exponential. The lines in the graphs are the predictions by a Poisson process. While there exist small variations, we can clearly see that a Poisson process predicts the observed data very well. We also plotted the same graph for the pages with other change intervals and got similar results when we had sufficient data.

Although our results indicate that a Poisson process describes the web page changes very well, they are limited due to the constraint of our experiment. We crawled web pages on a daily basis, so our result does not verify the Poisson model for the pages that change very often. Also, the pages that change very slowly were not verified either, because we conducted our experiment for four months and did not detect any changes to those pages. However, we believe that most crawlers may not have high interest in learning exactly how often those pages change. For example, the crawling interval of most crawlers is much longer than a day, so they do not particularly care whether a page changes exactly once every day or more than once every day.

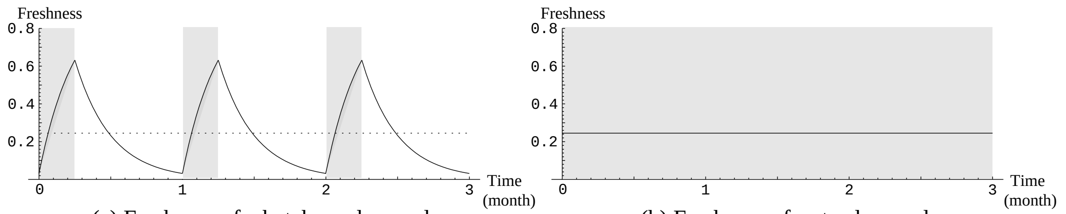
Also, a set of web pages may be updated at a regular interval, and their changes may not necessarily follow a Poisson process. However, a crawler cannot easily identify these pages when it maintains hundreds of millions of web pages, so the entire set of pages that the crawler manages may be considered to change by a random process on average. Thus, we believe it is safe to use the Poisson model to compare crawler strategies in the next section.

4 Crawler design issues

The results of previous section showed us how web pages change over time. Based on these results, we now discuss various design choices for a crawler and their possible trade-offs. One of our central goals is to maintain the local collection up-to-date. To capture how "fresh" a collection is, we will use the metric *freshness* in [CGM00b]. Informally, freshness represents the fraction of "up-to-date" pages in the local collection. For instance, when all pages in the collection are up-to-date (i.e., the same as the *current state* of their real-world counterparts), the freshness of the collection is 1, while the freshness of the collection is 0.5 when a half of the collection is up-to-date. (In [CGM00b] we also discuss a second metric, the "age" of crawled pages. This metric can also be used to compare crawling strategies, but the conclusions are not significantly different from the ones we reach here using the simpler metric of freshness.)

1. **Is the collection updated in batch-mode?** A crawler needs to revisit web pages in order to maintain the local collection up-to-date. Depending on how the crawler updates its collection, the crawler can be classified as one of the following:

   - **Batch-mode crawler:** A *batch-mode crawler* runs *periodically* (say, once a month), updating *all* pages in the collection in each crawl. We illustrate how such a crawler operates in Figure 7(a). In the figure, the horizontal axis represents time and the grey



(a) Freshness of a batch-mode crawler (b) Freshness of a steady crawler

Figure 7: Freshness evolution of a batch-mode/steady crawler

region shows when the crawler operates. The vertical axis in the graph represents the freshness of the collection, and the curve in the graph shows how freshness changes over time. The dotted line shows freshness *averaged over time*. The curves in this section are obtained analytically using a Poisson model. (We do not show the derivation here due to space constraints.) We use a high page change rate to obtain curves that more clearly show the trends. Later on we compute freshness values based on the actual rate of change we measured on the web.

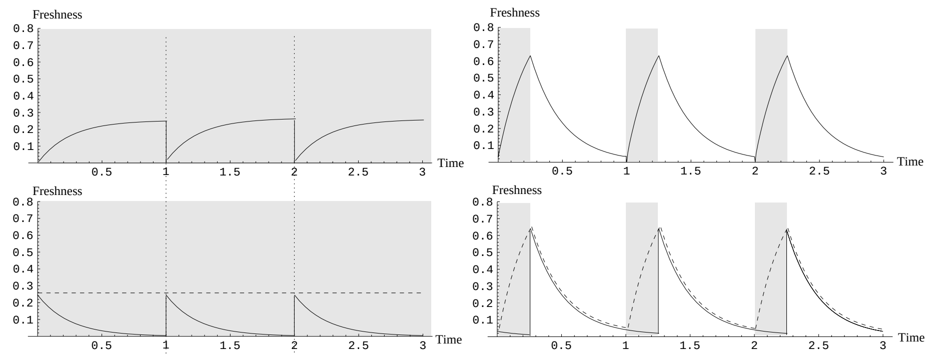
To plot the graph, we also assumed that the crawled pages are immediately made available to users, as opposed to making them all available at the end of the crawl. From the figure, we can see that the collection starts growing stale when the crawler is idle (freshness decreases in white regions), and the collection gets fresher when the crawler revisits pages (freshness increases in grey regions).

Note that the freshness is not equal to 1 even at the end of each crawl (the right ends of grey regions), because some pages have already changed during the crawl. Also note that the freshness of the collection decreases exponentially in the white region. This trend is consistent with the experimental result of Figure 5.

- **Steady crawler:** A *steady crawler* runs continuously without any pause (Figure 7(b)). In the figure, the entire area is grey, because the crawler runs continuously. Contrary to the batch-mode crawler, the freshness of the steady crawler is stable over time because the collection is continuously and incrementally updated.

While freshness evolves differently for the batch-mode and the steady crawler, one can *prove* (based on the Poisson model) that their freshness *averaged over time* is the *same*, if they visit pages at the same *average* speed. That is, when the steady and the batch-mode crawler revisit all pages every month (even though the batch-mode crawler finishes a crawl in a week), the freshness averaged over time is the same for both.

Even though both crawlers yield in the same average freshness, the steady crawler has an advantage over the batch-mode one, because it can collect pages at a lower *peak* speed. To get the same average speed, the batch-mode crawler must visit pages at a higher speed when it operates. This property increases the peak load on the crawler's local machine and on the network. From our crawling experience, we learned that the peak crawling speed is a *very* sensitive issue for many entities on the web. For instance, when the WebBase crawler ran at a very high speed, it once crashed the central router for the Stanford network. After



(a) Freshness of a steady crawler with shadowing (b) Freshness of a batch-mode crawler with shadowing

Figure 8: Freshness of the crawler's and the current collection

that incident, Stanford network managers have closely monitored our crawling activity to ensure it runs at a reasonable speed. Also, the webmasters of many web sites carefully trace how often a crawler accesses their sites. If they feel a crawler runs too fast, they sometimes block the crawler completely from accessing their sites.

2. **Is the collection updated in-place?** When a crawler replaces an old version of a page with a new one, it may update the page *in-place*, or it may perform *shadowing* [MJLF84]. With shadowing, a new set of pages is collected from the web, and stored in a *separate space* from the current collection. After all new pages are collected and processed, the current collection is instantaneously replaced by this new collection. To distinguish, we refer to the collection in the shadowing space as the *crawler's collection*, and the collection that is currently available to users as the *current collection*.

   Shadowing a collection may improve the availability of the current collection, because the current collection is completely shielded from the crawling process. Also, if the crawler's collection has to be pre-processed before it is made available to users (e.g., an indexer may need to build an inverted-index), the current collection can still handle users' requests during this period. Furthermore, it is probably easier to implement shadowing than in-place updates, again because the update/indexing and the access processes are separate.

   However, shadowing a collection may decrease freshness. To illustrate this issue, we use Figure 8. In the figure, the graphs on the top show the freshness of the crawler's collection, while the graphs at the bottom show the freshness of the current collection. To simplify our discussion, we assume that the current collection is instantaneously replaced by the crawler's collection right after all pages are collected.

   When the crawler is steady, the freshness of the crawler's collection will evolve as in Figure 8(a), top. Because a new set of pages are collected from scratch say every month, the freshness of the crawler's collection increases from zero every month. Then at the end of each month (dotted lines in Figure 8(a)), the current collection is replaced by the crawler's collection, making their freshness the same. From that point on, the freshness of the current collection decreases, until the current collection is replaced by a new set of

	Steady	Batch-mode
In-place	0.88	0.88
Shadowing	0.77	0.86

Table 2: Freshness of the collection for various choices

pages. To compare how freshness is affected by shadowing, we show the freshness of the current collection *without shadowing* as a dashed line in Figure 8(a), bottom. The dashed line is always higher than the solid curve, because when the collection is not shadowed, new pages are immediately made available. Freshness of the current collection is always higher *without* shadowing.

In Figure 8(b), we show the freshness of a *batch-mode* crawler when the collection is shadowed. The solid line in Figure 8(b) top shows the freshness of the crawler's collection, and the solid line at the bottom shows the freshness of the current collection. For comparison, we also show the freshness of the current collection *without shadowing* as a dashed line at the bottom. (The dashed line is slightly shifted to the right, to distinguish it from the solid line.) The grey regions in the figure represent the time when the crawler operates.

At the beginning of each month, the crawler starts to collect a new set of pages from scratch, and the crawl finishes in a week (the right ends of grey regions). At that point, the current collection is replaced by the crawler's collection, making their freshness the same. Then the freshness of the current collection decreases exponentially until the current collection is replaced by a new set of pages.

Note that the dashed line and the solid line in Figure 8(b) bottom, are the same most of the time. For the batch-mode crawler, freshness is mostly the same, regardless of whether the collection is shadowed or not. Only when the crawler is running (grey regions), the freshness of the *in-place update* crawler is higher than that of *shadowing* crawler, because new pages are immediately available to users with the in-place update crawler.

In Table 2 we contrast the four possible choices we have discussed (shadowing versus in-place, and steady versus batch), using the change rates measured in our experiment. To construct the table, we assumed that all pages change with an *average* 4 month interval, based on the result of Section 3.1. Also, we assumed that the steady crawler revisits pages steadily over a month, and that the batch-mode crawler recrawls pages only in the first week of every month. The entries in Table 2 give the expected freshness of the current collection. From the table, we can see that the freshness of the steady crawler significantly decreases with shadowing, while the freshness of the batch-mode crawler is not much affected by shadowing. Thus, if one is building a batch crawler, shadowing is a good option since it is simpler to implement, and in-place updates are not a significant win in this case. In contrast, the gains are significant for a steady crawler, so in-place updates may be a good option.

Note that, however, this conclusion is very sensitive to how often web pages change and how often a crawler runs. For instance, consider a scenario where web pages change every

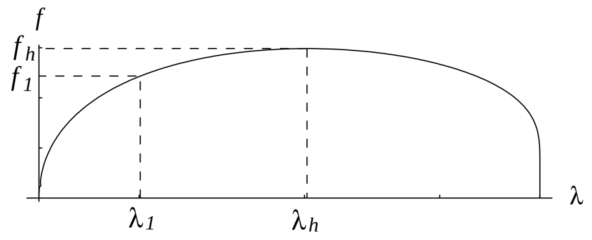


Figure 9: Change frequency of a page vs. optimal revisit frequency of the page

month (as opposed to every 4 months), and a batch crawler operates for the first two weeks of every month. Under these parameters, the freshness of a batch crawler with in-place updates is 0.63, while the freshness is 0.50 with a shadowing crawler. Therefore, if a crawler focuses on a dynamic portion of the web (e.g., `com` domain), the crawler may need to adopt the in-place update policy, even when it runs in batch mode.

3. **Are pages refreshed at the same frequency?** As the crawler updates pages in the collection, it may visit the pages either at the same frequency or at different frequencies.

   - **Fixed frequency:** The crawler revisits web pages at the same frequency, regardless of how often they change. We believe this fixed-frequency policy matches well with the batch-mode crawler, since it commonly revisits all pages in the collection in every crawl.
   - **Variable frequency:** The result of Section 3.1 showed that web pages change at widely different frequencies. Given this result, the crawler may optimize the *revisit frequency* for a page, based on how often the page changes. Note that the variable-frequency policy is well suited for the *steady* crawler with *in-place updates.* Since the steady crawler visits pages continuously, it can adjust the revisit frequency with arbitrary granularity and thus increase the freshness of the collection.

If a variable frequency is used, the crawler needs a strategy for deciding at what rate to visit each page. Intuitively, one may suspect that the crawler should revisit a page more often, when it changes more often. However, reference [CGM00b] shows that this intuition may not be right, depending on the freshness metric used. For instance, Figure 9 shows how often a crawler should visit a page, to optimize the freshness metric [CGM00b]. The horizontal axis represents the change frequency of a page, and the vertical axis shows the optimal revisit frequency for that page. For example, if a page in the collection changes at the frequency λ_1, the crawler should visit the page at the frequency f_1. (We do not show specific numbers in the graph, because the scale of the graph depends on how often pages change and how often the crawler revisits the pages. However, the *shape* of the graph is always the same regardless of the scenario. For details, see [CGM00b].) Note that when a page changes at a low frequency ($\lambda < \lambda_h$), the crawler should visit the page more often as it changes more often (f increases as λ increases). However, when the page changes at a high frequency ($\lambda > \lambda_h$), the crawler should visit the page less often as it changes more

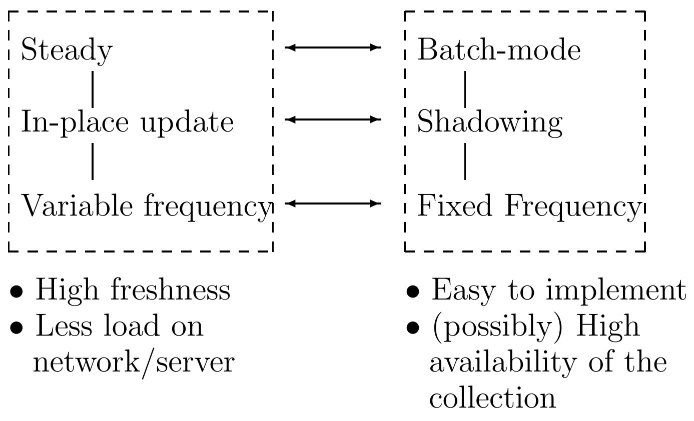


Figure 10: Two possible crawlers and their advantages

often (f decreases as λ increases).

We can understand this unexpected result through the following simple example. Suppose that a crawler maintains two pages, p_1 and p_2, in its collection. Also suppose that page p_1 changes every day and page p_2 changes every second. Due to bandwidth limitations, the crawler can crawl only one page per day, and it has to decide which page to crawl. Probabilistically, if the crawler revisits page p_1, p_1 will remain up-to-date for a half of the day. Therefore, the freshness of the collection will be 0.5 for a half of the day. (One out of two pages remain up-to-date for a half of the day.) Instead, if the crawler revisits page p_2, p_2 will remain up-to-date for a half second, so the freshness will be 0.5 only for a half second. Clearly, it is better to visit p_1 (which changes less often than p_2), than to visit p_2! From this example, we can see that the optimal revisit frequency is not always proportional to the change frequency of a page. The optimal revisit frequency depends on how often pages change and how often the crawler revisits pages, and it should be carefully determined. In reference [CGM00b], we study this problem in more detail. The reference shows that one can increase the freshness of the collection by 10%–23% by optimizing the revisit frequencies.

We summarize the discussion of this section in Figure 10. As we have argued, there exist two "reasonable" combinations of options, which have different advantages. The crawler on the left gives us high freshness and results in low peak loads. The crawler on the right may be easier to implement and interferes less with a highly utilized current collection. The left-hand side corresponds to the *incremental crawler* we discussed in the introduction, and the right-hand side corresponds to the *periodic crawler*. In the next section, we discuss how we can implement an effective incremental crawler, with the properties listed on the left-hand side of the diagram.

5 Architecture for an incremental crawler

In this section, we study how to implement an effective incremental crawler. To that end, we first identify two goals for the incremental crawler and explain how the incremental crawler conceptually operates. From this operational model, we will identify two key decisions that an incremental crawler constantly makes. Based on these observations, we propose an architecture for the incremental crawler.

5.1 Two goals for an incremental crawler

The incremental crawler continuously crawls the web, revisiting pages periodically. During its continuous crawl, it may also purge some pages in the local collection, in order to make room for newly crawled pages. During this process, the crawler should have two goals:

1. **Keep the local collection fresh:** Our results showed that freshness of a collection can vary widely depending on the strategy used. Thus, the crawler should use the best policies to keep pages fresh. This includes adjusting the revisit frequency for a page based on its *estimated* change frequency.

2. **Improve quality of the local collection:** The crawler should increase the "quality" of the local collection by replacing "less important" pages with "more important" ones. This refinement process is necessary for two reasons. First, our result in Section 3.2 showed that pages are constantly created and removed. Some of the new pages can be "more important" than existing pages in the collection, so the crawler should replace the old and "less important" pages with the new and "more important" pages. Second, the importance of existing pages change over time. When some of the existing pages become less important than previously ignored pages, the crawler should replace the existing pages with the previously ignored pages.

5.2 Operational model of an incremental crawler

In Figure 11 we show pseudo-code that describes how an incremental crawler operates. This code shows the *conceptual* operation of the crawler, not an efficient or complete implementation. (In Section 5.3, we show how an actual incremental crawler operates.) In the algorithm, AllUrls records the set of *all* URLs discovered, and CollUrls records the set of URLs in the collection. To simplify our discussion, we assume that the local collection maintains a *fixed number* of pages[1] and that the collection is at its maximum capacity from the beginning. In Step [2] and [3], the crawler selects the next page to crawl and crawls the page. If the page already exists in the collection (the condition of Step [4] is true), the crawler updates its image in the collection (Steps [5]). If not, the crawler discards an existing page from the collection (Steps [7] and [8]), saves the new page (Step [9]) and updates CollUrls (Step [10]). Finally, the crawler extracts links (or URLs) in the crawled page to add them to the list of all URLs (Steps [11] and [12]).

Note that the crawler makes decisions in Step [2] and [7]. In Step [2], the crawler decides on what page to crawl, and in Step [7] the crawler decides on what page to discard. However, note that the decisions in Step [2] and [7] are intertwined. That is, when the crawler decides to

[1]It might be more realistic to assume that the *size* of the collection is fixed, but we believe the *fixed-number* assumption is a good approximation to the *fixed-size* assumption, when the number of pages in the collection is large.

```
Algorithm 5.1 Operation of an incremental crawler
Input   AllUrls: a set of all URLs known
        CollUrls: a set of URLs in the local collection
        (We assume CollUrls is full from the beginning.)
Procedure
  [1] while (true)
  [2]     url ← selectToCrawl(AllUrls)
  [3]     page ← crawl(url)
  [4]     if (url ∈ CollUrls) then
  [5]         update(url, page)
  [6]     else
  [7]         tmpurl ← selectToDiscard(CollUrls)
  [8]         discard(tmpurl)
  [9]         save(url, page)
  [10]        CollUrls ← (CollUrls − {tmpurl}) ∪ {url}
  [11]    newurls ← extractUrls(page)
  [12]    AllUrls ← AllUrls ∪ newurls
```

Figure 11: Conceptual operational model of an incremental crawler

crawl a new page, it *has to* discard a page from the collection to make room for the new page. Therefore, when the crawler decides to crawl a new page, the crawler should decide what page to discard. We refer to this selection/discard decision as the *refinement decision*.

Note that this refinement decision should be based on the "importance" of pages. To measure importance, the crawler can use a number of metrics, including PageRank [CGMP98, PB98] and Hub and Authority [Kle98]. Clearly, the importance of the discarded page should be lower than the importance of the new page. In fact, the discarded page should have the *lowest* importance in the collection, to maintain the collection of the highest quality.

Together with the refinement decision, the crawler decides on what page to *update* in Step [2]. That is, instead of visiting a new page, the crawler may decide to visit an existing page to refresh its image. To maintain the collection "fresh," the crawler has to select the page that will increase the freshness most significantly, and we refer to this decision as *update decision*.

5.3 Architecture for an incremental crawler

To achieve the two goals for incremental crawlers, and to effectively implement the corresponding decision process, we propose the architecture for an incremental crawler shown in Figure 12. The architecture consists of three major modules (`RankingModule`, `UpdateModule` and `CrawlModule`) and three data structures (AllUrls, CollUrls and Collection). The lines and arrows show data flow between modules, and the labels on the lines show the corresponding commands. Two data structures, AllUrls and CollUrls, maintain information similar to that shown in Figure 11. AllUrls records *all* URLs that the crawler has discovered, and CollUrls records the URLs that are/will be in the Collection. CollUrls is implemented as a priority-queue, where the URLs to be crawled

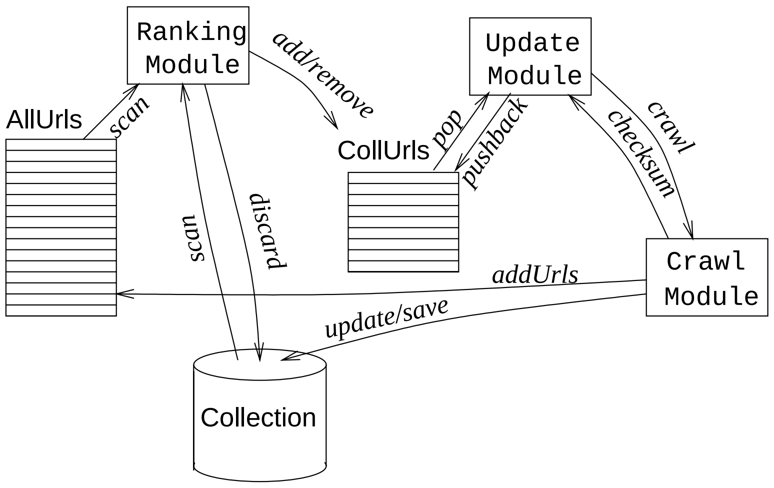


Figure 12: Architecture of the incremental crawler

early are placed in the front.

The URLs in CollUrls are chosen by the RankingModule. The RankingModule constantly scans through AllUrls and the Collection to make the *refinement decision*. For instance, if the crawler uses PageRank as its importance metric, the RankingModule constantly reevaluates the PageRanks of all URLs, based on the link structure captured in the Collection.[2] When a page *not* in CollUrls turns out to be more important than a page within CollUrls, the RankingModule schedules for replacement of the less-important page in CollUrls with that more-important page. The URL for this new page is placed on the top of CollUrls, so that the UpdateModule can crawl the page immediately. Also, the RankingModule discards the less-important page from the Collection to make space for the new page.

While the RankingModule refines the Collection, the UpdateModule maintains the Collection "fresh" (*update decision*). It constantly extracts the top entry from CollUrls, requests the CrawlModule to crawl the page, and puts the crawled URL back into CollUrls. The position of the crawled URL within CollUrls is determined by the page's *estimated* change frequency and its importance. (The closer a URL is to the head of the queue, the more frequently it will be revisited.)

To estimate how often a particular page changes, the UpdateModule records the checksum of the page from the last crawl and compares that checksum with the one from the current crawl. From this comparison, the UpdateModule can tell whether the page has changed or not. In our companion paper [CGM00a] (submitted to VLDB 2000, Research track), we explain how the UpdateModule can estimate the change frequency of a page based on this change history. In short, we propose two "estimators," E_P and E_B, for the change frequency of a page.

Estimator E_P is based on the Poisson process model verified in Section 3.4, while estimator E_B is based on a Bayesian inference method. Essentially, E_P is the same as the method described in Section 3.1. To implement E_P, the UpdateModule has to record how many times the crawler

[2]Note that even if a page p does not exist in the Collection, the RankingModule can estimate PageRank of p, based on how many pages in the Collection have a link to p.

detected changes to a page for, say, last 6 months. Then E_P uses this number to get a confidence interval for the change frequency of that page.

The goal of estimator E_B is slightly different from that of E_P. Instead of measuring a confidence interval, E_B tries to categorize pages into different frequency classes, say, pages that change every week (class C_W) and pages that change every month (class C_M). To implement E_B, the `UpdateModule` stores the probability that page p_i belongs to each frequency class ($P\{p_i \in C_W\}$ and $P\{p_i \in C_M\}$) and updates these probabilities based on detected changes. For instance, if the `UpdateModule` learns that page p_1 did not change for one month, the `UpdateModule` increases $P\{p_1 \in C_M\}$ and decreases $P\{p_1 \in C_W\}$. For details, see our companion paper [CGM00a].

Note that it is also possible to keep update statistics on larger units than a page, such as a web site or a directory. If web pages on a site change at similar frequencies, the crawler may trace how many times the pages on that site changed for last 6 months, and get a confidence interval based on the site-level statistics. In this case, the crawler may get a tighter confidence interval, because the frequency is estimated on *larger* number of pages (i.e., larger sample). However, if pages on a site change at highly different frequencies, this average change frequency may not be sufficient to determine how often to revisit pages in that site, leading to a less-than optimal revisit frequency.

Also note that the `UpdateModule` may need to consult the "importance" of a page in deciding on revisit frequency. If a certain page is "highly important" and the page needs to be always up-to-date, the `UpdateModule` may revisit the page much more often than other pages with similar change frequency. To implement this policy, the `UpdateModule` also needs to record the "importance" of each page.

Returning to our architecture, the `CrawlModule` crawls a page and saves/updates the page in the Collection, based on the request from the `UpdateModule`. Also, the `CrawlModule` extracts all links/URLs in the crawled page and forwards the URLs to AllUrls. The forwarded URLs are included in AllUrls, if they are new. While we show only one instance of the `CrawlModule` in the figure, note that multiple `CrawlModule`'s may run in parallel, depending on how fast we need to crawl pages.

Separating the update decision (`UpdateModule`) from the refinement decision (`RankingModule`) is crucial for performance reasons. For example, to visit 100 million pages every month,[3] the crawler has to visit pages at about 40 pages/second. However, it may take a while to select/deselect pages for Collection, because computing the importance of pages is often expensive. For instance, when the crawler computes PageRank, it needs to scan through the Collection multiple times, even if the link structure has changed little. Clearly, the crawler cannot recompute the importance of pages for every page crawled, when it needs to run at 40 pages/second. By separating the refinement decision from the update decision, the `UpdateModule` can focus on updating pages at high speed, while the `RankingModule` carefully refines the Collection.

[3]Many search engines report numbers similar to this.

6 Related Work

Several papers investigate how to build an effective crawler. Reference [CGMP98] studies what pages a crawler should visit, when it cannot store a complete web image. Reference [CvdBD99] looks at how to collect web pages related to a *specific* topic, in order to build a specialized web collection. The techniques discussed in these references can be used for the `RankingModule` in our architecture, to improve quality of the collection. In our companion paper [CGM00a], we study how to estimate the change frequency of a web page by revisiting the page periodically. References [CGM00b] and [CLW97] study how often a crawler should visit a page when it knows how often the page changes. The algorithms described in these references can be used for the `UpdateModule`, to improve freshness of the collection. We believe these references are complementary to our work, because we present an incremental-crawler architecture, which can use any of the algorithms in these papers.

References [WM99] and [DFK99] experimentally study how often web pages change. Reference [PP97] studies the relationship between the "desirability" of a page and its lifespan. However, none of these studies are as extensive as ours in terms of the scale and the length of the experiment. Also, their focus is different from ours. Reference [WM99] investigates page changes to improve *web caching policies*, and reference [PP97] studies how page changes are related to *access patterns*.

7 Conclusion

In this paper we have studied how to build an effective incremental crawler. To understand how the web evolves over time, we first described a comprehensive experiment, conducted on 720,000 web pages from 270 web sites over 4 months. Based on the results, we discussed various design choices for a crawler and the possible trade-offs. We then proposed an architecture for an incremental crawler, which combines the best strategies identified.

As the size of the web grows, it becomes more difficult to index the whole web (or a significant portion of it). This makes it imperative to build an effective crawler, which selectively chooses what pages to crawl and store. By adopting our incremental approach, we believe the crawler can improve the "freshness" and the "quality" of its index/collection significantly.

References

[CGM00a] Junghoo Cho and Hector Garcia-Molina. Estimating frequency of change, 2000. *Submitted to VLDB 2000, Research track.*

[CGM00b] Junghoo Cho and Hector Garcia-Molina. Synchronizing a database to improve freshness. In *Proceedings of the 2000 ACM SIGMOD*, 2000.

[CGMP98] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through URL ordering. In *Proceedings of the 7th World-Wide Web Conference*, 1998.

[CLW97] E.G. Coffman, Jr., Zhen Liu, and Richard R. Weber. Optimal robot scheduling for web search engines. Technical report, INRIA, 1997.

[CvdBD99] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused crawling: A new approach to topic-specific web resource discovery. In *Proceedings of the 8th World-Wide Web Conference*, 1999.

[DFK99] Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Rate of change and other metrics: a live study of the world wide web. In *USENIX Symposium on Internetworking Technologies and Systems*, 1999.

[Kle98] John M. Kleinberg. Authoritive sources in a hyperlinked environment. In *Proceedings of 9th ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[MJLF84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, 1984.

[PB98] Lawrence Page and Sergey Brin. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th World-Wide Web Conference*, 1998.

[PP97] James Pitkow and Peter Pirolli. Life, death, and lawfulness on the electronic frontier. In *Proceedings of International Conference on Computer and Human Interaction*, 1997.

[TK98] Howard M. Taylor and Samuel Karlin. *An Introduction To Stochastic Modeling.* Academic Press, 3rd edition, 1998.

[WM99] Craig E. Wills and Mikhail Mikhailov. Towards a better understanding of web resources and server responses for improved caching. In *Proceedings of the 8th World-Wide Web Conference*, 1999.