CRAWLING THE WEB: DISCOVERY AND MAINTENANCE OF LARGE-SCALE WEB DATA

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Junghoo Cho
November 2001

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Hector Garcia-Molina
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Jennifer Widom

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Andreas Paepcke

Approved for the University Committee on Graduate Studies:

Abstract

This dissertation studies the challenges and issues faced in implementing an effective Web crawler. A crawler is a program that retrieves and stores pages from the Web, commonly for a Web search engine. A crawler often has to download hundreds of millions of pages in a short period of time and has to constantly monitor and refresh the downloaded pages. In addition, the crawler should avoid putting too much pressure on the visited Web sites and the crawler's local network, because they are intrinsically shared resources.

This dissertation studies how we can build an effective Web crawler that can retrieve "high quality" pages quickly, while maintaining the retrieved pages "fresh." Towards that goal, we first identify popular definitions for the "importance" of pages and propose simple algorithms that can identify important pages at the early stage of a crawl. We then explore how we can parallelize a crawling process to maximize the download rate while minimizing the overhead from parallelization. Finally, we experimentally study how Web pages change over time and propose an optimal page refresh policy that maximizes the "freshness" of the retrieved pages.

This work has been a part of the WebBase project at Stanford University. The WebBase system currently maintains 130 million pages downloaded from the Web and these pages are being used actively by many researchers within and outside of Stanford. The crawler for the WebBase project is a direct result of this dissertation research.

Acknowledgments

First, I would like to thank my advisor, Hector Garcia-Molina, for being such a wonderful advisor. Through his support, care, and patience, he has transformed a struggling graduate student into an experienced researcher. His insight and ideas formed the foundation of this dissertation as much as mine did, and his guidance and care helped me get over various hurdles during my graduate years. It often amazes me how he can make himself available to every student when he works with 13 graduate students. I only hope that I will be able to practice half as much as what I learned from him after my graduation when I begin to work with students of my own.

Second, I thank my wife, Jihye Won, for her love and support. Frankly, it is quite tough to be the wife of a graduate student, but she went through the last 4 years without many complaints. I don't think I can ever make up for her lost sleep because of my typing at 4 in the morning, but now that she is a full-time student and I have a job, I hope that I will be able to give her as much support as I got during my graduate study.

Special thanks to Jennifer Widom and Andreas Paepcke for being the readers of my thesis. After reading a draft, they provided helpful comments on various aspects of my thesis. These comments made my thesis much better, and I learned a lot from their comments. Once I start doing my independent research, I will miss the comments that I could get from Jennifer, Andreas and Hector.

I would also like to thank other members of the Stanford database group, including Marianne Siroker, Andy Kacsmar, and Narayanan Shivakumar. Every day, Marianne ran this large group flawlessly, and Andy immediately recovered WebBase servers whenever I crashed them with my crawler. In my third year, I worked with Shiva very closely, and I learned how to shape and refine a crude idea into a research problem from him.

Sridhar Rajagopalan, Yoshinori Hara and Sougata Mukherjea were my mentors when

I was an intern at IBM and NEC. They also provided great ideas and a stimulating environment during the summer. I was very fortunate to meet such great people outside of Stanford.

Finally, I would like to thank my parents for their love, support and guidance. They firmly believe that I can be a great scholar, and whenever I am in doubt, they constantly remind me of their confidence and encourage me. Thinking back, it must have been a difficult decision for them to send their first son to a foreign country, and I am grateful for their selfless decision. One day I hope that our family will live somewhere nearby and we will all have a quiet and peaceful life together.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A Web crawler is a program that downloads Web pages, commonly for a Web search engine or a Web cache. Roughly, a crawler starts off with an initial set of URLs S_0. It first places S_0 in a queue, where all URLs to be retrieved are kept and prioritized. From this queue, the crawler gets a URL (in some order), downloads the page, extracts any URLs in the downloaded page, and puts the new URLs in the queue. This process is repeated until the crawler decides to stop, for any one of various reasons. Every page that is retrieved is given to a client that saves the pages, creates an index for the pages, or analyzes the content of the pages.

Crawlers are widely used today. Crawlers for the major search engines (e.g., Google, AltaVista, and Excite) attempt to visit a significant portion of textual Web pages, in order to build content indexes. Other crawlers may also visit many pages, but may look only for certain types of information (e.g., email addresses). At the other end of the spectrum, we have personal crawlers that scan for pages of interest to a particular user, in order to build a fast access cache (e.g., NetAttache).

Because the Web is gigantic and being constantly updated, the design of a good crawler poses many challenges. For example, according to various studies [BYBCW00, LG99, LG98, BB99], there exist more than a billion pages available on the Web. Given that the average size of a Web page is around 5–10K bytes, the textual data itself amounts to at least tens of terabytes. The growth rate of the Web is even more dramatic. According to [LG98, LG99], the size of the Web has doubled in less than two years, and this growth rate is projected to continue for the next two years. Aside from these newly created pages, existing pages are continuously being updated [PP97, WM99, DFK99, CGM01]. For example, according to our study in Chapter 4, roughly 40% of the Web pages that we monitored were updated almost once every week.

1.1 Challenges in implementing a crawler

Given this size and change rate of the Web, the crawler needs to address many important challenges, including the following:

1. **What pages should the crawler download?** In most cases, the crawler cannot download *all* pages on the Web. Even the most comprehensive search engine currently indexes a small fraction of the entire Web [LG99, BB99]. Given this fact, it is important for the crawler to carefully select the pages and to visit "important" pages first, so that the fraction of the Web that is visited (and kept up-to-date) is more meaningful.

2. **How should the crawler refresh pages?** Once the crawler has downloaded a significant number of pages, it has to start *revisiting* the downloaded pages in order to detect changes and refresh the downloaded collection. Because Web pages are changing at very different rates [CGM00a, WM99], the crawler needs to carefully decide which pages to revisit and which pages to skip in order to achieve high "freshness" of pages. For example, if a certain page rarely changes, the crawler may want to revisit the page less often, in order to visit more frequently changing ones.

3. **How should the load on the visited Web sites be minimized?** When the crawler collects pages from the Web, it consumes resources belonging to other organizations [Kos95]. For example, when the crawler downloads page p on site S, the site needs to retrieve page p from its file system, consuming disk and CPU resources. After this retrieval the page then needs to be transferred through the network, which is another resource shared by multiple organizations. Therefore, the crawler should minimize its impact on these resources [Rob]. Otherwise, the administrators of a Web site or a particular network may complain and sometimes may completely block access by the crawler.

4. **How should the crawling process be parallelized?** Due to the enormous size of the Web, crawlers often run on multiple machines and download pages in parallel [BP98, CGM00a]. This parallelization is often necessary in order to download a

large number of pages in a reasonable amount of time. Clearly these parallel crawlers should be coordinated properly, so that different crawlers do not visit the same Web site multiple times. However, this coordination can incur significant communication overhead, limiting the number of simultaneous crawlers.

1.2 Organization of dissertation

In this dissertation, we address the above challenges by designing, implementing, and evaluating a Web crawler. As part of the dissertation, we built the Stanford WebBase crawler, which can download Web pages at a high rate without imposing too much load on individual Web sites.[1] In this dissertation we present the challenges that we encountered during the implementation of the WebBase crawler and then describe our experimental and algorithmic solutions that address the challenges. To that end, the rest of this dissertation is organized as follows.

Chapter 2: Page Selection We start by discussing how a crawler can select "important" pages early, so that the quality of the downloaded pages are maximized. Clearly, the importance of a page is a subjective matter that may vary among applications. Thus we first identify possible definitions of "page importance," propose crawler evaluation models, and design page selection algorithms. We then experimentally evaluate the proposed algorithms.

Chapter 3: Parallel Crawlers We discuss how we can parallelize a crawling process to increase the page download rate. Compared to a single-process crawler, a parallel crawler needs to address some additional issues (e.g., communication between crawling processes). Thus we first propose various evaluation metrics for a parallel crawler. Then we explore the design space for a parallel crawler and how these design choices affect its effectiveness under the proposed metrics. We also experimentally study what design choices should be taken in various scenarios.

Chapter 4: Web Evolution Experiment In order to understand how a crawler can effectively refresh pages, it is imperative to know how Web pages change over time. In this

[1]Currently, WebBase maintains about 130 million pages downloaded from the Web and it runs at the rate of 100 pages per second. While the WebBase crawler can increase the download rate by running more processes, we limit the rate to 100 pages per second because of our network bandwidth constraint.

chapter, we first explain the design of our Web evolution experiment, in which we tried to understand how Web pages change over time. We then describe the results of the experiment and present observed change characteristics of Web pages. From the experimental data, we will also develop a mathematical model that describes the changes of Web pages well. The techniques described in this chapter shed light on how a crawler itself can learn how Web pages change over time.

Chapter 5: Page Refresh Policy Based on the results obtained from Chapter 4, we then discuss how a crawler should refresh pages effectively. In determining refresh policies, many interesting questions arise, including the following: Should a crawler refresh a page more often if the page is believed to change more often? How often should a crawler refresh its pages in order to maintain 80% of the pages "up to date"?

The discussion of this chapter answers these questions and helps us understand page refresh policies better. Surprisingly, some of the answers to the above questions are rather unexpected, but we explain why we get such unexpected answers.

Chapter 6: Change Frequency Estimation The crawler itself needs to estimate how often Web pages change in order to implement the policies described in Chapter 5. In this chapter, we finish our discussion of "page refresh policy" by describing how a crawler can estimate change frequencies of Web pages. Intuitively, a crawler can estimate the change frequency based on how many changes were detected in previous visits. But the crawler has to estimate the frequency correctly even when it may have missed some changes.

Chapter 7: Crawler Architecture In Chapter 7, we conclude by describing the general architecture of a crawler, which can incorporate the policies presented in this dissertation. We also discuss some remaining issues for the design of a crawler and explore their implications.

1.2.1 Related work

Web crawlers have been studied since the advent of the Web [McB94, Pin94, Bur98, PB98, HN99, CGM00a, Mil98, Eic94, CGMP98, CvdBD99b, DCL$^+$00, CLW98, CGM00b]. These and other relevant studies can be classified into the following categories.

Page update Web crawlers need to update the downloaded pages periodically, in order to maintain the pages up to date. Reference [CLW98] studies how to schedule a Web crawler to improve freshness. The model used for Web pages is similar to the one used in this dissertation; however, the model for the crawler and freshness are very different.

Many researchers have studied how to build a scalable and effective Web cache, to minimize access delay, server load, and bandwidth usage [YBS99, GS96, BBM+97]. While some of the work touches on the consistency issue of cached pages, they try to develop *new* protocols that can help reduce the inconsistency of cached pages. In contrast, this dissertation proposes a mechanism that can improve *freshness* of cached pages using the existing HTTP protocol without any modification.

In the data warehousing context, a lot of work has been done on efficiently maintaining local copies, or *materialized views* [HGMW+95, HRU96, ZGMHW95]. However, most of that work focused on different issues, such as minimizing the size of the view while reducing the query response time [HRU96].

Web evolution References [WM99, WVS+99, 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 the study in this dissertation, in terms of the scale and the length of the experiments. Also, their focus is different from this dissertation. Reference [WM99] investigates page changes to improve *Web caching policies*, and reference [PP97] studies how page changes are related to *access patterns*.

Change frequency estimation for Web pages The problem of estimating change frequency has long been studied in the statistics community [TK98, Win72, MS75, Can72]. However, most of the existing work assumes that the complete change history is known, which is not true in many practical scenarios. In Chapter 6, we study how to estimate the change frequency based on an *incomplete* change history. By using the estimator in this chapter, we can get a more accurate picture on how often Web pages change.

Page selection Since many crawlers can download only a small subset of the Web, crawlers need to carefully decide which pages to download. Following up on our work in [CGMP98], references [CvdBD99b, DCL+00, Muk00, MNRS99] explore how a crawler can discover and identify “important” pages early, and propose some algorithms to achieve this goal.

General crawler architecture References [PB98, HN99, Mil98, Eic94] describe the general architecture of various Web crawlers. For example, reference [HN99] describes the architecture of an AltaVista crawler and its major design goals. Reference [PB98] describes the architecture of the initial version of the Google crawler. In contrast to this work, this dissertation first explores the possible design space for a Web crawler and then compares these design choices carefully using experimental and analytical methods. During our discussion, we also try to categorize existing crawlers based on the issues we will describe. (Unfortunately, very little is known about the internal workings of commercial crawlers as they are closely guarded secrets. Our discussion will be limited to the crawlers in the open literature.)

Chapter 2

Page Selection

2.1 Introduction

The design of a good crawler presents many challenges. In particular, the crawler must deal with huge volumes of data. Unless it has unlimited computing resources and unlimited time, it must carefully decide what URLs to download and in what order. In this chapter we address this important challenge: How should a crawler select URLs to download from its list of known URLs? If a crawler intends to perform a single scan of the entire Web, and the load placed on target sites is not an issue, then any URL order will suffice. That is, eventually every single known URL will be visited, so the order is not critical. However, most crawlers will not be able to visit every possible page for two main reasons:

- The crawler or its client may have limited storage capacity, and may be unable to index or analyze all pages. Currently the Web is believed to have several terabytes of textual data and is growing rapidly, so it is reasonable to expect that most clients will not want or will not be able to cope with all that data [Kah97].

- Crawling takes time, so at some point the crawler may need to start revisiting previously retrieved pages, to check for changes. This means that it may never get to some pages. It is currently estimated that there exist more than one billion pages available on the Web [BYBCW00, LG99, LG98] and many of these pages change at rapid rates [PP97, WM99, DFK99, CGM01].

In either case, it is important for the crawler to visit "important" pages first, so that the fraction of the Web that is visited (and kept up to date) is more meaningful. In the following sections, we present several different useful definitions of importance, and develop crawling priorities so that important pages have a higher probability of being visited first. We also present experimental results from crawling the Stanford University Web pages that show how effective the different crawling strategies are.

2.2 Importance metrics

Not all pages are necessarily of equal interest to a crawler's client. For instance, if the client is building a specialized database on a particular topic, then pages that refer to that topic are more important, and should be visited as early as possible. Similarly, a search engine may use the number of Web URLs that point to a page, the so-called *backlink count*, to rank user query results. If the crawler cannot visit all pages, then it is better to visit those with a high backlink count, since this will give the end-user higher ranking results.

Given a Web page p, we can define the importance of the page, $I(p)$, in one of the following ways. (These metrics can be combined, as will be discussed later.)

1. *Similarity to a Driving Query Q:* A query Q drives the crawling process, and $I(p)$ is defined to be the textual similarity between p and Q. Similarity has been well studied in the Information Retrieval (IR) community [Sal83] and has been applied to the Web environment [YLYL95]. We use $IS(p)$ to refer to the importance metric in this case. We also use $IS(p, Q)$ when we wish to make the query explicit.

   To compute similarities, we can view each document (p or Q) as an n-dimensional vector $\langle w_1, \ldots, w_n \rangle$. The term w_i in this vector represents the ith word in the vocabulary. If w_i does not appear in the document, then w_i is zero. If it does appear, w_i is set to represent the significance of the word. One common way to compute the significance w_i is to multiply the number of times the ith word appears in the document by the inverse document frequency (*idf*) of the ith word. The *idf* factor is one divided by the number of times the word appears in the entire "collection," which in this case would be the entire Web. The *idf* factor corresponds to the content discriminating power of a word: A term that appears rarely in documents (e.g., "queue") has a high *idf*, while a term that occurs in many documents (e.g., "the") has a low *idf*. (The w_i terms can also take into account where in a page the word appears. For instance, words appearing in the title of an HTML page may be given a higher weight than other words in the body.) The similarity between p and Q can then be defined as the inner product of the p and Q vectors. Another option is to use the cosine similarity measure, which is the inner product of the normalized vectors.

Note that if we do not use *idf* terms in our similarity computation, the importance of a page, $IS(p)$, can be computed with "local" information, i.e., just p and Q. However, if we use *idf* terms, then we need global information. During the crawling process we have not seen the entire collection, so we have to estimate the *idf* factors from the pages that have been crawled, or from some reference *idf* terms computed at some other time. We use $IS'(p)$ to refer to the estimated importance of page p, which is different from the actual importance $IS(p)$ that is computable only after the entire Web has been crawled. If *idf* factors are not used, then $IS'(p) = IS(p)$.

2. *Backlink Count:* The value of $I(p)$ is the number of links to p that appear over the entire Web. We use $IB(p)$ to refer to this importance metric. Intuitively, a page p that is linked to by many pages is more important than one that is seldom referenced. This type of "citation count" has been used extensively to evaluate the impact of published papers. On the Web, $IB(p)$ is useful for ranking query results, giving end-users pages that are more likely to be of general interest.

   Note that evaluating $IB(p)$ requires counting backlinks over the entire Web. A crawler may estimate this value with $IB'(p)$, the number of links to p that have been seen so far.

3. *PageRank:* The $IB(p)$ metric treats all links equally. Thus, a link from the Yahoo home page counts the same as a link from some individual's home page. However, since the Yahoo home page is more important (it has a much higher IB count), it would make sense to value that link more highly. The PageRank backlink metric, $IR(p)$, recursively defines the importance of a page to be the weighted sum of the importance of the pages that have backlinks to p. Such a metric has been found to be very useful in ranking results of user queries [PB98, Goo]. We use $IR'(p)$ for the estimated value of $IR(p)$ when we have only a subset of pages available.

   More formally, if a page has no outgoing link, we assume that it has outgoing links to every single Web page. Next, consider a page p that is pointed at by pages $t_1, \ldots, t_n$. Let c_i be the number of links going out of page t_i. Also, let d be a damping factor (whose intuition is given below). Then, the weighted backlink count of page p is given by

$$IR(p) = (1 - d) + d\,[IR(t_1)/c_1 + \cdots + IR(t_n)/c_n]$$

This leads to one equation per Web page, with an equal number of unknowns. The equations can be solved for the IR values. They can be solved iteratively, starting with all IR values equal to 1. At each step, the new $IR(p)$ value is computed from the old $IR(t_i)$ values (using the equation above), until the values converge. This calculation corresponds to computing the principal eigenvector of the link matrices.

One intuitive model for PageRank is that we can think of a user "surfing" the Web, starting from any page, and randomly selecting from that page a link to follow. When the user reaches a page with no outlinks, he jumps to a random page. Also, when the user is on a page, there is some probability, d, that the next visited page will be completely random. This damping factor d makes sense because users will only continue clicking on one task for a finite amount of time before they go on to something unrelated. The $IR(p)$ values we computed above give us the probability that our random surfer is at p at any given time.

4. *Forward Link Count:* For completeness we may want to consider a metric $IF(p)$ that counts the number of links that emanate from p. Under this metric, a page with many outgoing links is very valuable, since it may be a Web directory. This metric can be computed directly from p, so $IF'(p) = IF(p)$. This kind of metric has been used in conjunction with other factors to reasonably identify index pages [PPR96]. We could also define a weighted forward link metric, analogous to $IR(p)$, but we do not consider it here.

5. *Location Metric:* The $IL(p)$ importance of page p is a function of its location, not of its contents. If URL u leads to p, then $IL(p)$ is a function of u. For example, URLs ending with ".com" may be deemed more useful than URLs with other endings, or URLs containing the string "home" may be more of interest than other URLs. Another location metric that is sometimes used considers URLs with fewer slashes more useful than those with more slashes. All these examples are local metrics since they can be evaluated simply by looking at the URL u.

As stated earlier, our importance metrics can be combined in various ways. For example, we may define a metric $IC(p) = k_1 \cdot IS(p, Q) + k_2 \cdot IB(p)$, for some constants k_1, k_2. This combines the similarity metric (under some given query Q) and the backlink metric. Pages that have relevant content *and* many backlinks would be the highest ranked. (Note that a similar approach was used to improve the effectiveness of a search engine [Mar97].)

2.3 Problem definition

Our goal is to design a crawler that if possible visits high $I(p)$ pages before lower ranked ones, for some definition of $I(p)$. Of course, the crawler will only have available $I'(p)$ values, so based on these it will have to guess what are the high $I(p)$ pages to fetch next.

Our general goal can be stated more precisely in three ways, depending on how we expect the crawler to operate. (In our evaluations of Section 2.5 we use the second model in most cases, but we do compare it against the first model in one experiment. Nevertheless, we believe it is useful to discuss all three models to understand the options.)

- **Crawl & Stop:** Under this model, the crawler C starts at its initial page p_0 and stops after visiting K pages. At this point an "ideal" crawler would have visited pages $r_1, \ldots, r_K$, where r_1 is the page with the highest importance value, r_2 is the next highest, and so on. We call pages r_1 through r_K the "hot" pages. The K pages visited by our real crawler will contain only M pages with rank higher than or equal to $I(r_K)$. We define the performance of the crawler C to be $P_{CS}(C) = M/K$. The performance of the ideal crawler is of course 1. A crawler that somehow manages to visit pages entirely at random, and may revisit pages, would have a performance of K/T, where T is the total number of pages in the Web. (Each page visited is a hot page with probability K/T. Thus, the expected number of desired pages when the crawler stops is K^2/T.)

- **Crawl & Stop with Threshold:** We again assume that the crawler visits K pages. However, we are now given an importance target G, and any page with $I(p) \geq G$ is considered hot. Let us assume that the total number of hot pages is H. The performance of the crawler, $P_{ST}(C)$, is the fraction of the H hot pages that have been visited when the crawler stops. If $K < H$, then an ideal crawler will have performance K/H. If $K \geq H$, then the ideal crawler has the perfect performance 1. A purely random crawler that revisits pages is expected to visit $(H/T) \cdot K$ hot pages when it stops. Thus, its performance is K/T. Only when the random crawler visits all T pages is its performance expected to be 1.

- **Limited Buffer Crawl:** In this model we consider the impact of limited storage on the crawling process. We assume that the crawler can only keep B pages in its buffer. Thus, after the buffer fills up, the crawler must decide what pages to flush to make room for new pages. An ideal crawler could simply drop the pages with lowest $I(p)$ value, but a real crawler must guess which of the pages in its buffer will eventually have low $I(p)$ values. We allow the crawler to visit a total of T pages, equal to the total number of Web pages. At the end of this process, the fraction of the B buffer pages that are hot gives us the performance $P_{BC}(C)$. We can define hot pages to be those with $I(p) \geq G$, where G is a target importance, or those with $I(p) \geq I(r_B)$, where r_B is the page with the Bth highest importance value. The performance of an ideal and a random crawler are analogous to those in the previous cases.

Note that to evaluate a crawler under any of these metrics, we need to compute the actual $I(p)$ values of pages, and this involves crawling the "entire" Web. To keep our experiments (Section 2.5) manageable, we imagine that the Stanford University pages form the entire Web, and we only evaluate performance in this context. That is, we assume that all pages outside of Stanford have $I(p) = 0$, and that links to pages outside of Stanford or links from pages outside of Stanford do not count in $I(p)$ computations. In Section 2.5.2 we study the implications of this assumption by also analyzing a smaller Web within the Stanford domain, and seeing how Web size impacts performance.

2.4 Ordering metrics

A crawler keeps a queue of URLs it has seen during a crawl, and must select from this queue the next URL to visit. The ordering metric O is used by the crawler for this selection, i.e., it selects the URL u such that $O(u)$ has the highest value among all URLs in the queue. The O metric can only use information seen (and remembered if space is limited) by the crawler.

The O metric should be designed with an importance metric in mind. For instance, if we are searching for high $IB(p)$ pages, it makes sense to use $O(u) = IB'(p)$, where p is the page u points to. However, it might also make sense to use $O(u) = IR'(p)$, even if our importance metric is not weighted. In our experiments, we explore the types of ordering metrics that are best suited for either $IB(p)$ or $IR(p)$.

For a location importance metric $IL(p)$, we can use that metric directly for ordering since the URL of p directly gives the $IL(p)$ value. However, for forward link $IF(p)$ and similarity $IS(p)$ metrics, it is much harder to devise an ordering metric since we have not seen p yet. As we will see, for similarity, we may be able to use the text that anchors the URL u as a predictor of the text that p might contain. Thus, one possible ordering metric $O(u)$ is $IS(A, Q)$, where A is the anchor text of the URL u, and Q is the driving query.

2.5 Experiments

To avoid network congestion and heavy loads on the servers, we did our experimental evaluation in two steps. In the first step, we physically crawled all Stanford Web pages and built a local repository of the pages. This was done with the Stanford WebBase [BP98], a system designed to create and maintain large Web repositories.

After we built the repository, we ran our *virtual* crawlers on it to evaluate different crawling schemes. Note that even though we had the complete image of the Stanford domain in the repository, our virtual crawler based its crawling decisions only on the pages it saw for itself. In this section we briefly discuss how the particular database was obtained for our experiments.

2.5.1 Description of dataset

To download an image of the Stanford Web pages, we started WebBase with an initial list of "`stanford.edu`" URLs. These 89,119 URLs were obtained from an earlier crawl. During the crawl, non-Stanford URLs were ignored. Also, we limited the actual data that we collected for two reasons. The first is that many heuristics are needed to avoid automatically generated, and potentially infinite, sets of pages. For example, any URLs containing "`/cgi-bin/`" are not crawled, because they are likely to contain programs which generate infinite sets of pages, or produce other undesirable side effects such as an unintended vote in an online election. We used similar heuristics to avoid downloading pages generated by programs. Another way the data set is reduced is through the *robots exclusion protocol* [Rob], which allows Webmasters to define pages they do not want crawled by automatic systems.

At the end of the process, we had downloaded 375,746 pages and had 784,592 known URLs to Stanford pages. The crawl was stopped before it was complete, but most of the uncrawled URLs were on only a few servers, so we believe the dataset we used to be a reasonable representation of the `stanford.edu` Web. In particular, it should be noted that 352,944 of the known URLs were on one server, `http://www.slac.stanford.edu`, which has a program that could generate an unlimited number of Web pages. Since the dynamically-generated pages on the server had links to other dynamically generated pages, we would have downloaded an infinite number of pages if we naively followed the links.

Our dataset consisted of about 225,000 crawled "valid" HTML pages,[1] which consumed roughly 2.5GB of disk space. However, out of these 225,000 pages, 46,000 pages were unreachable from the starting point of the crawl, so the total number of pages for our experiments was 179,000.

We should stress that the virtual crawlers that will be discussed next do not use WebBase directly. As stated earlier, they use the dataset collected by the WebBase crawler, and do their own crawling on it. The virtual crawlers are simpler than the WebBase crawler. For instance, they can detect if a URL is invalid simply by seeing if it is in the dataset. Similarly, they do not need to distribute the load to visited sites. These simplifications are fine, since the virtual crawlers are only used to evaluate ordering schemes, and not to do real crawling.

2.5.2 Backlink-based crawlers

In this section we study the effectiveness of various ordering metrics, for the scenario where importance is measured through backlinks (i.e., either the $IB(p)$ or $IR(p)$ metrics). We start by describing the structure of the virtual crawler, and then consider the different ordering metrics. Unless otherwise noted, we use the Stanford dataset described in Section 2.5.1, and all crawls are started from the Stanford homepage. For the PageRank metric we use a damping factor d of 0.9 (for both $IR(p)$ and $IR'(p)$) for all of our experiments.

Figure 2.1 shows our basic virtual crawler. The crawler manages three main data structures. Queue `url_queue` contains the URLs that have been seen and need to be visited. Once a page is visited, it is stored (with its URL) in `crawled_pages`. `links` holds pairs of the form (u_1, u_2), where URL u_2 was seen in the visited page with URL u_1. The crawler's ordering metric is implemented by the function `reorder_queue()`, shown in Figure 2.2. We used three ordering metrics: (1) breadth-first (2) backlink count $IB'(p)$, and (3) PageRank $IR'(p)$. The breadth-first metric places URLs in the queue in the order in which they are discovered, and this policy makes the crawler visit pages in breadth-first order.

[1]We considered a page *valid* when its Web server responded with the HTTP header "`200 OK`."

Algorithm 2.5.1 Crawling algorithm (backlink based)
Input: `starting_url`: seed URL
Procedure:

```
[1] enqueue(url_queue, starting_url)
[2] while (not empty(url_queue))
[3]    url = dequeue(url_queue)
[4]    page = crawl_page(url)
[5]    enqueue(crawled_pages, (url, page))
[6]    url_list = extract_urls(page)
[7]    foreach u in url_list
[8]       enqueue(links, (url, u))
[9]       if (u∉url_queue and (u,-)∉crawled_pages)
[10]         enqueue(url_queue, u)
[11]   reorder_queue(url_queue)
```

Function description:

```
enqueue(queue, element): append element at the end of queue
dequeue(queue)         : remove the element at the beginning
                         of queue and return it
reorder_queue(queue)   : reorder queue using information in
                         links (refer to Figure 2.2)
```

Figure 2.1: Basic crawling algorithm

(1) breadth first

```
    do nothing (null operation)
```

(2) backlink count, $IB'(p)$

```
    foreach u in url_queue
       backlink_count[u] = number of terms (-,u) in links
    sort url_queue by backlink_count[u]
```

(3) PageRank $IR'(p)$

```
    solve the following set of equations:
       $IR[u] = (1 - 0.9) + 0.9 \sum_i \frac{IR[v_i]}{c_i}$, where
       $(v_i, u) \in$ links and $c_i$ is the number of links in the page $v_i$
    sort url_queue by $IR(u)$
```

Figure 2.2: Description of `reorder_queue()` of each ordering metric

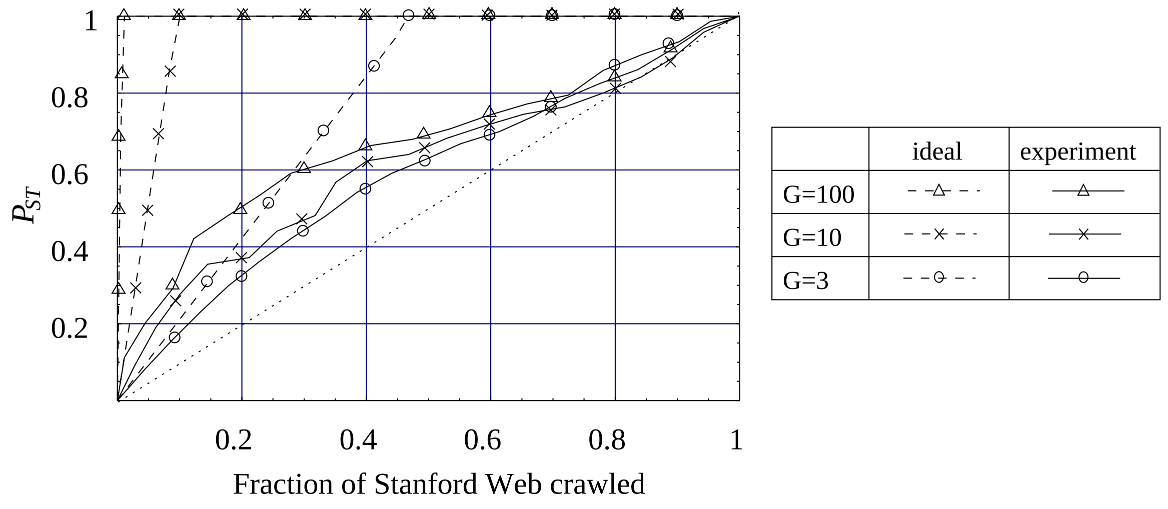


Figure 2.3: Fraction of Stanford Web crawled vs. P_{ST}. $I(p) = IB(p)$; $O(u) = IB'(p)$.

We start by showing in Figure 2.3 the crawler's performance with the backlink ordering metric. In this scenario, the importance metric is the number of backlinks to a page ($I(p) = IB(p)$) and we consider a *Crawl & Stop with Threshold* model in Section 2.3 with G either 3, 10, or 100. Recall that a page with G or more backlinks is considered important, i.e., hot. Under these hot page definitions, about H = 85,000 (47%), 17,500 (10%) and 1,400 (0.8%) pages out of 179,000 total Web pages were considered hot, respectively.

In Figure 2.3, the horizontal axis is the fraction of the Stanford Web pages that has been crawled over time. At the right end of the horizontal axis, all 179,000 pages have been visited. The vertical axis represents P_{ST}, the fraction of the total hot pages that has been crawled at a given point. The solid lines in the figure show the results from our experiments. For example, when the crawler in our experiment visited 0.2 (20%) of the Stanford pages, it crawled 0.5 (50%) of the total hot pages for G = 100. The dashed lines in the graph show the expected performance of ideal crawlers. An ideal crawler reaches performance 1 when H pages have been crawled. The dotted line represents the performance of a random crawler, and it increases linearly over time.

The graph shows that as our definition of a hot page becomes more stringent (larger G), the faster the crawler can locate the hot pages. This result is to be expected, since pages with many backlinks will be seen quickly after the crawl starts. Figure 2.3 also shows that even if G is large, finding the "last" group of hot pages is always difficult. That is, to the right of the 0.8 point on the horizontal axis, the crawler finds hot pages at roughly the same

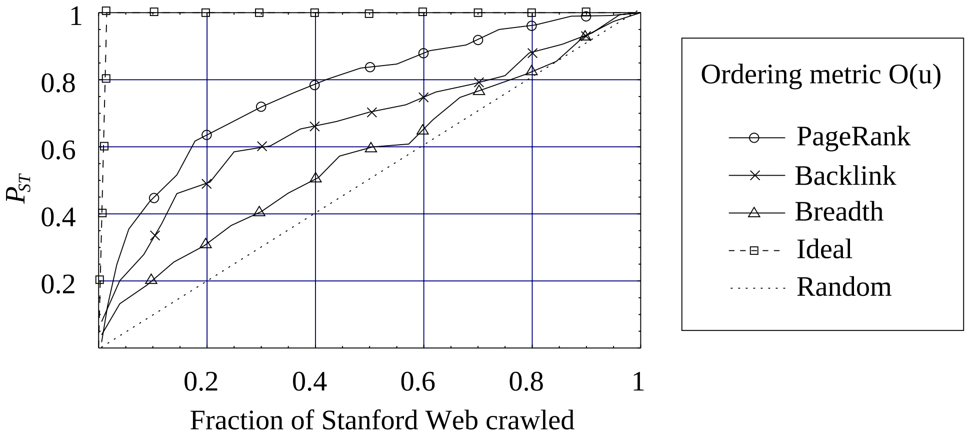


Figure 2.4: Fraction of Stanford Web crawled vs. P_{ST}. $I(p) = IB(p)$; $G = 100$.

rate as a random crawler.

In our next experiment we compare three different ordering metrics: 1) breadth-first 2) backlink-count and 3) PageRank (corresponding to the three functions of Figure 2.2). We continue to use the *Crawl & Stop with Threshold* model, with $G = 100$, and a $IB(p)$ importance metric. Figure 2.4 shows the results of this experiment. The results are rather counterintuitive. Intuitively one would expect that a crawler using the backlink ordering metric $IB'(p)$ that matches the importance metric $IB(p)$ would perform the best. However, this is not the case, and the PageRank metric $IR'(p)$ outperforms the $IB'(p)$ one. To understand why, we manually traced the crawler's operation. We noticed that often the $IB'(p)$ crawler behaved like a depth-first one, frequently visiting pages in one "cluster" before moving on to the next. On the other hand, the $IR'(p)$ crawler combined breadth and depth in a better way.

To illustrate, let us consider the Web fragment of Figure 2.5. With $IB'(p)$ ordering, the crawler visits a page like the one labeled p_1 and quickly finds a cluster A of pages that point to each other. The A pages temporarily have more backlinks than page p_2, so the visit of page p_2 is delayed even if page p_2 actually has more backlinks than the pages in cluster A. On the other hand, with $IR'(p)$ ordering, page p_2 may have higher rank (because its link comes from a high ranking page) than the pages in cluster A (that only have pointers from low ranking pages within the cluster). Therefore, page p_2 is reached faster.

In summary, during the early stages of a crawl, the backlink information is biased by the starting point. If the crawler bases its decisions on this skewed information, it tries

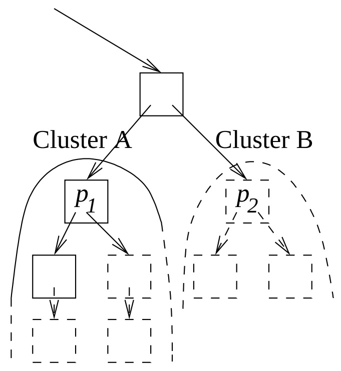


Figure 2.5: Crawling order

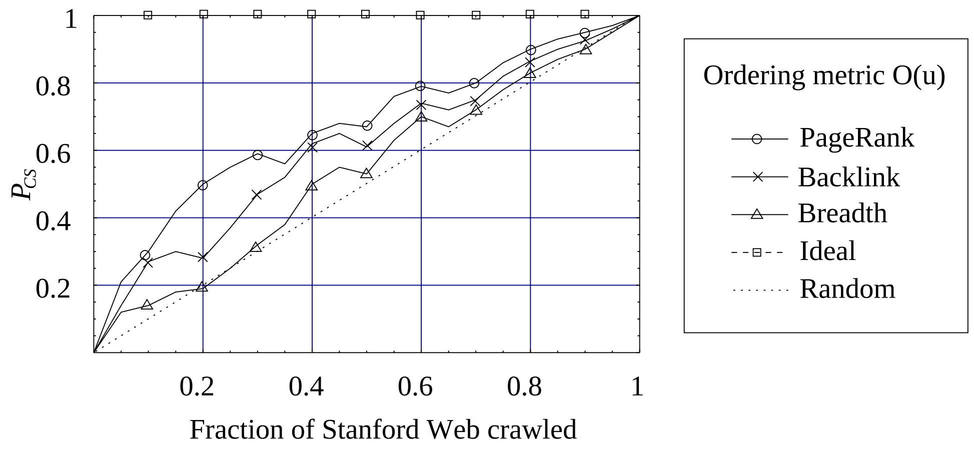


Figure 2.6: Fraction of Stanford Web crawled vs. P_{CS}. $I(p) = IB(p)$.

getting locally hot pages instead of globally hot pages, and this bias gets worse as the crawl proceeds. On the other hand, the $IR'(p)$ PageRank crawler is not as biased towards locally hot pages, so it gives better results regardless of the starting point.

Figure 2.6 shows that this conclusion is not limited to the *Crawl & Stop with Threshold* model. In the figure we show the performance of the crawlers under the *Crawl & Stop* model (Section 2.3). Remember that under the *Crawl & Stop* model, the definition of hot pages changes over time. That is, the crawler does not have a predefined notion of hot pages, and instead, when the crawler has visited, say, 30% of the entire Web, it considers the top 30% pages as hot pages. Therefore, an ideal crawler would have performance 1 at all times because it would download pages in the order of their importance. Figure 2.6 compares 1) breadth-first 2) backlink and 3) PageRank ordering metrics for the $IB(p)$ importance

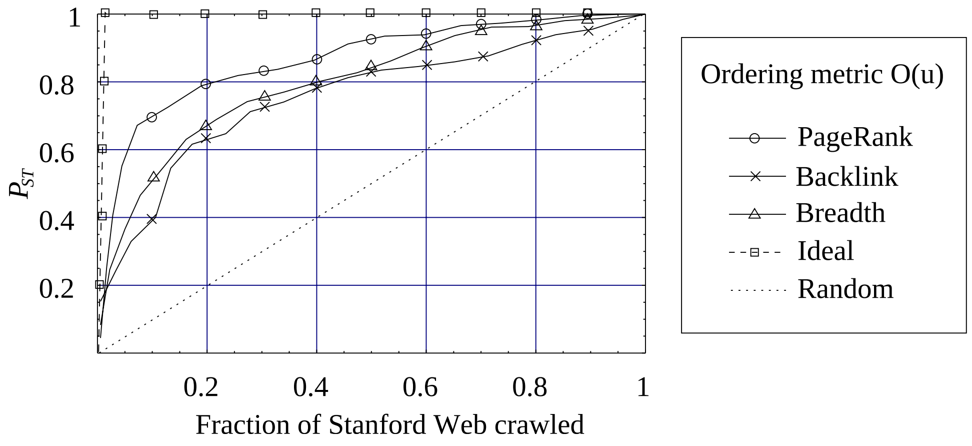


Figure 2.7: Fraction of Stanford Web crawled vs. P_{ST}. $I(p) = IR(p)$; $G = 13$.

metric under this model. The vertical axis represents P_{CS}, the crawled fraction of hot pages at each point under the varying definition of hot pages. The figure shows that the results of the *Crawl & Stop* model are analogous to those of the *Crawl & Stop with Threshold* model: The PageRank ordering metric shows the best performance.

Returning to the *Crawl & Stop with Threshold* model, Figure 2.7 shows the results when we use the $IR(p)$ PageRank importance metric with $G = 13$.[2] Again, the PageRank ordering metric shows the best performance. The backlink and the breadth-first metrics show similar performance. Based on these results, we recommend using the PageRank ordering metric for both the $IB(p)$ and the $IR(p)$ importance metrics.

2.5.3 Small-scale crawl

In some cases the crawler's client may only be interested in small portions of the Web. For instance, the client may be interested in a single site (to create a mirror, say). In this subsection we evaluate the ordering metrics in such a scenario.

To study the impact of scale on the performance of a crawler we ran experiments similar to those in Section 2.5.2 only on the pages of the Stanford Database Group (on the server `http://www-db.stanford.edu`). This subset of the Stanford pages consists of about 1,100 HTML pages, which is much smaller than the entire Stanford domain. In most of our experiments for the Database Group domain, crawling performance was not as good as for

[2]When $G = 13$ for the $IR(p)$ metric, the number of hot pages was about $1,700$ (1%), which is close to the $1,400$ of Figure 2.4.

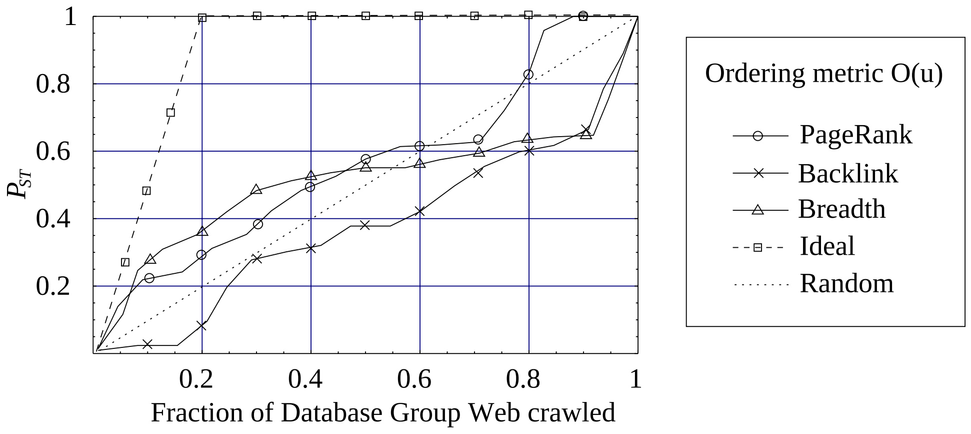


Figure 2.8: Fraction of DB group Web crawled vs. P_{ST}. $I(p) = IB(p)$; $G = 5$.

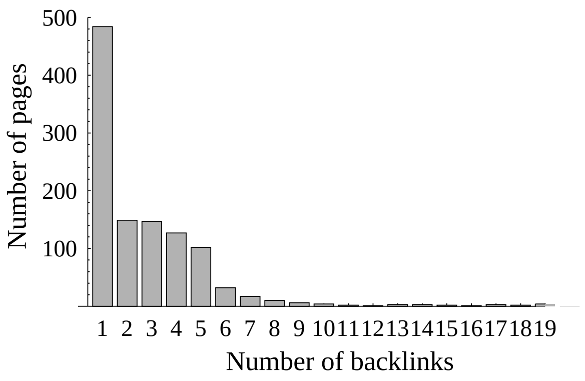


Figure 2.9: Histogram of backlink counts within DB group Web

the Stanford domain. Figure 2.8 shows one of the results. In this case, we use the *Crawl & Stop with Threshold* model with $G = 5$ with the importance metric $IB(p)$. The graph shows that performance can be even worse than that of a random crawler at times, for all ordering metrics.

This poor performance is mainly because an importance metric based on backlinks is not a good measure of importance for a small domain. In a small domain, most pages have only a small number of backlinks and the number of backlinks therefore is very sensitive to a page creator's style. For example, Figure 2.9 shows the histogram for the number of backlinks in the Database Group domain. The vertical axis shows the number of pages with a given backlink count. We can see that most pages have fewer than 5 backlinks. In this range, the rank of each page varies greatly according to the style used by the creator of the

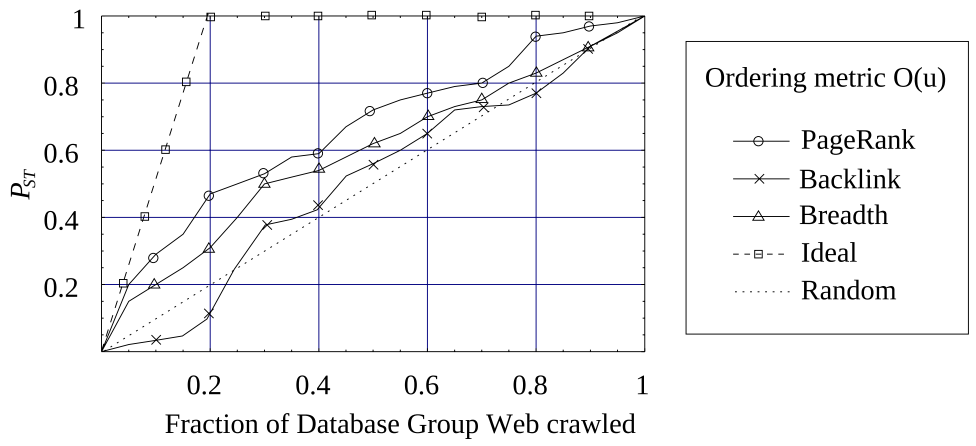


Figure 2.10: Percentage of DB group Web crawled vs. P_{ST}. $I(p) = IR(p)$; $G = 3$.

page. If the creator generates many cross-links between his pages, then his pages have a high $IB(p)$ rank, otherwise they do not. Therefore, the rank is very sensitive and is not a good measure of the importance of the pages.

In Figure 2.8 we can see the impact of "locally dense" clusters that have many "locally popular" but "globally unpopular" pages. The performance of the backlink $IB'(p)$ crawler is initially quite flat: It initially does a depth-first crawl for the first cluster it found. After visiting about 20% of the pages, the crawler suddenly discovers a large cluster, and this accounts for the jump in the graph. On the other hand, the PageRank $IR'(p)$ crawler found this large cluster earlier, so its performance is much better initially.

In Figure 2.10 we show the results on the Database Group Web when the importance metric is $IR(p)$ PageRank metric with $G = 3$.[3] All three ordering metrics show better performance under the $IR(p)$ metric than under the $IB(p)$ metric, but still performance is not as good as that of the larger Stanford domain. Again, the $IR'(p)$ ordering metric shows the best performance.

2.5.4 Similarity-based crawlers

In the experiments of two previous subsections, we compared three different backlink-based crawlers. In this subsection, we present the results of our experiments on similarity-based crawlers. The similarity-based importance metric, $IS(p)$, measures the relevance of each page to a topic or a query that the user has in mind. There are clearly many possible

[3]When $G = 3$, the number of hot pages is similar to that of Figure 2.8.

Algorithm 2.5.2 Crawling algorithm (modified similarity based)
Input: `starting_url`: seed URL
Procedure:

```
[1] enqueue(url_queue, starting_url)
[2] while (not empty(hot_queue) and not empty(url_queue))
[3]   url = dequeue2(hot_queue, url_queue)
[4]   page = crawl_page(url)
[5]   enqueue(crawled_pages, (url, page))
[6]   url_list = extract_urls(page)
[7]   foreach u in url_list
[8]     enqueue(links, (url, u))
[9]     if (u∉url_queue and u∉hot_queue and (u,-)∉crawled_pages)
[10]      if (u contains computer in anchor or url)
[11]        enqueue(hot_queue, u)
[12]      else
[13]        enqueue(url_queue, u)
[14]  reorder_queue(url_queue)
[15]  reorder_queue(hot_queue)
```

Function description:

```
dequeue2(queue1, queue2): if (not empty(queue1)) dequeue(queue1)
                          else dequeue(queue2)
```

Figure 2.11: Similarity-based crawling algorithm

$IS(p)$ metrics to consider, so our experiments here are not intended to be comprehensive. Instead, our goal is to briefly explore the *potential* of various ordering schemes in some sample scenarios. In particular, for our first two experiments we consider the following $IS(p)$ definition: A page is considered hot if it contains the word *computer* in its title or if it has more than 10 occurrences of *computer* in its body.[4]

For similarity-based crawling, the crawler of Figure 2.1 is not appropriate, since it does not take the content of the page into account. To give priority to the pages related to the topic of interest, we modified our crawler as shown in Figure 2.11. This crawler keeps two queues of URLs to visit: `hot_queue` stores the URLs with the topic word *computer* in their anchors or in the URLs themselves. The second queue, `url_queue`, keeps the rest of the URLs. The crawler always prefers to take URLs to visit from `hot_queue`.

[4]In our third experiment we consider a different topic, *admission*, and show the results.

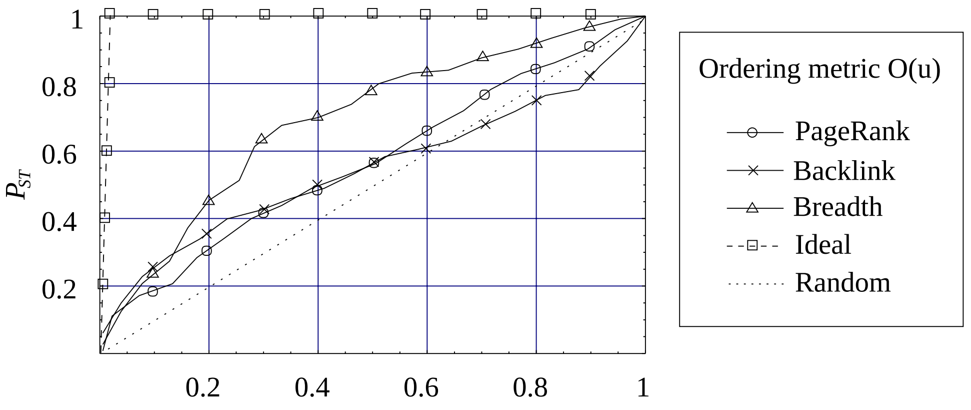


Figure 2.12: Basic similarity-based crawler. $I(p) = IS(p)$; topic is *computer*.

Figure 2.12 shows the P_{ST} results for this crawler for the $IS(p)$ importance metric defined above. The horizontal axis represents the fraction of the Stanford Web pages that has been crawled and the vertical axis shows the crawled fraction of the total hot pages. The results show that the backlink-count and the PageRank crawler behaved no better than a random crawler. Only the breadth-first crawler gave a reasonable result. This result is rather unexpected: All three crawlers differ only in their ordering metrics, which are neutral to the page content. All crawlers visited computer-related URLs immediately after their discovery. Therefore, all the schemes are theoretically equivalent and should give comparable results.

The observed unexpected performance difference arises from the breadth-first crawler's FIFO nature. The breadth-first crawler fetches pages in the order they are found. If a computer-related page is crawled earlier, then the crawler discovers and visits its child pages earlier as well. These pages have a tendency to be computer related, so performance is better.

Thus the observed property is that if a page has a high $IS(p)$ value, then its children are likely to have a high $IS(p)$ value too. To take advantage of this property, we modified our crawler as shown in Figure 2.13. This crawler places in the `hot_queue` URLs that have the target keyword in their anchor or URL, or that are within 3 links from a hot page.

Figure 2.14 illustrates the result of this crawling strategy. All crawlers showed significant improvement and the difference between the breadth-first crawler and the others decreased. While the breadth-first crawler is still superior to the other two, we believe that

Algorithm 2.5.3 Crawling algorithm (similarity based)
Input: `starting_url`: seed URL
Procedure:

```
[1] enqueue(url_queue, starting_url)
[2] while (not empty(hot_queue) and not empty(url_queue))
[3]   url = dequeue2(hot_queue, url_queue)
[4]   page = crawl_page(url)
[5]   if (page contains 10 or more computer in body
          or one computer in title)
[6]     hot[url] = True
[7]   enqueue(crawled_pages, (url, page))
[8]   url_list = extract_urls(page)
[9]   foreach u in url_list
[10]    enqueue(links, (url, u))
[11]    if (u ∉ url_queue and u ∉ hot_queue and (u,-) ∉ crawled_pages)
[12]      if (u contains computer in anchor or url)
[13]        enqueue(hot_queue, u)
[14]      else if (distance_from_hotpage(u) < 3)
[15]        enqueue(hot_queue, u)
[16]      else
[17]        enqueue(url_queue, u)
[18]  reorder_queue(url_queue)
[19]  reorder_queue(hot_queue)
```

Function description:

```
distance_from_hotpage(u):
  return 0 if (hot[u] = True)
  return 1 if (hot[v] = True and (v, u) ∈ links for some v)
  ...
```

Figure 2.13: Modified similarity-based crawling algorithm

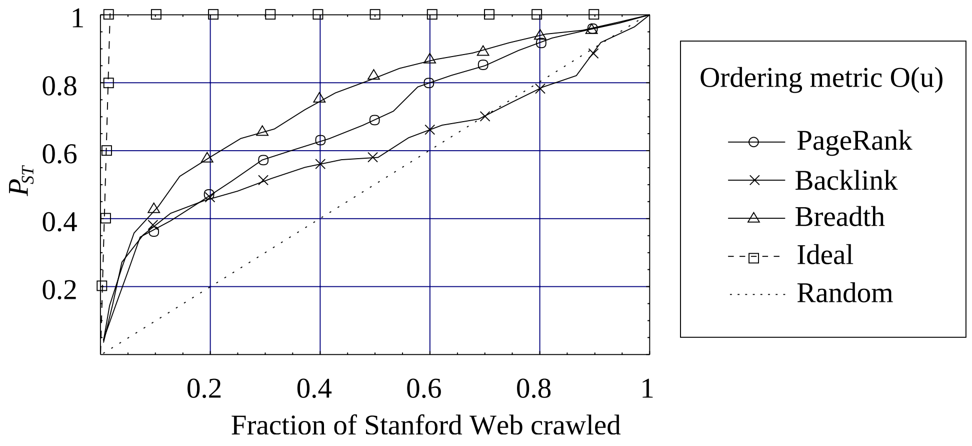


Figure 2.14: Modified similarity-based crawler. $I(p) = IS(p)$; topic is *computer*.

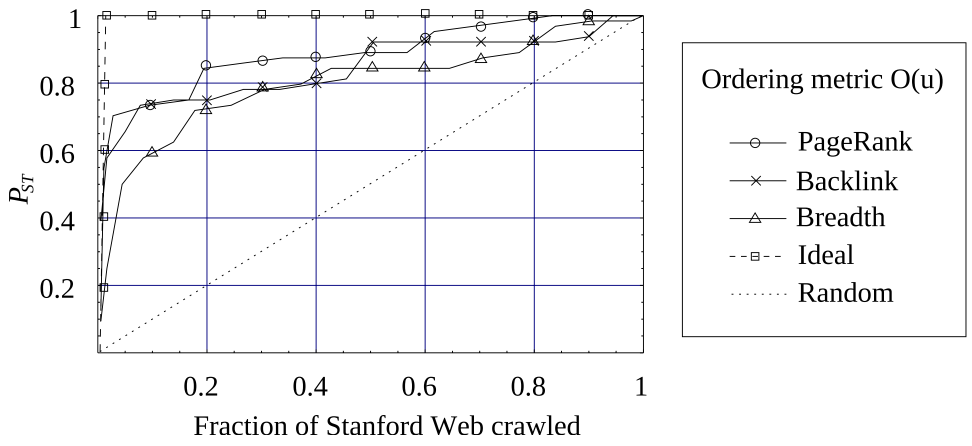


Figure 2.15: Modified similarity-based crawler. Topic is *admission*.

this difference is mainly due to statistical variation. In our other experiments, including the next one, the PageRank crawler shows similar or sometimes better performance than the breadth-first crawler.

In our final experiment, with results shown in Figure 2.15, we repeat the scenario reported in Figure 2.14 with a different query topic. In this case, the word *admission* is considered of interest. Details are not identical to the previous case, but the overall conclusion is the same: When similarity is important, it is effective to use an ordering metric that considers 1) the content of anchors and URLs and 2) the distance to the hot pages that have been discovered.

2.6 Related work

Recently, a *focused crawler* has been an active area of research. A focused crawler tries to collect pages related to a *particular topic* [CvdBD99b, CvdBD99a, DCL$^+$00, Muk00, AAGY01, MPR01, CGMP98, MNRS99, HJ$^+$98, NW01], by analyzing keywords and the link structure of the Web.

For instance, references [CvdBD99b, CvdBD99a, DCL$^+$00, AAGY01] propose various types of focused crawlers. The crawler described in [CvdBD99b, CvdBD99a] has two modules, a classifier and a distiller, that judge the relevance of pages and determine their download order. The decision is based on the relevance of the page and its potential proximity to other relevant pages. Reference [DCL$^+$00] uses Web link structure, so called *context graphs*, as a means to model the paths leading to relevant web pages. Similarly to our results in Section 2.5.4, reference [NW01] reports that a breadth-first crawler shows a good performance when the crawler starts from an appropriate Web page.

Researchers have also tried to use *machine learning* techniques to build a focused crawler. For instance, reference [MNRS99] adopts *reinforcement learning* and applies Bayes classifiers to the full text and anchor text of a page when classifying it.

Web believe our work described in this chapter [CGMP98] is one of the earliest work that studied the issue of downloading "important" or "relevant" pages early.

2.7 Conclusion

In this chapter we addressed the general problem of ordering URLs for crawling. We defined several different kinds of importance metrics, and built three models to evaluate crawlers. We experimentally evaluated several combinations of importance and ordering metrics, using the Stanford Web pages.

In general, our results show that PageRank, $IR'(p)$, is an excellent ordering metric when either pages with many backlinks or with high PageRank are sought. In addition, if the similarity to a driving query is important, it is useful to visit earlier URLs that:

- Have anchor text that is similar to the driving query;
- Have some of the query terms within the URL itself; or
- Have a short link distance to a page that is known to be hot.

With a good ordering strategy, we can build crawlers that can obtain a significant portion of the hot pages relatively early. This property can be extremely useful when we are trying to crawl a fraction of the Web, when our resources are limited, or when we need to revisit pages often to detect changes.

One limitation of our experiments is that they were run only over the Stanford Web pages. We believe that the Stanford pages are a reasonable sample: For example, they are managed by many different people who structure their pages in a variety of ways. They include many individual home pages, and also many clusters that are carefully managed by organizations. Nevertheless, it will be interesting to investigate in the future non-Stanford Web pages to analyze structural differences and their implication for crawling.

Chapter 3

Parallel Crawlers

3.1 Introduction

In this chapter we study how we should parallelize the crawling process so that we can maximize the download rate while minimizing the overhead from parallelization. As the size of the Web grows, it becomes more difficult – or impossible – to crawl the entire Web by a single process. Many search engines run multiple processes in parallel. We refer to this type of crawler as a *parallel crawler*.

While many existing search engines already use a parallel crawler internally, there has been little scientific research conducted on the parallelization of a crawler. Thus, little has been known about the tradeoffs among various design choices for a parallel crawler. In particular, we believe the following issues make the study of a parallel crawler difficult, yet interesting:

- *Overlap:* When multiple processes run in parallel to download pages, different processes may download the same page multiple times. One process may not be aware of the fact that another process has already downloaded a page. Clearly, multiple downloads should be minimized to save network bandwidth and increase the crawler's effectiveness. How can we coordinate the processes to prevent overlap?

- *Quality:* As discussed in Chapter 2, a crawler wants to download "important" pages first. However, in a parallel crawler, each process may not be aware of the whole image of the Web that they have collectively downloaded so far. For this reason, each process may make a crawling decision solely based on *its own* image of the Web (that it has downloaded) and thus make a poor decision. How can we make sure that the *quality* of downloaded pages is as good for a parallel crawler as for a single-process crawler?

- *Communication bandwidth:* In order to prevent overlap, or to improve the quality of the downloaded pages, crawling processes need to periodically communicate and coordinate with each other. However, this communication may grow significantly as the number of crawling processes increases. Exactly what do the processes need to communicate and how significant would the communication overhead be? Can we minimize this communication while maintaining the effectiveness of the crawler?

While parallel crawlers are challenging to operate, we believe that they have many important advantages, compared to single-process crawlers:

- *Scalability:* Due to the enormous size of the Web, it may be imperative to run a parallel crawler to achieve the required download rate in certain cases.

- *Network-load dispersion:* Multiple crawling processes of a parallel crawler may run at geographically distant locations, each downloading "geographically-adjacent" pages. For example, a process in Germany may download all European pages, while another one in Japan crawls all Asian pages. In this way, we can *disperse* the network load to multiple regions. In particular, this dispersion might be necessary when a single network cannot handle the heavy load from a large-scale crawl.

- *Network-load reduction:* In addition to dispersing load, a parallel crawler may actually *reduce* the network load. For example, assume that a crawler in North America retrieves a page from Europe. To be downloaded by the crawler, the page first has to go through the network in Europe, then the Europe-to-North America inter-continental network and finally the network in North America. Instead, if a crawling process in Europe collects all European pages, and if another process in North America crawls all North American pages, the overall network load will be reduced, because pages only go through "local" networks.

  Note that the downloaded pages may need to be transferred later to a central location, so that a central index can be built. However, even in that case, we believe that the transfer can be made significantly smaller than the original page download traffic, by using some of the following methods:

- *Compression:* Once the pages are collected and stored, it is easy to *compress* the data before sending them to a central location.
- *Difference:* Instead of sending the entire image of all downloaded pages, we may first take the *difference* between the previous image and the current one and send only this difference. Since many pages are static and do not change very often, this scheme may significantly reduce network traffic.
- *Summarization:* In certain cases, we may need only a central index, not the original pages themselves. In this case, we may extract the necessary information for an index construction (e.g., postings lists) and transfer this information only.

In summary, we believe a parallel crawler has many advantages and poses interesting challenges. This chapter makes the following contributions:

- We identify major issues and problems related to a parallel crawler and discuss how we can solve these problems.
- We present multiple architectures and techniques for a parallel crawler and discuss their advantages and disadvantages. As far as we know, most of these techniques have not been described in open literature. (Very little is known about the internal workings of commercial crawlers, as they are closely guarded secrets.)
- Using a large dataset (40M web pages) collected from the Web, we experimentally compare the design choices and study their tradeoffs quantitatively.
- We propose various optimization techniques that can minimize the coordination effort among crawling processes so that they can operate more independently while maximizing their effectiveness.

3.2 Architecture of a parallel crawler

In Figure 3.1 we illustrate the general architecture of a parallel crawler. A parallel crawler consists of multiple crawling processes, which we refer to as C-proc's. Each C-proc performs the basic tasks that a single-process crawler conducts. It downloads pages from the Web, stores the pages locally, extracts URLs from them and follows links. Depending on how the C-proc's split the download task, some of the extracted links may be sent to other C-proc's.

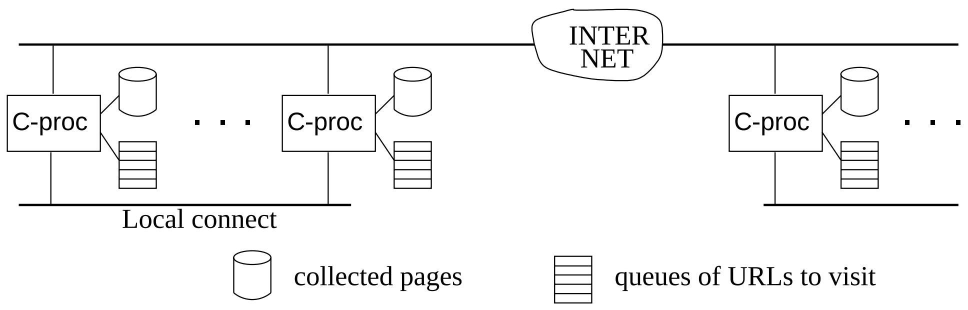


Figure 3.1: General architecture of a parallel crawler

The C-proc's performing these tasks may be distributed either on the same local network or at geographically distant locations.

- *Intra-site parallel crawler:* When all C-proc's run on the same local network and communicate through a high speed interconnect (such as LAN), we call it an *intra-site parallel crawler.* In Figure 3.1, this scenario corresponds to the case where all C-proc's run only on the local network on the left. In this case, all C-proc's use the same local network when they download pages from remote Web sites. Therefore, the network load from C-proc's is centralized at a single location where they operate.

- *Distributed crawler:* When a crawler's C-proc's run at geographically distant locations connected by the Internet (or a wide area network), we call it a *distributed crawler.* For example, one C-proc may run in the US, crawling all US pages, and another C-proc may run in France, crawling all European pages. As we discussed in the introduction, a distributed crawler can *disperse* and even *reduce* the load on the overall network.

  When C-proc's run at distant locations and communicate through the Internet, it becomes important how often and how much C-proc's need to communicate. The bandwidth between C-proc's may be limited and sometimes unavailable, as is often the case with the Internet.

When multiple C-proc's download pages in parallel, different C-proc's may download the same page multiple times. In order to avoid this overlap, C-proc's need to coordinate with each other on what pages to download. This coordination can be done in one of the following ways:

- *Independent:* At one extreme, C-proc's may download pages totally independently, without any coordination. That is, each C-proc starts with its own set of seed URLs and follows links without consulting with other C-proc's. In this scenario, downloaded pages may overlap, but we may hope that this overlap will not be significant, if all C-proc's start from different seed URLs.

While this scheme has minimal coordination overhead and can be very scalable, we do not directly study this option due to its overlap problem. Later we will consider an improved version of this option which significantly reduces overlap.

- *Dynamic assignment:* When there exists a central coordinator that logically divides the Web into small partitions (using a certain partitioning function) and dynamically assigns each partition to a C-proc for download, we call it *dynamic assignment.*

  For example, assume that a central coordinator partitions the Web by the site name of a URL. That is, pages in the same site (e.g., `http://cnn.com/top.html` and `http://cnn.com/content.html`) belong to the same partition, while pages in different sites belong to different partitions. Then during a crawl, the central coordinator constantly decides on what partition to crawl next (e.g., the site `cnn.com`) and sends URLs within this partition (that have been discovered so far) to a C-proc as seed URLs. Given this request, the C-proc downloads the pages and extracts links from them. When the extracted links point to pages in the same partition (e.g., `http://cnn.com/article.html`), the C-proc follows the links, but if a link points to a page in another partition (e.g., `http://nytimes.com/index.html`), the C-proc reports the link to the central coordinator. The central coordinator later uses this link as a seed URL for the appropriate partition.

  Note that the Web can be partitioned at various granularities. At one extreme, the central coordinator may consider every page as a separate partition and assign individual URLs to C-proc's for download. In this case, a C-proc does not follow links, because different pages belong to separate partitions. It simply reports all extracted URLs back to the coordinator. Therefore, the communication between a C-proc and the central coordinator may vary dramatically, depending on the granularity of the partitioning function.

- *Static assignment:* When the Web is partitioned and assigned to each C-proc *before* the start of a crawl, we call it *static assignment.* In this case, every C-proc knows which C-proc is responsible for which page during a crawl, and the crawler does not need

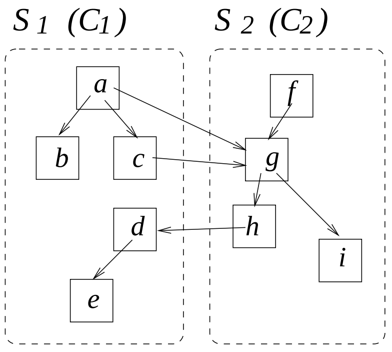


Figure 3.2: Site S_1 is crawled by C_1 and site S_2 is crawled by C_2

a central coordinator. We will shortly discuss in more detail how C-proc's operates under this scheme.

In this thesis, we mainly focus on static assignment and defer the study of dynamic assignment to future work. Note that in dynamic assignment, the central coordinator may become a major bottleneck because it has to maintain a large number of URLs reported from all C-proc's and has to constantly coordinate all C-proc's.

3.3 Crawling modes for static assignment

Under static assignment, each C-proc is responsible for a certain partition of the Web and has to download pages within the partition. However, some pages in the partition may have links to pages in another partition. We refer to this type of link as an *inter-partition link*. To illustrate how a C-proc may handle inter-partition links, we use Figure 3.2 as our example. In the figure, we assume that two C-proc's, C_1 and C_2, are responsible for sites S_1 and S_2, respectively. For now, we assume that the Web is partitioned by sites and that the Web has only S_1 and S_2. Also, we assume that each C-proc starts its crawl from the root page of each site, a and f.

1. **Firewall mode:** In this mode, each C-proc downloads only the pages within its partition and does not follow any inter-partition link. All inter-partition links are ignored and thrown away. For example, the links $a \to g$, $c \to g$ and $h \to d$ in Figure 3.2 are ignored and thrown away by C_1 and C_2.

   In this mode, the overall crawler does not have any overlap in the downloaded pages, because a page can be downloaded by only one C-proc, if ever. Also, C-proc's can run quite independently in this mode because they do not conduct any run-time coordination or URL exchanges. However, because some pages may be reachable only through inter-partition links, the overall crawler may not download all pages that it has to download. For example, in Figure 3.2, C_1 can download a, b and c, but not d and e, because they can be reached only through $h \rightarrow d$ link.

2. **Cross-over mode:** Primarily, each C-proc downloads pages within its partition, but when it runs out of pages in its partition, it also follows inter-partition links. For example, consider C_1 in Figure 3.2. Process C_1 first downloads pages a, b and c by following links from a. At this point, C_1 runs out of pages in S_1, so it follows a link to g and starts exploring S_2. After downloading g and h, it discovers a link to d in S_1, so it comes back to S_1 and downloads pages d and e.

   In this mode, downloaded pages may clearly overlap (pages g and h are downloaded twice), but the overall crawler can download more pages than the firewall mode (C_1 downloads d and e in this mode). Also, as in the firewall mode, C-proc's do not need to communicate with each other, because they follow only the links discovered by themselves.

3. **Exchange mode:** When C-proc's periodically and incrementally exchange inter-partition URLs, we say that they operate in an *exchange mode*. Processes do not follow inter-partition links.

   For example, C_1 in Figure 3.2 informs C_2 of page g after it downloads page a (and c) and C_2 transfers the URL of page d to C_1 after it downloads page h. Note that C_1 does not follow the links to page g. It only transfers the links to C_2, so that C_2 can download the page. In this way, the overall crawler can avoid overlap, while maximizing coverage.

Note that the firewall and the cross-over modes give C-proc's much independence (C-proc's do not need to communicate with each other), but the cross-over mode may download the same page multiple times, and the firewall mode may not download some pages. In contrast, the exchange mode avoids these problems but requires constant URL exchange among C-proc's. To reduce this URL exchange, a crawler based on the exchange mode may use some of the following techniques:

1. **Batch communication:** Instead of transferring an inter-partition URL immediately after it is discovered, a C-proc may wait for a while, to collect *a set of URLs* and send them *in a batch*. That is, with batching, a C-proc collects all inter-partition URLs until it downloads k pages. Then it partitions the collected URLs and sends them to an appropriate C-proc. Once these URLs are transferred, the C-proc then purges them and starts to collect a new set of URLs from the next downloaded pages. Note that a C-proc does *not* maintain the list of *all* inter-partition URLs discovered so far. It only maintains the list of inter-partition links in the *current* batch, in order to minimize the memory overhead for URL storage.

   This *batch communication* has various advantages over incremental communication. First, it incurs less communication overhead, because a set of URLs can be sent in a batch, instead of sending one URL per message. Second, the absolute number of exchanged URLs will also decrease. For example, consider C_1 in Figure 3.2. The link to page g appears twice, in page a and in page c. Therefore, if C_1 transfers the link to g after downloading page a, it needs to send the same URL again after downloading page c.[1] In contrast, if C_1 waits until page c and sends URLs in a batch, it needs to send the URL for g only once.

2. **Replication:** It is known that the number of *incoming* links to pages on the Web follows a Zipfian distribution [BKM$^+$00, BA99, Zip49]. That is, a small number of Web pages have an extremely large number of links pointing to them, while a majority of pages have only a small number of incoming links.

   Thus, we may significantly reduce URL exchanges, if we replicate the most "popular" URLs at each C-proc (by most popular, we mean the URLs with most incoming links) and stop transferring them between C-proc's. That is, before we start crawling pages, we identify the most popular k URLs based on the image of the Web collected in a *previous* crawl. Then we replicate these URLs at each C-proc, so that the C-proc's do not exchange them during a crawl. Since a small number of Web pages have a large number of incoming links, this scheme may significantly reduce URL exchanges between C-proc's, even if we replicate a small number of URLs.

   Note that some of the replicated URLs may be used as seed URLs for a C-proc. That is, if some URLs in the replicated set belong to the same partition that a C-proc is

[1]When it downloads page c, it does not remember whether the link to g has been already sent.

responsible for, the C-proc may use those URLs as its seeds rather than starting from other pages.

Another approach would be to have each C-proc try to discover popular URLs *on the fly* during a crawl, instead of identifying them based on a previous image. However, we believe that the popular URLs from a previous crawl will be a good approximation for the popular URLs in the current Web; most popular Web pages (such as Yahoo) maintain their popularity for a relatively long period of time, even if their exact popularity may change slightly.

So far, we have mainly assumed that the Web pages are partitioned by Web sites. Clearly, there exist a multitude of ways to partition the Web, including the following:

1. **URL-hash based:** Based on the hash value of the URL of a page, we assign the page to a C-proc. In this scheme, pages in the same site can be assigned to different C-proc's. Therefore, the locality of link structure[2] is not reflected in the partition, and there will be many inter-partition links.

2. **Site-hash based:** Instead of computing the hash value on the entire URL, we compute the hash value only on the *site name* of a URL (e.g., `cnn.com` in `http://cnn.com/index.html`) and assign the page to a C-proc.

   In this scheme, note that pages in the same site will be allocated to the *same* partition. Therefore, only some of *inter-site* links will be *inter-partition* links, and thus we can reduce the number of inter-partition links quite significantly compared to the URL-hash based scheme.

3. **Hierarchical:** Instead of using a hash-value, we may partition the Web hierarchically based on the URLs of pages. For example, we may divide the Web into three partitions (the pages in the `.com` domain, `.net` domain and all other pages) and allocate them to three C-proc's.

   Note that this scheme may have even fewer inter-partition links than the site-hash based scheme, because pages may tend to be linked to the pages in the same domain.

[2]Our experiments suggest that on average about 90% of the links in a page point to pages in the same site.

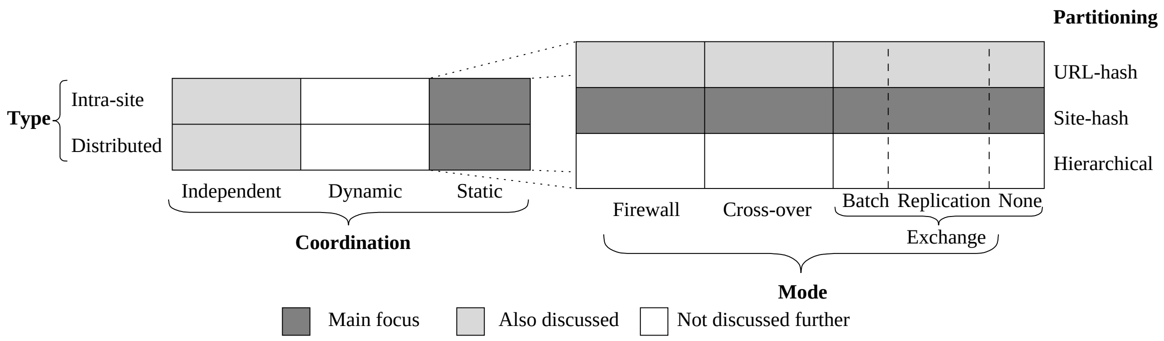


Figure 3.3: Summary of the options discussed

> However, in our preliminary experiments, we could not observe any significant difference between the two schemes, as long as each scheme splits the Web into roughly the same number of partitions.

In our later experiments, we will mainly use the site-hash based scheme as our partitioning function. We chose this option because it is simple to implement and because it captures the core issues that we want to study. For example, under the hierarchical scheme, it is not easy to divide the Web into equal size partitions, while it is relatively straightforward under the site-hash based scheme. Also, the URL-hash based scheme generates many inter-partition links, resulting in more URL exchanges in the exchange mode and less coverage in the firewall mode.

In Figure 3.3, we summarize the options that we have discussed so far. The right-hand table in the figure shows a more detailed view of the static coordination scheme. In the diagram, we highlight the main focus of this chapter with dark gray. That is, we mainly study the static coordination scheme (the third column in the left-hand table) and we use the site-hash based partitioning for our experiments (the second row in the right-hand table). However, during our discussion, we will also briefly explore the implications of other options. For instance, the firewall mode is an "improved" version of the independent coordination scheme (in the left-hand table), so our study on the firewall mode will show the implications of the independent coordination scheme. Also, we show the performance of the URL-hash based scheme (the first row in the right-hand table) when we discuss the results from the site-hash based scheme.

Given our table of the crawler design space, it would be very interesting to see what options existing search engines selected for their own crawlers. Unfortunately, this information is impossible to obtain in most cases because companies consider their technologies proprietary and want to keep them secret. The only crawler design that we know of is the prototype Google crawler [PB98] when it was developed at Stanford. At that time, it took the intra-site, static and site-hash based scheme and ran in the exchange mode [PB98]. Reference [HN99] describes the architecture of the Compaq SRC parallel crawler. However, the parallelization options were not explicitly described.

3.4 Evaluation models

In this section, we define the metrics that will let us quantify the advantages or disadvantages of different parallel crawling schemes. These metrics will be used later in our experiments.

1. **Overlap:** When multiple C-proc's are downloading Web pages simultaneously, it is possible that different C-proc's will download the same page multiple times. Multiple downloads of the same page are clearly undesirable.

   More precisely, we define the *overlap* of downloaded pages as $\frac{N-I}{I}$. Here, N represents the *total* number of pages downloaded by the overall crawler, and I represents the number of *unique* pages downloaded, again, by the overall crawler. Thus, the goal of a parallel crawler is to minimize the overlap.

   Note that a parallel crawler does not have an overlap problem if it is based on the firewall mode (Item 1 on page 34) or the exchange mode (Item 3 on page 35). In these modes, a C-proc downloads pages only within its own partition, so the overlap is always zero.

2. **Coverage:** When multiple C-proc's run independently, it is possible that they may not download all pages that they have to. In particular, a crawler based on the firewall mode (Item 1 on page 34) may have this problem because its C-proc's do not follow inter-partition links or exchange the links with others.

   To formalize this notion, we define the *coverage* of downloaded pages as $\frac{I}{U}$, where U represents the total number of pages that the overall crawler has to download and I is the number of unique pages downloaded by the overall crawler. For example, in Figure 3.2, if C_1 downloaded pages a, b and c, and if C_2 downloaded pages f through

i, the coverage of the overall crawler is $\frac{7}{9} = 0.77$ because it downloaded 7 pages out of 9.

3. **Quality:** As we discussed in Chapter 2, crawlers often cannot download the whole Web, and thus they try to download an "important" or "relevant" section of the Web. For example, if a crawler has storage space only for 1 million pages and if it uses *backlink counts* as its *importance metric*,[3] the goal of the crawler is to download the most highly-linked 1 million pages. For evaluation we adopt the *Crawl & Stop* model of Chapter 2, and use P_{CS}, the fraction of the top 1 million pages that are downloaded by the crawler, as its quality metric.

   Note that the quality of a parallel crawler may be worse than that of a single-process crawler, because many importance metrics depend on the global structure of the Web (e.g., backlink count): Each C-proc in a parallel crawler may know only the pages that are downloaded *by itself* and may make a poor crawling decision based solely on its own pages. In contrast, a single-process crawler knows *all* pages it has downloaded and may make a more informed decision.

   In order to avoid this quality problem, C-proc's need to periodically exchange information on page importance. For example, if the backlink count is the importance metric, a C-proc may periodically notify other C-proc's of how many pages in its partition have links to pages in other partitions.

   Note that this backlink exchange can be naturally incorporated in an exchange-mode crawler (Item 3 on page 35). In this mode, C-proc's exchange inter-partition URLs periodically, so a process C_1 may send a message like `[http://cnn.com/index.html, 3]` to process C_2 to notify C_2 that C_1 has seen 3 links to the page that C_2 is responsible for. On receipt of this message, C_2 can properly adjust the estimated importance of the page. By incorporating this scheme, an exchange-mode crawler may achieve better quality than a firewall mode or cross-over mode crawler.

   However, note that the quality of an exchange-mode crawler may vary depending on how often it exchanges backlink messages. For instance, if C-proc's exchange backlink messages after every page download, they will have essentially the same backlink information as a single-process crawler does. (They know backlink counts from *all* pages that have been downloaded.) Therefore, the quality of the downloaded pages

[3]Backlink counts and importance metrics were described in Chapter 2.

Mode	*Coverage*	*Overlap*	*Quality*	*Communication*
Firewall	Bad	Good	Bad	Good
Cross-over	Good	Bad	Bad	Good
Exchange	Good	Good	Good	Bad

Table 3.1: Comparison of three crawling modes

would be virtually the same as that of a single-process crawler. In contrast, if C-proc's rarely exchange backlink messages, they do not have "accurate" backlink counts from the downloaded pages, so they may make poor crawling decisions, resulting in poor quality. Later, we will study how often C-proc's should exchange backlink messages in order to maximize quality.

4. **Communication overhead:** The C-proc's in a parallel crawler need to exchange messages to coordinate their work. In particular, C-proc's based on the exchange mode (Item 3 on page 35) swap their inter-partition URLs periodically. To quantify how much communication is required for this exchange, we define *communication overhead* as the average number of inter-partition URLs exchanged per downloaded page. For example, if a parallel crawler has downloaded 1,000 pages in total and if its C-proc's have exchanged 3,000 inter-partition URLs, its communication overhead is $3,000/1,000 = 3$. Note that crawlers based on the the firewall and the cross-over mode do not have any communication overhead because they do not exchange any inter-partition URLs.

In Table 3.1 we compare the relative merits of the three crawling modes (Items 1–3 on page 34). In the table, "Good" means that the mode is expected to perform relatively well for that metric, and "Bad" means that it may perform worse compared to other modes. For instance, the firewall mode does not exchange any inter-partition URLs (Communication: Good) and downloads pages only once (Overlap: Good), but it may not download every page (Coverage: Bad). Also, because C-proc's do not exchange inter-partition URLs, the downloaded pages may be of lower quality than those of an exchange-mode crawler. Later, we will examine these issues more quantitatively through experiments based on real Web data.

3.5 Description of dataset

We have discussed various issues related to parallel crawlers and identified multiple alternatives for their design. In the remainder of this chapter, we quantitatively study these issues through experiments conducted on real Web data.

In all of the following experiments, we used the 40 million Web pages in our Stanford WebBase repository that was constructed in December 1999. Because the properties of this dataset may significantly impact the result of our experiments, readers might be interested in how we collected these pages.

We downloaded the pages using our Stanford WebBase crawler in December 1999 for a period of 2 weeks. In downloading the pages, the WebBase crawler started with the URLs listed in Open Directory (`http://www.dmoz.org`) and followed links. We decided to use the Open Directory URLs as seed URLs because these pages are the ones that are considered "important" by its maintainers. In addition, some of our local WebBase users were keenly interested in the Open Directory pages and explicitly requested that we cover them. The total number of URLs in the Open Directory was around 1 million at that time. Conceptually, the WebBase crawler downloaded all these pages, extracted the URLs within the downloaded pages, and followed the links in a breadth-first manner. (The WebBase crawler used various techniques to expedite and prioritize the crawling process, but we believe these optimizations do not affect the final dataset significantly.)

Because our dataset was downloaded by a crawler in a particular way, our dataset may not correctly represent the *actual* Web as it is. In particular, our dataset may be biased towards more "popular pages" because we started from the Open Directory pages. Also, our dataset does *not* cover the pages that are accessible *only* through a query interface. For example, our crawler did not download pages generated by keyword-based search engines because it did not try to "guess" appropriate keywords to fill in. However, we emphasize that many of the dynamically-generated pages were still downloaded by our crawler. For example, the pages on the Amazon Web site (`http://amazon.com`) are dynamically generated, but we could still download most of the pages on the site by following links.

In summary, our dataset may not necessarily reflect the actual image of the Web, but we believe it represents the image that a *parallel crawler would see* in its crawl. In most cases, crawlers are mainly interested in downloading "popular" or "important" pages, and they download pages by following links, which is what we did for our data collection.

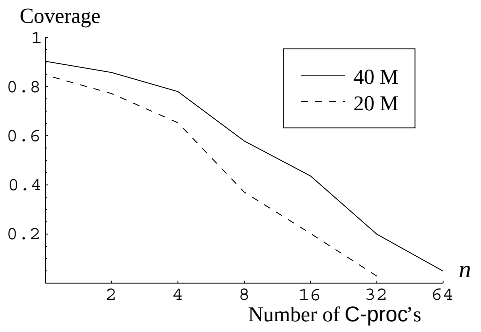


Figure 3.4: Number of processes vs. Coverage

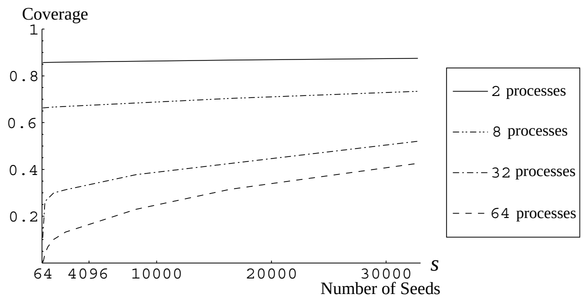


Figure 3.5: Number of seed URLs vs. Coverage

3.6 Firewall mode and coverage

As we discussed in Section 3.4, a firewall-mode crawler (Item 1 on page 34) has minimal communication overhead, but it may have coverage and quality problems. In this section, we quantitatively study the effectiveness of a firewall-mode crawler using the 40 million pages in our repository. In particular, we estimate the coverage (Section 3.4, Item 2) of a firewall-mode crawler when it employs n C-proc's in parallel.

In our experiments, we considered the 40 million pages within our WebBase repository as the entire Web, and we used site-hash based partitioning (Item 2 on page 37). As seed URLs, each C-proc was given 5 *random* URLs from its own partition, so $5n$ seed URLs were used in total by the overall crawler.[4] Since the crawler ran in a firewall mode, C-proc's followed only *intra-partition* links. Under these settings, we ran C-proc's until they ran out of URLs, and we measured the overall coverage at the end.

In Figure 3.4, we summarize the results from the experiments. The horizontal axis represents n, the number of parallel C-proc's, and the vertical axis shows the coverage of the overall crawler for the given experiment. The solid line in the graph is the result from the 40M-page experiment.[5] Note that the coverage is only 0.9 even when $n = 1$ (a single-process case). This happened because the crawler in our experiment started with only 5 URLs, while the actual dataset was collected with 1 million seed URLs. Some of the 40 million pages were unreachable from the 5 seed URLs.

From the figure it is clear that the coverage decreases as the number of processes increases. This happens because the number of inter-partition links increases as the Web is

[4]We discuss the effect of the number of seed URLs shortly.
[5]The dashed line will be explained later.

split into smaller partitions, and thus more pages are reachable only through inter-partition links.

From this result we can see that a firewall-mode crawler gives good coverage when it runs 4 or fewer C-proc's. For example, for the 4-process case, the coverage decreases only 10% from the single-process case. At the same time, we can also see that the firewall-mode crawler yields quite low coverage when a large number of C-proc's run. Less than 10% of the Web can be downloaded when 64 C-proc's run together, each starting with 5 seed URLs.

Clearly, coverage may depend on the number of seed URLs that each C-proc starts with. To study this issue, we also ran experiments varying the number of seed URLs, s. We show the results in Figure 3.5. The horizontal axis in the graph represents s, the *total* number of seed URLs that the overall crawler used, and the vertical axis shows the coverage for that experiment. For example, when $s = 128$, the overall crawler used 128 total seed URLs, so each C-proc started with 2 seed URLs for the 64 C-proc case. We performed the experiments for 2, 8, 32, 64 C-proc cases and plotted their coverage values. From this figure, we can observe the following trends:

- *When a large number of* C-proc*'s run in parallel, (e.g.,* 32 *or* 64*), the total number of seed URLs affects coverage very significantly.* For example, when 64 processes run in parallel the coverage value jumps from 0.4% to 10% if the number of seed URLs increases from 64 to 1024.

- *When only a small number of processes run in parallel (e.g.,* 2 *or* 8*), coverage is not significantly affected by the number of seed URLs.* The coverage increase in these cases is marginal.

Based on these results, we draw the following conclusions:

1. When a relatively small number of C-proc's run in parallel, a crawler using the firewall mode provides good coverage. In this case, the crawler may start with only a small number of seed URLs because coverage is not much affected by the number of seed URLs.

2. The firewall mode is not a good choice if the crawler wants to download every single page on the Web. The crawler may miss some portion of the Web, particularly when it runs many C-proc's in parallel.

Our results in this section are based on a 40 million page dataset, so it is important to consider how coverage might change with a different dataset, or equivalently, how it might change as the Web grows or evolves. Unfortunately, it is difficult to predict how the Web will grow. On one hand, if all "newly created" pages are well connected to existing pages at their creation site, then coverage will increase. On the other hand, if new pages tend to form disconnected groups, the overall coverage will decrease. Depending on how the Web grows, coverage could go either way.

As a preliminary study of the growth issue, we conducted the same experiments with a subset of 20M pages and measured how the coverage changes. We first randomly selected half of the *sites* in our dataset, and ran the experiments using only the pages from those sites. Thus, one can roughly view the smaller dataset as a smaller Web, that then "over time" doubled its number of sites to yield the second dataset. The dotted line in Figure 3.4 shows the results from the 20M-page experiments. From the graph we can see that as "our Web doubles in size," one can double the number of C-proc's and retain roughly the same coverage. That is, the new sites can be visited by new C-proc's without significantly changing the coverage they obtain. If the growth did not come exclusively from new sites, then one should not quite double the number of C-proc's each time the Web doubles in size, to retain the same coverage.

Example 3.1 (Generic search engine) To illustrate how our results could guide the design of a parallel crawler, consider the following example. Assume that to operate a Web search engine we need to download 1 billion pages in one month. Each machine that we run our C-proc's on has a 10 Mbps link to the Internet, and we can use as many machines as we want.

Given that the average size of a Web page is around 10K bytes, we roughly need to download $10^4 \times 10^9 = 10^{13}$ bytes in one month. This download rate corresponds to 34 Mbps, and we need 4 machines (thus 4 C-proc's) to obtain the rate. If we want to be conservative, we can use the results of our 40M-page experiment (Figure 3.4) and estimate that the coverage will be at least 0.8 with 4 C-proc's. Therefore, in this scenario, the firewall mode may be good enough unless it is very important to download the "entire" Web. □

Example 3.2 (High freshness) As a second example, let us now assume that we have strong "freshness" requirement on the 1 billion pages and need to revisit every page once every week, not once every month. This new scenario requires approximately 140 Mbps for

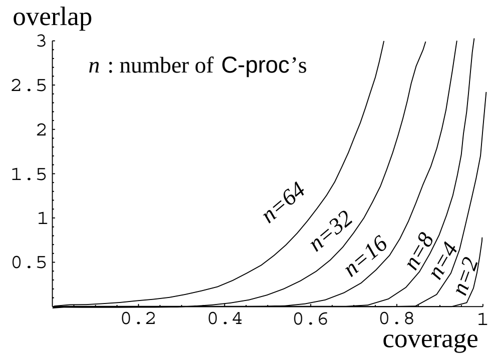


Figure 3.6: Coverage vs. Overlap for a cross-over mode crawler

page download, and we need to run 14 C-proc's. In this case, the coverage of the overall crawler decreases to less than 0.5 according to Figure 3.4. Of course, the coverage could be larger than our conservative estimate, but to be safe one would probably want to consider using a crawler mode different than the firewall mode. □

3.7 Cross-over mode and overlap

In this section, we study the effectiveness of a cross-over mode crawler (Item 2 on page 35). A cross-over crawler may yield improved coverage of the Web, since it follows inter-partition links when a C-proc runs out of URLs in its own partition. However, this mode incurs overlap in downloaded pages (Section 3.4, Item 1) because a page can be downloaded by multiple C-proc's.

In Figure 3.6, we show the relationship between the coverage and the overlap of a cross-over mode crawler obtained from the following experiments. We partitioned the 40M pages using site-hash partitioning and assigned them to n C-proc's. Each of the n C-proc's then was given 5 random seed URLs from its partition and followed links in the cross-over mode. During this experiment, we measured how much overlap the overall crawler incurred when its coverage reached various points. The horizontal axis in the graph shows the coverage at a particular time and the vertical axis shows the overlap at the given coverage. We performed the experiments for $n = 2, 4, \ldots, 64$.

Note that in most cases the overlap stays at zero until the coverage becomes relatively large. For example, when $n = 16$, the overlap is zero until the coverage reaches 0.5. We can understand this result by looking at the graph in Figure 3.4. According to that graph, a crawler with 16 C-proc's can cover around 50% of the Web by following only intra-partition links. Therefore, even a cross-over mode crawler will follow only intra-partition links until its coverage reaches that point. Only after that point will each C-proc start to follow inter-partition links, thus increasing the overlap. If we adopted the independent model (Item 3.2 on page 32), the crawler would have followed inter-partition links even before it ran out of intra-partition links, so the overlap would have been worse at the same coverage.

While the cross-over crawler is better than the independent-model-based crawler, it is clear that the cross-over crawler still incurs quite significant overlap. For example, when 4 C-proc's run in parallel in the cross-over mode, the overlap becomes almost 2.5 to obtain coverage close to 1. For this reason, we do not recommend the cross-over mode unless it is absolutely necessary to download every page without any communication between C-proc's.

3.8 Exchange mode and communication

To avoid the overlap and coverage problems, an exchange-mode crawler (Item 3 on page 35) constantly exchanges inter-partition URLs between C-proc's. In this section, we study the communication overhead (Section 3.4, Item 4) of an exchange-mode crawler and how much we can reduce it by replicating the most popular k URLs.

For now, let us assume that a C-proc *immediately* transfers inter-partition URLs. (We will discuss batch communication later when we discuss the quality of a parallel crawler.)

In the experiments, again, we split the 40 million pages into n partitions based on site-hash values and ran n C-proc's in the exchange mode. At the end of the crawl, we measured how many URLs had been exchanged during the crawl. We show the results in Figure 3.7. For comparison purposes, the figure also shows the overhead for a URL-hash based scheme, although the curve is clipped at the top because of its large overhead values. In the figure, the horizontal axis represents the number of parallel C-proc's, n, and the vertical axis shows the communication overhead (the average number of URLs transferred per page).

To explain the graph, we first note that an average page has 10 out-links, and about 9 of them point to pages in the *same* site. Therefore, the 9 links are internally followed by a C-proc under site-hash partitioning. Only the remaining 1 link points to a page in a different site and may be exchanged between processes. Figure 3.7 indicates that this URL exchange increases with the number of processes. For example, the C-proc's exchanged 0.4 URLs per page when 2 processes ran, while they exchanged 0.8 URLs per page when 16

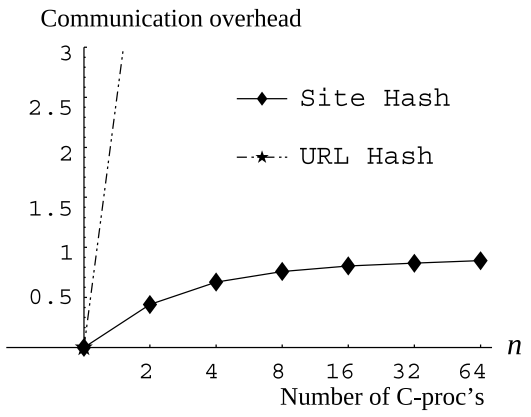


Figure 3.7: Number of crawling processes vs. Number of URLs exchanged per page

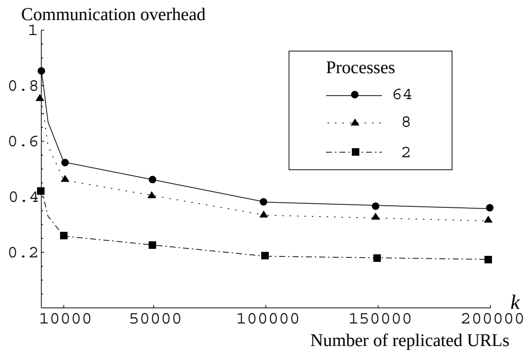


Figure 3.8: Number of replicated URLs vs. Number of URLs exchanged per page

processes ran. Based on the graph, we draw the following conclusions:

- *The site-hash based partitioning scheme significantly reduces communication overhead* compared to the URL-hash based scheme. We need to transfer only *up to one link per page* (or 10% of the links), which is significantly smaller than the URL-hash based scheme. For example, when we ran 2 C-proc's using the URL-hash based scheme the crawler exchanged 5 links per page under the URL-hash based scheme, which was significantly larger than 0.5 links per page under the site-hash based scheme.

- *The network bandwidth used for the URL exchange is relatively small, compared to the actual page download bandwidth.* Under the site-hash based scheme, at most 1 URL is exchanged per page, which is about 40 bytes.[6] Given that the average size of a Web page is 10 KB, the URL exchange consumes less than 40/10K = 0.4% of the total network bandwidth.

- However, *the overhead of the URL exchange on the overall system can be quite significant.* The processes need to exchange up to one message per page, and the message has to go through the TCP/IP network stack at the sender *and* the receiver. Thus it is copied to and from kernel space twice, incurring two context switches between the kernel and the user mode. Since these operations pose significant overhead even if the message size is small, the overall overhead can be important if the processes exchange one message per every downloaded page.

[6]In our estimation, an average URL was about 40 bytes long.

In order to study how much we can reduce this overhead by replication (Item 2 on page 36), in Figure 3.8 we show the communication overhead when we replicate the top k popular URLs. In the figure, the horizontal axis shows the number of replicated URLs, k, and the vertical axis shows the communication overhead.

Remember that in a real-world scenario we would identify the most popular URLs based on the image of the Web from a previous crawl. However, in our experiment we identified the top k URLs based on the 40 million pages in our *current* WebBase repository, which was also used for our experiments. Therefore, there had been no change over time, and the replicated URLs were *exactly* the most popular ones in the repository. For this reason, in a real-world scenario, the actual communication overhead might be slightly worse than what our results show.

From Figure 3.8 it is clear that we can significantly reduce the communication overhead by replicating a relatively small number of URLs. For example, when 64 C-proc's run in parallel, the overhead reduces from 0.86 URLs/page to 0.52 URLs/page (40% reduction) when we replicate only $10,000$ URLs. From the figure, we can also see that this reduction diminishes as the number of replicated URLs increases. For example, when we replicate $100,000$ URLs, we can get about 51% reduction (from 0.86 to 0.42, the 64 process case), while we can get only 53% reduction when we replicate $200,000$ URLs. Based on this result, we recommend replicating between $10,000$ and $100,000$ URLs in each C-proc in order to minimize the communication overhead while maintaining a low replication overhead.

While our experiments were based on 40 million pages, the results will be similar even if we were to run our experiments on a larger dataset. Note that our results depend on the fraction of the links pointing to the top k URLs. We believe that the links in our experiments are good samples for the Web because the pages were downloaded from the Web and the links in the pages were created by independent authors. Therefore a similar fraction of links would point to the top k URLs even if we download more pages or, equivalently, if the Web grows over time.

3.9 Quality and batch communication

As we discussed, the quality (Section 3.4, Item 3) of a parallel crawler can be worse than that of a single-process crawler because each C-proc may make crawling decisions solely based on the information collected within its own partition. We now study this quality issue and the impact of the batch communication technique (Item 1 on page 36) on quality.

Throughout the experiments in this section, we assume that the crawler uses the *backlink counts*, $IB(p)$, as its importance metric, and it uses $IB'(p)$ as its ordering metric.[7] We use these metrics because they are easy to implement and test and we believe they capture the core issues that we want to study. In particular, note that the $IB(p)$ metric depends on the global structure of the Web. If we use an importance metric that solely depends on a page itself, not on the global structure of the Web, each C-proc in a parallel crawler will be able to make good decisions based on the pages that it has downloaded. The quality of a parallel crawler therefore will be essentially the same as that of a single crawler.

Under the $IB'(p)$ ordering metric, note that the C-proc's need to periodically exchange `[URL, backlink count]` messages so that each C-proc can incorporate backlink counts from *inter-partition* links. Depending on how often they exchange the messages, the quality of the downloaded pages will differ. For example, if the C-proc's *never* exchange messages, the quality will be the same as that of a firewall-mode crawler. If they exchange messages after every downloaded page, the quality will be similar to that of a single-process crawler.

To study these issues, we compared the quality of the downloaded pages when C-proc's exchanged backlink messages at various intervals under the *Crawl & Stop model* of page 11. Figures 3.9(a), 3.10(a) and 3.11(a) show the quality, P_{CS}, achieved by the overall crawler when it downloaded a total of 500K, 2M, and 8M pages, respectively. The horizontal axis in the graphs represents *the total number of URL exchanges* during a crawl, x, and the vertical axis shows the quality for the given experiment. For example, when $x = 1$, the C-proc's exchanged backlink counts *only once* in the middle of the crawl. Therefore, the case when $x = 0$ represents the quality of a firewall-mode crawler, and the case when $x \to \infty$ shows the quality of a single-process crawler. In Figures 3.9(b), 3.10(b) and 3.11(b), we also show the communication overhead (Item 4 on page 41), which is the average number of `[URL, backlink count]` pairs exchanged per a downloaded page.

From these figures, we can observe the following trends:

- *As the number of crawling processes increases, the quality of the downloaded pages becomes worse unless they exchange backlink messages often.* For example, in Figure 3.9(a), the quality achieved by a 2-process crawler (0.12) is significantly higher than that of a 64-process crawler (0.025) in the firewall mode ($x = 0$). Again, this

[7]The $IB(p)$ importance metric and the $IB'(p)$ ordering metric were discussed in Chapter 2 on page 9.

happens because each C-proc learns less about the global backlink counts when the Web is split into smaller parts.

- *The quality of the firewall-mode crawler ($x = 0$) is significantly worse than that of the single-process crawler ($x \to \infty$) when the crawler downloads a relatively small fraction of the pages (Figure 3.9(a) and 3.10(a)).* However, the difference is not very significant when the crawler downloads a relatively large fraction (Figure 3.11(a)). In other experiments, when the crawler downloaded more than 50% of the pages, the difference was almost negligible in any case. Intuitively, this result makes sense because quality is an important issue only when the crawler downloads a small portion of the Web. (If the crawler will visit all pages anyway, quality is not relevant.)

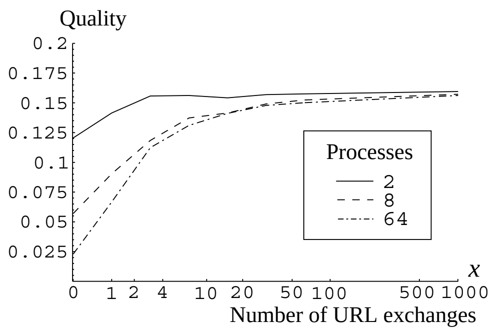
- *The communication overhead does not increase linearly as the number of URL exchanges increases.* The graphs in Figure 3.9(b), 3.10(b) and 3.11(b) are not straight lines. This is because a popular URL will appear multiple times between backlink exchanges. Therefore, a popular URL can be transferred as *one* entry (URL and its backlink count) in the exchange, even if it has appeared multiple times. This reduction increases as C-proc's exchange backlink messages less frequently.

- *One does not need a large number of URL exchanges to achieve high quality.* Through multiple experiments, we tried to identify how often C-proc's should exchange backlink messages to achieve the highest quality value. From these experiments, we found that a parallel crawler can get the highest quality values even if its processes communicate *less than 100 times* during a crawl.

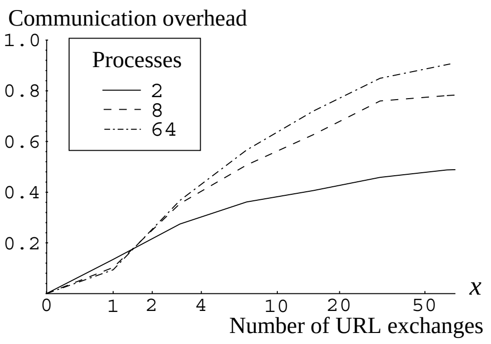
We use the following example to illustrate how one can use the results of our experiments.

Example 3.3 (Medium-Scale Search Engine) Say we plan to operate a medium-scale search engine, and we want to maintain about 20% of the Web (200 M pages) in our index. Our plan is to refresh the index once a month. Each machine that we can use has a separate T1 link (1.5 Mbps) to the Internet.

In order to update the index once a month, we need about 6.2 Mbps download bandwidth, so we have to run at least 5 C-proc's on 5 machines. According to Figure 3.11(a) (20% download case), we can achieve the highest quality if the C-proc's exchange backlink messages 10 times during a crawl when 8 processes run in parallel. (We use the 8 process case because it is the closest number to 5). Also, from Figure 3.11(b), we can see that

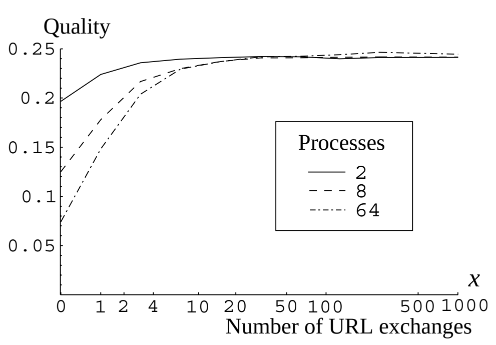


(a) URL exchange vs. Quality

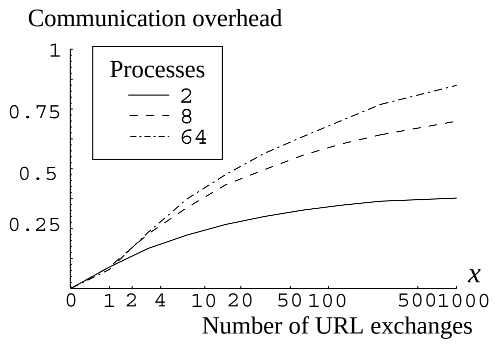


(b) URL exchange vs. Communication

Figure 3.9: Crawlers downloaded 500K pages (1.2% of 40M)

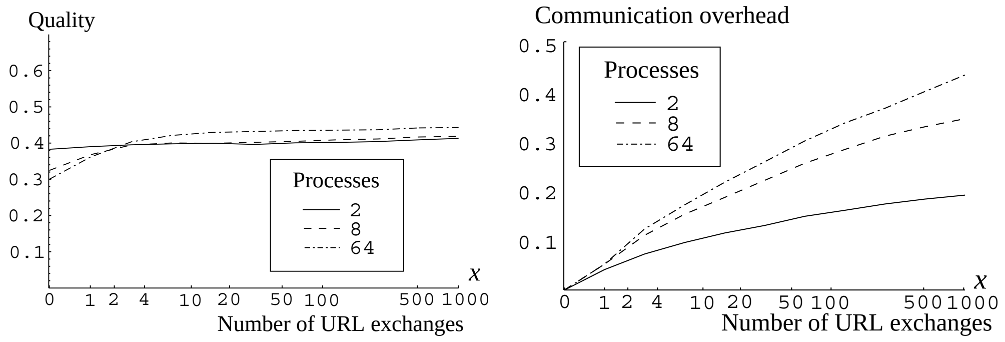


(a) URL exchange vs. Quality



(b) URL exchange vs. Communication

Figure 3.10: Crawlers downloaded 2M pages (5% of 40M)



(a) URL exchange vs. Quality (b) URL exchange vs. Communication

Figure 3.11: Crawlers downloaded 8M pages (20% of 40M)

when C-proc's exchange messages 10 times during a crawl they need to exchange fewer than $0.17 \times 200\text{M} = 34\text{M}$ pairs of `[URL, backlink count]` in total. Therefore, the total network bandwidth used by the backlink exchange is only $(34\text{M} \cdot 40)/(200\text{M} \cdot 10\text{K}) \approx 0.06\%$ of the bandwidth used by actual page downloads. Also, since the exchange happens only 10 times during a crawl, the context-switch overhead for message transfers (discussed on page 48) is minimal.

Note that in this scenario we need to exchange 10 backlink messages in one month or one message every three days. Therefore, even if the connection between C-proc's is unreliable or sporadic, we can still use the exchange mode without any problem. □

3.10 Related work

References [PB98, HN99, CGM00a, Mil98, Eic94] describe the general architecture of a Web crawler and studies how a crawler works. For example, Reference [HN99] describes the architecture of the Compaq SRC crawler and its major design goals. Some of these studies briefly describe how the crawling task is parallelized. For instance, Reference [PB98] describes a crawler that distributes individual URLs to multiple machines, which download Web pages in parallel. The downloaded pages are then sent to a central machine, on which links are extracted and sent back to the crawling machines. However, in contrast to our work, these studies do not try to understand the fundamental issues related to a parallel crawler and how various design choices affect performance. In this thesis, we first identified multiple techniques for a parallel crawler and compared their relative merits carefully using real Web data.

There also exists a significant body of literature studying the general problem of parallel and distributed computing [MDP$^+$00, OV99, QD84, TR85]. Some of these studies focus on the design of efficient parallel algorithms. For example, References [QD84, NS82, Hir76] present various architectures for parallel computing, propose algorithms that solve various problems (e.g., finding maximum cliques) under the architecture, and study the complexity of the proposed algorithms. While the general principles described are being used in our work,[8] none of the existing solutions can be directly applied to the crawling problem.

Another body of literature designs and implements *distributed operating systems*, where a process can use distributed resources transparently (e.g., distributed memory, distributed file systems) [TR85, SKK$^+$90, ADN$^+$95, LH89]. Clearly, such OS-level support makes it easy to build a general distributed application, but we believe that we cannot simply run a centralized crawler on a distributed OS to achieve parallelism. A web crawler contacts millions of web sites in a short period of time and consumes extremely large network, storage and memory resources. Since these loads push the limit of existing hardwares, the task should be carefully partitioned among processes and they should be carefully coordinated. Therefore, a general-purpose distributed operating system that does not understand the semantics of web crawling will lead to unacceptably poor performance.

3.11 Conclusion

In this chapter, we studied various strategies and design choices for a parallel crawler.

As the size of the Web grows, it becomes increasingly important to use a parallel crawler. Unfortunately, almost nothing is known (at least in the open literature) about options for parallelizing crawlers and their performance using 40 million pages downloaded from the Web. This chapter addressed this shortcoming by presenting several design choices and strategies for parallel crawlers and by studying their performance. We believe this chapter offers some useful guidelines for crawler designers, helping them, for example, select the right number of crawling processes or select the proper inter-process coordination scheme.

In summary, the main conclusions of our study were the following:

[8]For example, we may consider that our proposed solution is a variation of "divide and conquer" approach, since we partition and assign the Web to multiple processes.

- When a small number of crawling processes run in parallel (in our experiment, fewer than or equal to 4), the firewall mode provides good coverage. Given that firewall-mode crawlers can run totally independently and are easy to implement, we believe that it is a good option to consider. The cases when the firewall mode might not be appropriate are:

  1. when we need to run more than 4 crawling processes
     or
  2. when we download only a small subset of the Web and the quality of the downloaded pages is important.

- A crawler based on the exchange mode consumes small network bandwidth for URL exchanges (less than 1% of the network bandwidth). It can also minimize other overheads by adopting the batch communication technique. In our experiments, the crawler could maximize the quality of the downloaded pages, even if it exchanged backlink messages fewer than 100 times during a crawl.

- By replicating between $10,000$ and $100,000$ popular URLs, we can reduce the communication overhead by roughly 40%. Replicating more URLs does not significantly reduce the overhead.

Chapter 4

Web Evolution Experiments

4.1 Introduction

Because Web pages constantly change, a crawler should periodically refresh downloaded pages, so that the pages are maintained up to date. In Chapters 4 through 6, we study how a crawler can effectively refresh downloaded pages to maximize their "freshness."

To study this problem, it is imperative to understand how the Web pages change over time, because the effectiveness of various refresh policies may depend on the change frequency. For example, if Web pages change at vastly different rates, we may improve the "freshness" of the downloaded pages by refreshing them at varying frequencies, depending on how often the pages change. In contrast, if most Web pages change at similar frequencies, it may be acceptable to visit the pages exactly at the same frequency.

In this chapter, therefore, we present our experimental results that show how often Web pages change, how the Web as a whole evolves over time, and how we can model page changes. Based on this understanding, in subsequent chapters we develop good refresh policies that can maximize the "freshness" of downloaded pages.

4.2 Experimental setup

Through our experiment, we try 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 the changes of Web pages by a mathematical model?

Note that a Web 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 refresh the page. The techniques used for our experiment will shed light on how a 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. Again, this experiment was done with the Stanford WebBase crawler, a system designed to create and maintain large Web repositories. In this section we briefly discuss how the particular sites and pages were selected.

4.2.1 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 would have forced 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.

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

Table 4.1: Number of sites within a domain

4.2.2 Site selection

To select the actual sites for our experiment, we used the snapshot of 25 million Web pages in our WebBase repository in December 1999. Based on this snapshot, we identified the top 400 "popular" sites as the candidate sites (The definition of a "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.

To measure the popularity of a site, we used a modified PageRank metric. As we discussed in Section 2.2, the PageRank metric considers a page "popular" if it is linked to by many other Web pages. Roughly, 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. However, note that the PageRank computes the popularity of Web *pages* not of Web *sites*, so we need to slightly modify the definition of the PageRank. To do that, we first construct a hypergraph, where the nodes correspond to Web *sites* and the edges correspond to the links between the *sites*. Then for this hypergraph, we can define a PR value for each node (site) using the formula above. The value for a site then gives us the measure of the popularity of the Web site.

In Table 4.1, we show how many sites in our list are from which domain. In our site list, 132 sites belong to .com (`com`) and 78 sites to .edu (`edu`). The sites ending with ".net" and ".org" are classified as `netorg` and the sites ending with ".gov" and ".mil" as `gov`.

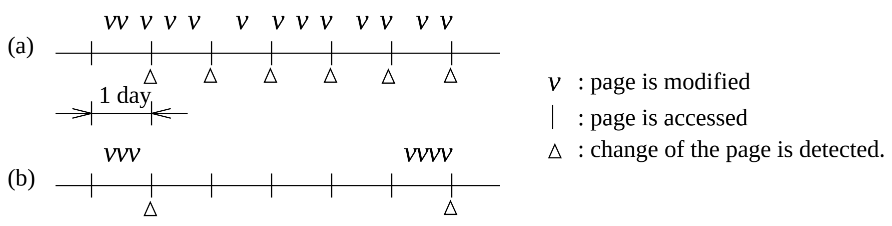


Figure 4.1: The cases when the estimated change interval is lower than the real value

4.2.3 Number of pages at each site

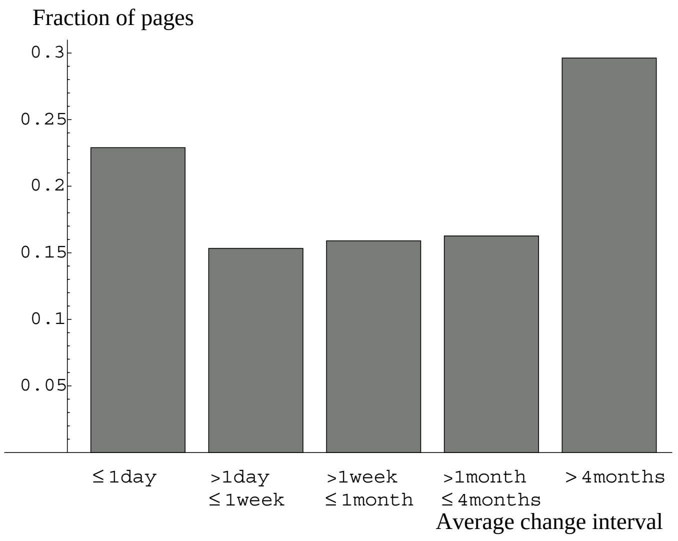
After selecting the Web sites to monitor, we still need to decide the window of pages to crawl from each site. In our experiment, we crawled 3,000 pages at each site. That is, starting from the root pages of the selected sites we followed links in a breadth-first search, up to 3,000 pages per site. This "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.

4.3 How often does a page change?

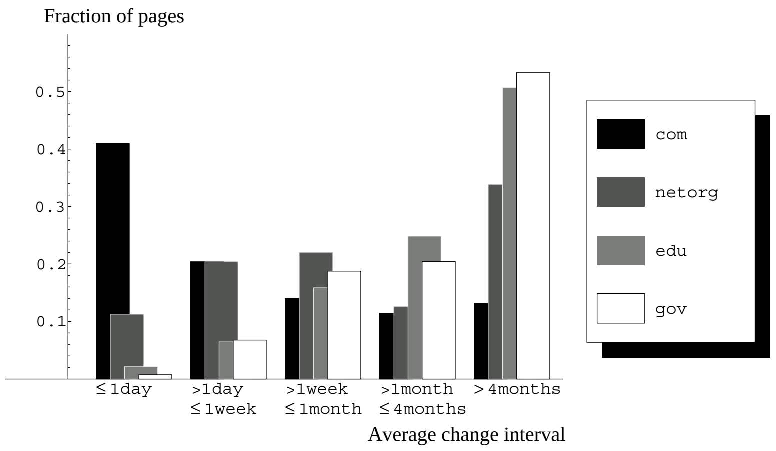
From the experiment described in the previous section, we collected statistics on how often pages change and how long they stay on the Web, and we report the results in the following sections.

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 4.1(a)). Also, if a page changes several times a day and then remains unchanged, say, for a week (Figure 4.1(b)), the estimated interval might be much longer than the true value. Later in Chapter 6, we will discuss how we can account for these "missed" changes and estimate the change frequency more accurately, but for now we assume the described estimation method gives us a good picture on how often Web pages change.

In Figure 4.2 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



(a) Over all domains



(b) For each domain

Figure 4.2: Fraction of pages with given average interval of change

pages changed at the given average interval. Figure 4.2(a) shows the statistics collected over all domains, and Figure 4.2(b) shows the statistics broken down to each domain. For instance, from the second bar of Figure 4.2(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 4.2(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 4.2(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 4.2(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 4.2(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 pages, 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 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 4.2(a)). As a crude approximation, if we assume that the pages in the first bar change every day and the pages in the fifth bar change every year, 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.

4.4 What is the lifespan of a page?

In this section 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

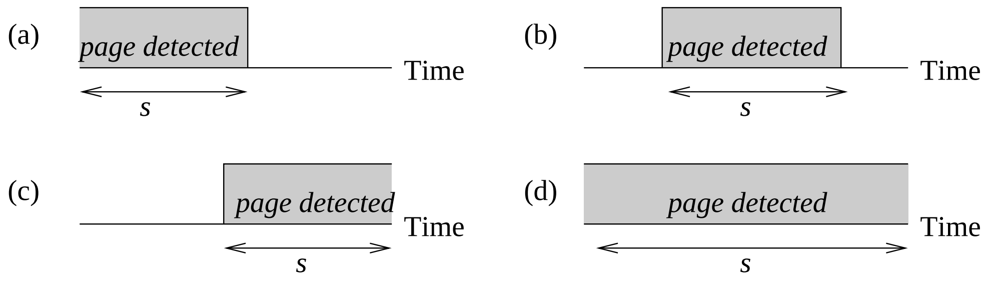
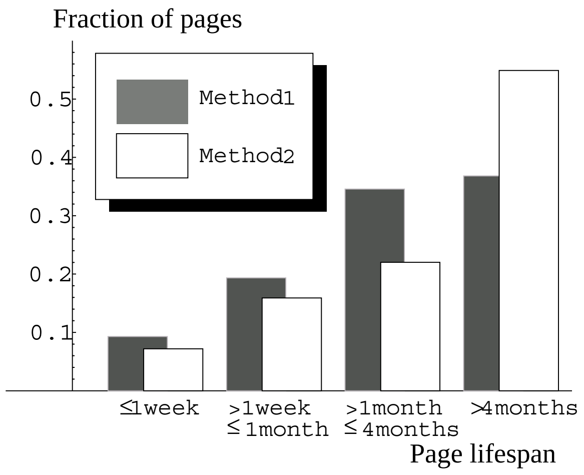


Figure 4.3: Issues in estimating the lifespan of a page

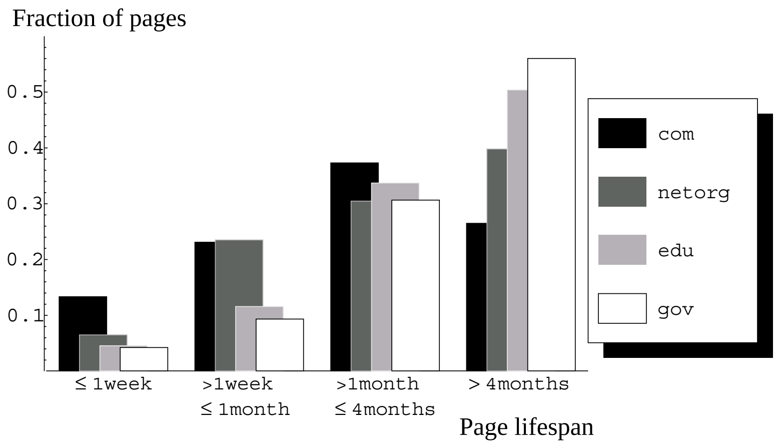
approximation to the lifespan of a page *conceived by users* of the Web. That is, when a user looks for information from a particular site, she often starts from its root page and follows links. Since the user cannot infinitely follow links, she concludes that the page of interest does not exist or has 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 4.3 illustrates the problem in detail. For the pages that appeared *and* disappeared during our experiment (Figure 4.3(b)), we can measure how long the page stayed in our window precisely. However, for the pages that existed from the beginning (Figure 4.3(a) and (d)) or at the end of our experiment (Figure 4.3(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 lifespan in two different ways. First, we used the length s in Figure 4.3 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 4.4(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 a given lifespan. For instance, from the second bar of Figure 4.4(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

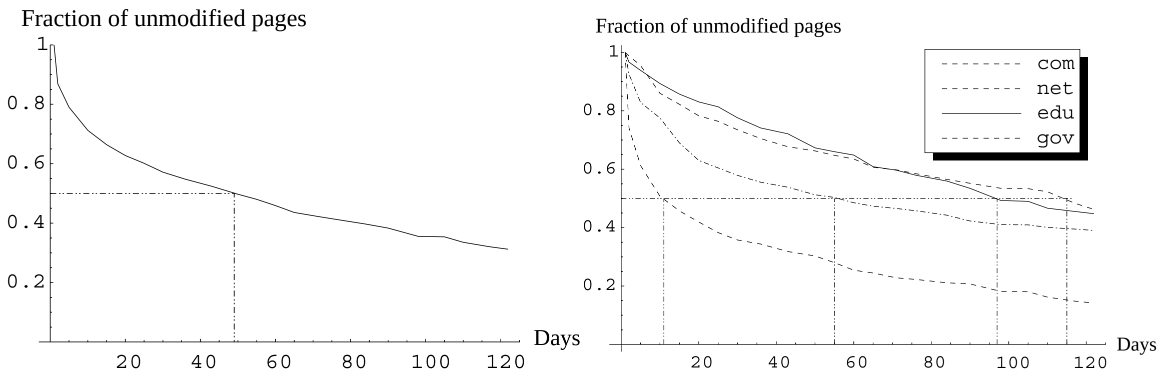


(a) Over all domains



(b) For each domain

Figure 4.4: Percentage of pages with given visible lifespan



(a) Over all domains (b) For each domain

Figure 4.5: Fraction of pages that did not change or disappear until given date.

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 4.4(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 4.4(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 4.4(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.

4.5 How long does it take for 50% of the Web to change?

In the previous sections, 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 4.5. In the figure, the horizontal axis shows the number of days from the beginning of the experiment, and the vertical axis shows the fraction of pages that were unchanged by the given day.

From Figure 4.5(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 4.5(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 4.5(b)). According to these 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.

4.6 What is a good Web-page change model?

Now we study whether we can describe the changes of Web pages by a mathematical model. In particular, we study whether the 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 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 a *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.

More precisely, let us use $X(t)$ to refer to the number of occurrences of a certain event in the interval $(0, t]$. If the event happens *randomly*, *independently*, and with a *fixed average rate* λ (events/unit interval), it is called a Poisson process of *rate* or *frequency* λ. In a Poisson process, the random variable $X(t)$ has the following properties [TK98]:

1. for any time points $t_0 = 0 < t_1 < t_2 < \cdots < t_n$, the process increments (the number of events occurring in a certain interval) $X(t_1) - X(t_0),\ X(t_2) - X(t_1),\ \ldots, X(t_n) - X(t_{n-1})$ are independent random variables;

2. for $s \geq 0$ and $t > 0$, the random variable $X(s + t) - X(s)$ has the Poisson probability

distribution

$$\Pr\{X(s+t) - X(s) = k\} = \frac{(\lambda t)^k}{k!} e^{-\lambda t} \text{ for } k = 0, 1, \ldots$$

3. $X(0) = 0$.

By calculating how many events are expected to occur in a unit interval, we can verify that the parameter λ corresponds to the *rate*, or the *frequency* of the event:

$$E[X(t+1) - X(t)] = \sum_{k=0}^{\infty} k \Pr\{X(t+1) - X(t) = k\} = \lambda e^{-\lambda} \sum_{k=1}^{\infty} \frac{\lambda^{k-1}}{(k-1)!} = \lambda e^{-\lambda} e^{\lambda} = \lambda$$

In summary, an event may occur randomly at any point of time, but the average rate of the event is fixed to λ for a Poisson process.

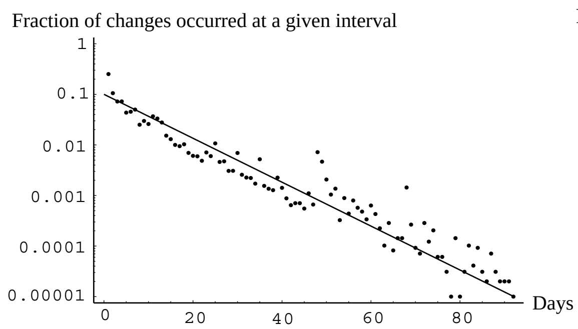
We believe a Poisson process is a good model for the changes of Web pages, because many Web pages have the properties that we just mentioned. For instance, pages in the CNN Web site change at the *average* rate of once a day, but the change of a particular page is quite random, because updates of the page depend 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 lemma [TK98].

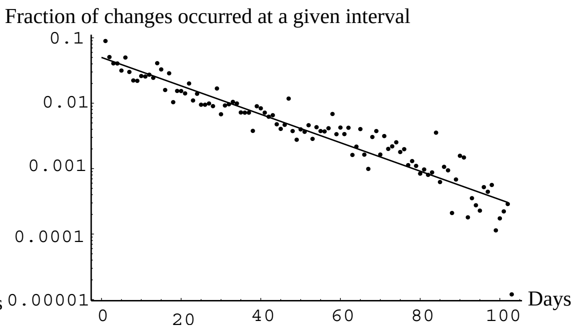
Lemma 4.1 *If T is the time to the occurrence of the next event in a* Poisson process *with rate λ, the probability density function for T is*

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

We can use Lemma 4.1 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



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



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

Figure 4.6: Change intervals of pages

process, this graph should be distributed exponentially. In Figure 4.6, we show some of the graphs plotted this way. Figure 4.6(a) is the graph for the pages with 10 day change interval, and Figure 4.6(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, in the remainder of this dissertation, we will mainly use the Poisson model to compare crawler strategies.

4.7 Related work

There exist a body of literature that studies the evolution of the Web [WM99, WVS$^+$99, DFK99, PP97]. For example, reference [PP97] studies the relationship between the "desirability" of a page and its lifespan to facilitate the user's ability to make sense of large collections of Web pages. Reference [WM99] presents statistics on the Web page changes and the responses from Web servers to improve Web caching policies. However, none of these studies are as extensive as the study in this dissertation, in terms of the scale and the length of the experiments. Also, their focus is different from this dissertation. As we said, reference [WM99] investigates page changes to improve *Web caching policies*, and reference [PP97] studies how page changes are related to *access patterns*.

The study of [BC00] is very similar to the work in this chapter, because it also presents change statistics of Web pages based on the analysis of real Web data. While it does not explicitly propose a Poisson model as the Web change model, it shows some results that we believe is a good indicator of a Poisson model. While some part of the work overlaps with ours, most of the analysis of data is quite different, and thus it presents another interesting set of Web change statistics.

Lawrence et al. [LG98, LG99] tried to measure the number of publicly-indexable pages on the Web. They conducted two experiments (first in 1997 and second in 1999) and reported that the number of public Web pages increased from 320 million in December 1999 to 800 million in February 1999. Since they used slightly different methods for the two experiments, the actual growth rate of the Web may not be accurate. However, their work still presented an interesting estimate on how rapidly the Web grows over time. In this chapter, we mainly focused on the changes of existing Web pages, not on the growth rate of the overall Web.

4.8 Conclusion

In this chapter we tried to understand how Web pages evolve over time, through a comprehensive experiment conducted on 720,000 Web pages. Through our discussion, we identified various issues that a Web crawler itself may encounter when it tries to estimate how often Web pages change. Also, we identified a good model that describes well the changes of Web pages. In the remainder of this dissertation, we will frequently refer to the results in this chapter when we need actual statistics.

Chapter 5

Page Refresh Policy

5.1 Introduction

In order to maintain locally stored pages "fresh," a crawler has to update its Web pages periodically. In this chapter we study how a crawler can effectively synchronize the local copy. As the size of the Web grows, it becomes more difficult to maintain the copy "fresh," making it crucial to synchronize the copy effectively. A recent study shows that it takes up to 6 months for a new page to be indexed by popular Web search engines [LG99]. Also, a lot of users express frustration, when a search engine returns obsolete links, and the users follow the links in vain. According to the same study, up to 14% of the links in search engines are invalid.

While our study is driven by the needs of a Web crawler, local copies of remote data sources are frequently made to improve performance or availability. For instance, a *data warehouse* may copy remote sales and customer tables for local analysis. Similarly, a Web cache stores Web pages locally, to minimize access delay by the users. In many cases, the remote source is updated independently without pushing updates to the client that has a copy, so the client must periodically poll the source to detect changes and refresh its copy. This scenario is illustrated in Figure 5.1.

The effective synchronization of a local copy introduces many interesting challenges. First of all, measuring the freshness of the copy is not trivial. Intuitively, the copy is considered fresh when it is not different from the remote source data. Therefore, we can measure its freshness only when we know the *current status* of the source data. But how can we know the current status of the source data, when it is spread across thousands of

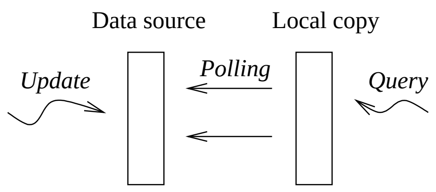


Figure 5.1: Conceptual diagram of the problem

Web sites? Second, we do not know exactly when a particular data item will change, even if it changes at a certain average rate. For instance, the pages in the CNN Web site are updated about once a day, but the update of a particular page depends on how the news related to that page develops over time. Therefore, visiting the page once a day does not guarantee its freshness.

In this chapter, we will formally study how to synchronize the data to maximize its freshness. The main contributions we make are:

- We present a formal framework to study the synchronization problem, and we define the notions of freshness and age of a copy. While our study focuses on the Web environment, we believe our analysis can be applied to other contexts, such as a data warehouse. In a warehouse, *materialized views* are maintained on top of *autonomous* databases, and again, we need to *poll* the underlying database periodically to guarantee some level of freshness.

- We present several synchronization policies that are currently employed, and we compare how effective they are. Our study will show that some policies that may be intuitively appealing might actually perform *worse* than a naive policy.

- We also propose a new synchronization policy which may improve the freshness by orders of magnitude in certain cases.

- We validate our analysis using the experimental data from Chapter 4. The results show that our new policy is indeed better than any of the current policies.

5.2 Framework

To study the synchronization problem, we first need to understand the meaning of "freshness," and we need to know how data changes over time. In this section we present our framework to address these issues. In our discussion, we refer to the Web sites (or the data sources) that we monitor as the *real-world database* and their local copies as the *local database*, when we need to distinguish them. Similarly, we refer to individual Web pages (or individual data items) as the *real-world elements* and as the *local elements*.

In Section 5.2.1, we start our discussion with the definition of two freshness metrics, *freshness* and *age*. Then in Section 5.2.2, we discuss how we model the evolution of individual real-world elements. Finally in Section 5.2.3 we discuss how we model the real-world database as a whole.

5.2.1 Freshness and age

Intuitively, we consider a database "fresher" when the database has more up-to-date elements. For instance, when database A has 10 up-to-date elements out of 20 elements, and when database B has 15 up-to-date elements, we consider B to be fresher than A. Also, we have a notion of "age:" Even if all elements are obsolete, we consider database A "more current" than B, if A was synchronized 1 day ago, and B was synchronized 1 year ago. Based on this intuitive notion, we define *freshness* and *age* as follows:

1. **Freshness:** Let $S = \{e_1, \ldots, e_N\}$ be the local database with N elements. Ideally, all N elements will be maintained up-to-date, but in practice, only $M (< N)$ elements will be up-to-date at a specific time. (By up-to-date we mean that their values equal those of their real-world counterparts.) We define the *freshness* of S at time t as $F(S;t) = M/N$. Clearly, the *freshness* is the fraction of the local database that is up-to-date. For instance, $F(S;t)$ will be one if all local elements are up-to-date, and $F(S;t)$ will be zero if all local elements are out-of-date. For mathematical convenience, we reformulate the above definition as follows:

   Definition 5.1 The *freshness* of a local element e_i at time t is

   $$F(e_i;t) = \begin{cases} 1 & \text{if } e_i \text{ is up-to-date at time } t \\ 0 & \text{otherwise.} \end{cases}$$

   Then, the *freshness* of the local database S at time t is

   $$F(S;t) = \frac{1}{N}\sum_{i=1}^{N} F(e_i;t). \qquad \square$$

Note that freshness is hard to measure exactly in practice since we need to "instantaneously" compare the real-world data to the local copy. But as we will see, it is possible to estimate freshness given some information about how the real-world data changes.

2. **Age:** To capture "how old" the database is, we define the metric *age* as follows:

   Definition 5.2 The *age* of the local element e_i at time t is

$$A(e_i;t) = \begin{cases} 0 & \text{if } e_i \text{ is up-to-date at time } t \\ t - \text{modification time of } e_i & \text{otherwise.} \end{cases}$$

   Then the *age* of the local database S is

$$A(S;t) = \frac{1}{N}\sum_{i=1}^{N} A(e_i;t). \qquad \square$$

   The *age* of S tells us the average "age" of the local database. For instance, if all real-world elements changed one day ago and we have not synchronized them since, $A(S;t)$ is one day.

In Figure 5.2, we show the evolution of $F(e_i;t)$ and $A(e_i;t)$ of an element e_i. In this graph, the horizontal axis represents time, and the vertical axis shows the value of $F(e_i;t)$ and $A(e_i;t)$. We assume that the real-world element changes at the dotted lines and the local element is synchronized at the dashed lines. The *freshness* drops to zero when the real-world element changes, and the *age* increases linearly from that point on. When the local element is synchronized to the real-world element, its *freshness* recovers to one, and its *age* drops to zero.

Obviously, the freshness (and age) of the local database may change over time. For instance, the freshness might be 0.3 at one point of time, and it might be 0.6 at another point of time. To compare different synchronization methods, it is important to have a metric that fairly considers freshness over a period of time, not just at one instant. In this thesis we use the freshness *averaged over time* as this metric.

Definition 5.3 We define the time average of freshness of element e_i, $\bar{F}(e_i)$, and the time

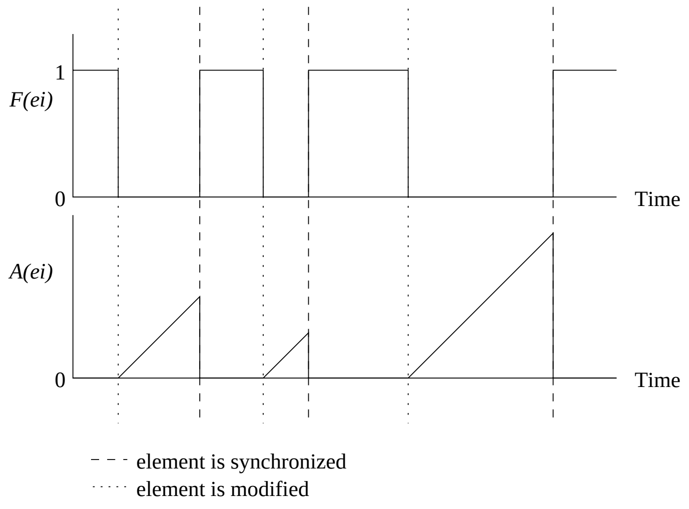


Figure 5.2: An example of the time evolution of $F(e_i;t)$ and $A(e_i;t)$

average of freshness of database S, $\bar{F}(S)$, as

$$\bar{F}(e_i) = \lim_{t\to\infty} \frac{1}{t}\int_0^t F(e_i;t)dt \qquad \bar{F}(S) = \lim_{t\to\infty} \frac{1}{t}\int_0^t F(S;t)dt.$$

The time average of age can be defined similarly. □

From the definition, we can prove that $\bar{F}(S)$ is the average of $\bar{F}(e_i)$.

Theorem 5.1 $\bar{F}(S) = \frac{1}{N}\sum_{i=1}^{N}\bar{F}(e_i) \qquad \bar{A}(S) = \frac{1}{N}\sum_{i=1}^{N}\bar{A}(e_i)$ □

Proof

$$\begin{aligned}
\bar{F}(S) &= \lim_{t\to\infty} \frac{1}{t}\int_0^t F(S;t)dt && \text{(definition of } \bar{F}(S))\\
&= \lim_{t\to\infty} \frac{1}{t}\int_0^t \left(\frac{1}{N}\sum_{i=1}^{N} F(e_i;t)\right) dt && \text{(definition of } F(S;t))\\
&= \frac{1}{N}\sum_{i=1}^{N} \lim_{t\to\infty} \frac{1}{t}\int_0^t F(e_i;t)dt\\
&= \frac{1}{N}\sum_{i=1}^{N} \bar{F}(e_i) && \text{(definition of } \bar{F}(e_i))
\end{aligned}$$

The proof for age is similar. ■

5.2.2 Poisson process and probabilistic evolution of an element

To study how effective different synchronization methods are, we need to know how the real-world elements change. In this thesis, we assume that each element e_i is modified by a Poisson process with change rate λ_i, based on the result in Chapter 4. That is, each element e_i changes at its own average rate λ_i, and this rate may differ from element to element. For example, one element may change once a day, and another element may change once a year.

Under the Poisson process model, we can analyze the freshness and the age of the element e_i over time. More precisely, let us compute the *expected value* of the *freshness* and the *age* of e_i at time t. For the analysis, we assume that we synchronize e_i at $t = 0$ and at $t = I$. Since the time to the next event follows an exponential distribution under a Poisson process (Lemma 4.1 on page 67), we can obtain the probability that e_i changes in the interval $(0, t]$ by the following integration:

$$\Pr\{T \le t\} = \int_0^t f_T(t)dt = \int_0^t \lambda_i e^{-\lambda_i t} dt = 1 - e^{-\lambda_i t}$$

Because e_i is not synchronized in the interval $(0, I)$, the local element e_i may get out-of-date with probability $\Pr\{T \le t\} = 1 - e^{-\lambda_i t}$ at time $t \in (0, I)$. Hence, the *expected freshness* is

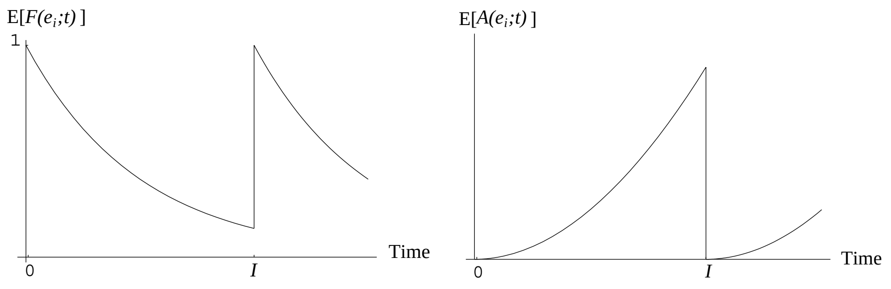
$$\mathrm{E}[F(e_i;t)] = 0 \cdot (1 - e^{-\lambda_i t}) + 1 \cdot e^{-\lambda_i t} = e^{-\lambda_i t} \quad \text{for } t \in (0, I).$$

Note that the expected freshness is 1 at time $t = 0$ and that the expected freshness approaches 0 as time passes.

We can obtain the *expected value* of *age* of e_i similarly. If e_i is modified at time $s \in (0, I)$, the age of e_i at time $t \in (s, I)$ is $(t - s)$. From Lemma 4.1, e_i changes at time s with probability $\lambda_i e^{-\lambda_i s}$, so the expected age at time $t \in (0, I)$ is

$$\mathrm{E}[A(e_i;t)] = \int_0^t (t - s)(\lambda_i e^{-\lambda_i s}) ds = t\left(1 - \frac{1 - e^{-\lambda_i t}}{\lambda_i t}\right)$$

Note that $\mathrm{E}[A(e_i;t)] \to 0$ as $t \to 0$ and that $\mathrm{E}[A(e_i;t)] \approx t$ as $t \to \infty$; the expected age is 0 at time 0 and the expected age is approximately the same as the elapsed time when t is large. In Figure 5.3, we show the graphs of $\mathrm{E}[F(e_i;t)]$ and $\mathrm{E}[A(e_i;t)]$. Note that when we resynchronize e_i at $t = I$, $\mathrm{E}[F(e_i;t)]$ recovers to one and $\mathrm{E}[A(e_i;t)]$ goes to zero.



(a) E[$F(e_i;t)$] graph over time (b) E[$A(e_i;t)$] graph over time

Figure 5.3: Time evolution of E[$F(e_i;t)$] and E[$A(e_i;t)$]

5.2.3 Evolution model of database

In the previous subsection we modeled the evolution of an element. Now we discuss how we model the database as a whole. Depending on how its elements change over time, we can model the real-world database by one of the following:

- **Uniform change-frequency model:** In this model, we assume that all real-world elements change at the *same* frequency λ. This is a simple model that could be useful when:
  - we do not know how often the *individual* element changes over time. We only know how often the entire database changes *on average*, so we may assume that all elements change at the same *average* rate λ.
  - the elements change at *slightly* different frequencies. In this case, this model will work as a good approximation.

- **Non-uniform change-frequency model:** In this model, we assume that the elements change at *different* rates. We use λ_i to refer to the change frequency of the element e_i. When the λ_i's vary, we can plot the histogram of λ_i's as we show in Figure 5.4. In the figure, the horizontal axis shows the range of change frequencies (e.g., $9.5 < \lambda_i \leq 10.5$) and the vertical axis shows the fraction of elements that change at the given frequency range. We can approximate the discrete histogram by a continuous distribution function $g(\lambda)$, when the database consists of many elements. We will adopt the continuous distribution model whenever convenient.

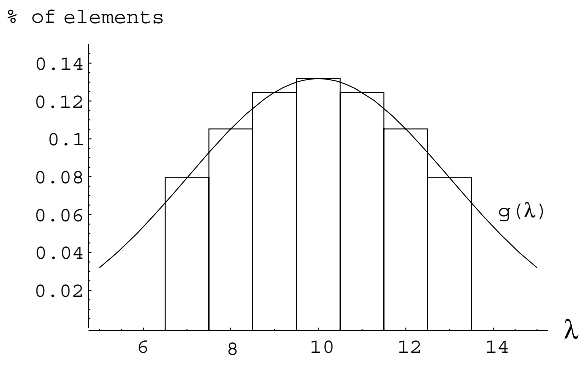


Figure 5.4: Histogram of the change frequencies

symbol	meaning
(a) $\bar{F}(S)$, $\bar{F}(e_i)$	Freshness of database S (and element e_i) averaged over time
(b) $\bar{A}(S)$, $\bar{A}(e_i)$	Age of database S (and element e_i) averaged over time
(c) $\bar{F}(\lambda_i, f_i)$, $\bar{A}(\lambda_i, f_i)$	Freshness (and age) of element e_i averaged over time, when the element changes at the rate λ_i and is synchronized at the frequency f_i
(i) λ_i	Change frequency of element e_i
(j) f_i $(= 1/I_i)$	Synchronization frequency of element e_i
(k) λ	Average change frequency of database elements
(l) f $(= 1/I)$	Average synchronization frequency of database elements

Table 5.1: The symbols that are used throughout this chapter and their meanings

For the reader's convenience, we summarize our notation in Table 5.1. As we continue our discussion, we will explain some of the symbols that have not been introduced yet.

5.3 Synchronization policies

So far we discussed how the real-world database changes over time. In this section we study how the local copy can be refreshed. There are several dimensions to this synchronization process:

1. **Synchronization frequency:** We first need to decide *how frequently* we synchronize the local database. Obviously, as we synchronize the database more often, we can maintain the local database fresher. In our analysis, we assume that we synchronize N elements per I time-units. By varying the value of I, we can adjust how often we synchronize the database.

2. **Resource allocation:** Even after we decide how many elements we synchronize per unit interval, we still need to decide how frequently we synchronize *each individual* element. We illustrate this issue by an example.

   Example 5.1 A database consists of three elements, e_1, e_2 and e_3. It is known that the elements change at the rates $\lambda_1 = 4$, $\lambda_2 = 3$, and $\lambda_3 = 2$ (times/day). We have decided to synchronize the database at the *total* rate of 9 elements/day. In deciding how frequently we synchronize each element, we consider the following options:

   - Synchronize all elements uniformly at the same rate. That is, synchronize e_1, e_2 and e_3 at the same rate of 3 (times/day).
   - Synchronize an element proportionally more often when it changes more often. In other words, synchronize the elements at the rates of $f_1 = 4$, $f_2 = 3$, $f_3 = 2$ (times/day) for e_1, e_2 and e_3, respectively. □

   Based on how fixed synchronization resources are allocated to individual elements, we can classify synchronization policies as follows. We study these policies later in Section 5.5.

   (a) **Uniform-allocation policy:** We synchronize all elements at the same rate, regardless of how often they change. That is, each element e_i is synchronized at the fixed frequency f. In Example 5.1, the first option corresponds to this policy.

   (b) **Non-uniform-allocation policy:** We synchronize elements at different rates. In particular, with a *proportional-allocation policy* we synchronize element e_i with a frequency f_i that is proportional to its change frequency λ_i. Thus, the frequency ratio λ_i/f_i, is the same for any i under the proportional-allocation policy. In Example 5.1, the second option corresponds to this policy.

3. **Synchronization order:** Now we need to decide in *what order* we synchronize the elements in the database.

   Example 5.2 We maintain a local database of 10,000 Web pages from site A. In order to maintain the local copy up-to-date, we continuously update our local database by revisiting the pages in the site. In performing the update, we may adopt one of the following options:

- We maintain an explicit list of all URLs in the site, and we visit the URLs repeatedly in the same order. Notice that if we update our local database at a fixed rate, say 10,000 pages/day, then we synchronize a page, say p_1, at the fixed interval of one day.
- We only maintain the URL of the root page of the site, and whenever we crawl the site, we start from the root page, following links. Since the link structure (and the order) at a particular crawl determines the page visit order, the synchronization order may change from one crawl to the next. Notice that under this policy, we synchronize a page, say p_1, at variable intervals. For instance, if we visit p_1 at the end of one crawl and at the beginning of the next crawl, the interval is close to zero, while in the opposite case it is close to two days.
- Instead of actively synchronizing pages, we synchronize pages on demand, as they are *requested* by a user. Since we do not know which page the user will request next, the synchronization order may appear random. Under this policy, the synchronization interval of p_1 is not bound by any value. It may range from zero to infinity. □

We can summarize the above options as follows. In Section 5.4 we will compare how effective these synchronization order policies are.

(a) **Fixed order:** We synchronize all elements in the database in the *same* order repeatedly. Therefore, a particular element is synchronized at a *fixed interval* under this policy. This policy corresponds to the first option of the above example.

(b) **Random order:** We synchronize all elements repeatedly, but the synchronization order may be different in each iteration. This policy corresponds to the second option in the example.

(c) **Purely random:** At each synchronization point, we select a random element from the database and synchronize it. Therefore, an element is synchronized at intervals of arbitrary length. This policy corresponds to the last option in the example.

4. **Synchronization points:** In some cases, we may need to synchronize the database only in a limited time window. For instance, if a Web site is heavily accessed during daytime, it might be desirable to crawl the site only in the night, when it is less

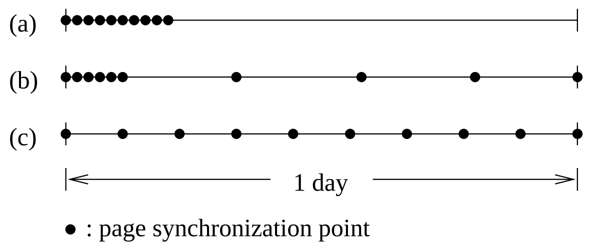


Figure 5.5: Several options for the synchronization points

frequently visited. We illustrate several options for dealing with this constraint by an example.

Example 5.3 We maintain a local database of 10 pages from site A. The site is heavily accessed during daytime. We consider several synchronization policies, including the following:

- **Figure 5.5(a):** We synchronize all 10 pages in the beginning of a day, say midnight.
- **Figure 5.5(b):** We synchronize most pages in the beginning of a day, but we still synchronize some pages during the rest of the day.
- **Figure 5.5(c):** We synchronize 10 pages uniformly over a day. □

In this chapter we assume that we synchronize a database uniformly over time. We believe this assumption is valid especially for the Web environment. Because the Web sites are located in many different time zones, it is not easy to identify which time zone a particular Web site resides in. Also, the access pattern to a Web site varies widely. For example, some Web sites are heavily accessed during daytime, while others are accessed mostly in the evening, when the users are at home. Since crawlers cannot guess the best time to visit each site, they typically visit sites at a uniform rate that is convenient to the crawler.

5.4 Synchronization-order policies

Clearly, we can increase the database freshness by synchronizing more often. But exactly how often should we synchronize, for the freshness to be, say, 0.8? Conversely, how much freshness do we get if we synchronize 100 elements per second? In this section, we will

```
Algorithm 5.4.1 Fixed-order synchronization
Input: ElemList = $\{e_1, e_2, \ldots, e_N\}$
Procedure
  [1] While (TRUE)
  [2]     SyncQueue := ElemList
  [3]     While (not Empty(SyncQueue))
  [4]         e := Dequeue(SyncQueue)
  [5]         Synchronize(e)
```

Figure 5.6: Algorithm of fixed-order synchronization policy

address these questions by analyzing synchronization-order policies. Through the analysis, we will also learn which synchronization-order policy is the best in terms of freshness and age.

In this section we assume that all real-world elements are modified at the same average rate λ. That is, we adopt the *uniform change-frequency model* (Section 5.2.3). When the elements change at the same rate, it does not make sense to synchronize the elements at different rates, so we also assume *uniform-allocation policy* (Item 2a in Section 5.3). These assumptions significantly simplify our analysis, while giving us solid understanding on the issues that we address. Based on these assumptions, we analyze different synchronization-order policies in the subsequent subsections. A reader who is not interested in mathematical details may skip to Section 5.4.4.

5.4.1 Fixed-order policy

Under the fixed-order policy, we synchronize the local elements in the *same order* repeatedly. We describe the fixed-order policy more formally in Figure 5.6. Here, `ElemList` records *ordered* list of *all* local elements, and `SyncQueue` records the elements *to be synchronized* in each iteration. In step [3] through [5], we synchronize all elements once, and we repeat this loop forever. Note that we synchronize the elements in the same order in every iteration, because the order in `SyncQueue` is always the same,

Now we compute the freshness of the database S. (Where convenient, we will refer to the time-average of freshness simply as *freshness*, if it does not cause any confusion.) Since we can compute the freshness of S from freshness of its elements (Theorem 5.1), we first study how a random element e_i evolves over time.

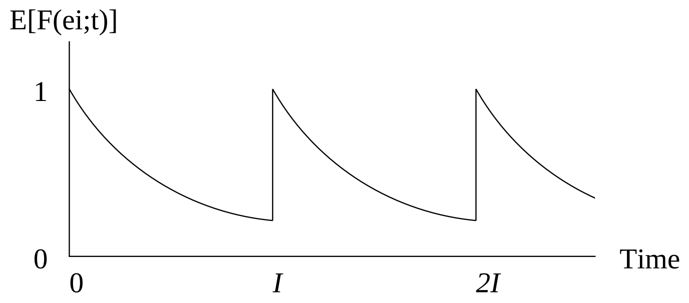


Figure 5.7: Time evolution of $\mathrm{E}[F(e_i;t)]$ for fixed-order policy

Assuming that it takes I seconds to synchronize all elements in S, the expected freshness of e_i will evolve as in Figure 5.7. In the graph, we assumed that we synchronize e_i initially at $t = 0$, without losing generality. Note that $\mathrm{E}[F(e_i;t)]$ recovers to 1 every I seconds, when we synchronize it. Intuitively, e_i goes through exactly the same process every I seconds, so we can expect that we can learn anything about e_i by studying how e_i evolves in the interval $(0, I)$. In particular, we suspect that the freshness of e_i averaged over time ($\lim_{t\to\infty} \frac{1}{t}\int_0^t F(e_i;t)dt$) should be equal to the expected freshness of e_i averaged over the interval $(0, I)$ (i.e., $\frac{1}{I}\int_0^I \mathrm{E}[F(e_i;t)]dt$). The following theorem shows that our intuition is correct.

Theorem 5.2 *When the element e_i is synchronized at the fixed interval of I seconds, the time average of the freshness of e_i is the same as the time average of $\mathrm{E}[F(e_i;t)]$ over the interval $(0, I)$.*

$$\bar{F}(e_i) = \frac{1}{I}\int_0^I \mathrm{E}[F(e_i;t)]dt \qquad \square$$

Proof

$$\begin{aligned}
\bar{F}(e_i) &= \lim_{t\to\infty} \frac{\int_0^t F(e_i;t)dt}{t} \\
&= \lim_{n\to\infty} \frac{\sum_{j=0}^{n-1}\int_{jI}^{(j+1)I} F(e_i;t)dt}{\sum_{j=0}^{n-1} I} \\
&= \lim_{n\to\infty} \frac{\sum_{j=0}^{n-1}\int_0^I F(e_i;t+jI)dt}{nI} \\
&= \frac{1}{I}\int_0^I \left[\lim_{n\to\infty}\frac{1}{n}\sum_{j=0}^{n-1} F(e_i;t+jI)\right] dt \qquad (5.1)
\end{aligned}$$

Because we synchronize e_i every I seconds from $t = 0$, $F(e_i;t+jI)$ is the freshness of e_i

at t *seconds after each synchronization.* Therefore, $\frac{1}{n}\sum_{j=0}^{n-1} F(e_i; t + jI)$, the average of freshness at t seconds after synchronization, will converge to its expected value, $\mathrm{E}[F(e_i;t)]$, as $n \to \infty$. That is,

$$\lim_{n\to\infty} \frac{1}{n}\sum_{j=0}^{n-1} F(e_i; t + jI) = \mathrm{E}[F(e_i;t)].$$

Then,

$$\frac{1}{I}\int_0^I \left[\lim_{n\to\infty} \frac{1}{n}\sum_{j=0}^{n-1} F(e_i; t + jI)\right] dt = \frac{1}{I}\int_0^I \mathrm{E}[F(e_i;t)]dt. \tag{5.2}$$

From Equation 5.1 and 5.2, $F(e_i) = \frac{1}{I}\int_0^I \mathrm{E}[F(e_i;t)]dt.$ ■

Based on Theorem 5.2, we can compute the freshness of e_i.

$$\bar{F}(e_i) = \frac{1}{I}\int_0^I \mathrm{E}[F(e_i;t)]dt = \frac{1}{I}\int_0^I e^{-\lambda t}dt = \frac{1 - e^{-\lambda I}}{\lambda I} = \frac{1 - e^{-\lambda/f}}{\lambda/f}$$

We assumed that all elements change at the same frequency λ and that they are synchronized at the same interval I, so the above equation holds for any element e_i. Therefore, the freshness of database S is

$$\bar{F}(S) = \frac{1}{N}\sum_{i=1}^{N} \bar{F}(e_i) = \frac{1 - e^{-\lambda/f}}{\lambda/f}.$$

We can analyze the age of S similarly, and we get

$$\bar{A}(S) = I(\frac{1}{2} - \frac{1}{\lambda/f} + \frac{1 - e^{-\lambda/f}}{(\lambda/f)^2}).$$

5.4.2 Random-order policy

Under the random-order policy, the synchronization order of elements might be different from one crawl to the next. Figure 5.8 describes the random-order policy more precisely. Note that we randomize the order of elements before every iteration by applying random permutation (step [2]).

Obviously, the random-order policy is more complex to analyze than the fixed-order policy. Since we may synchronize e_i at any point during interval I, the synchronization

```
**Algorithm 5.4.2 Random-order synchronization**
Input: ElemList = $\{e_1, e_2, \ldots, e_N\}$
Procedure
  [1] While (TRUE)
  [2]     SyncQueue := RandomPermutation(ElemList)
  [3]     While (not Empty(SyncQueue))
  [4]         e := Dequeue(SyncQueue)
  [5]         Synchronize(e)
```

Figure 5.8: Algorithm of random-order synchronization policy

interval of e_i is not fixed any more. In one extreme case, it may be almost $2I$, when e_i is synchronized at the beginning of the first iteration and at the end of the second iteration. In the opposite case, it may be close to 0, when e_i is synchronized at the end of the first iteration and at the beginning of second iteration. Therefore, the synchronization interval of e_i, W, is not a fixed number any more, but follows a certain distribution $f_W(t)$. Therefore the equation of Theorem 5.2 should be modified accordingly:

$$\bar{F}(e_i) = \lim_{t\to\infty} \frac{1}{t} \int_0^t F(e_i;t)dt = \frac{\int_0^{2I} f_W(s) \left(\int_0^s \mathrm{E}[F(e_i;t)]dt\right) ds}{\int_0^{2I} f_W(s)\ s\ ds} \tag{5.3}$$

In Theorem 5.2, we simply divided $\int_0^I \mathrm{E}[F(e_i;t)]dt$ by I, because the synchronization interval was fixed to I. But now we take the average of $\int_0^s \mathrm{E}[F(e_i;t)]dt$ and s weighted by the relative frequencies of the interval s ($f_W(s)$). To perform the above integration, we need to derive the closed form of $f_W(t)$.

Lemma 5.1 *Let T_1 (T_2) be the time when element e_i is synchronized in the first (second) iteration under the random-order policy. Then the p.d.f. of $W = T_1 - T_2$, the synchronization interval of e_i, is*

$$f_W(t) = \begin{cases} \frac{t}{I^2} & 0 \le t \le I \\ \frac{2I-t}{I^2} & I \le t \le 2I \\ 0 & \text{otherwise.} \end{cases} \qquad \Box$$

Proof The p.d.f.'s of T_1 and T_2 are

$$f_{T_1}(t) = \begin{cases} \frac{1}{I} & 0 \le t \le I \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad f_{T_2}(t) = \begin{cases} \frac{1}{I} & I \le t \le 2I \\ 0 & \text{otherwise.} \end{cases}$$

Then

$$\begin{aligned} f_W(t) &= f(T_2 - T_1 = t) \\ &= \int_0^I f(T_1 = s) f(T_2 - T_1 = t | T_1 = s)\, ds \\ &= \int_0^I f(T_1 = s) f(T_2 = s + t)\, ds \\ &= \frac{1}{I} \int_0^I f(T_2 = s + t)\, ds. \end{aligned}$$

■

When $t < 0$ or $t > 2I$, $f(T_2 = s + t) = 0$ for any $s \in (0, I)$. Therefore,

$$f_W(t) = \frac{1}{I} \int_0^I f(T_2 = s + t)\, ds = 0.$$

When $0 \le t \le I$, $f(T_2 = s + t) = \frac{1}{I}$ for $s \in (I - t, I)$. Then,

$$f_W(t) = \frac{1}{I} \int_{I-t}^I \frac{1}{I}\, ds = \frac{t}{I^2}.$$

When $I \le t \le 2I$, $f(T_2 = s + t) = \frac{1}{I}$ for $s \in (0, 2I - t)$, and therefore

$$f_W(t) = \frac{1}{I} \int_0^{2I-t} \frac{1}{I}\, ds = \frac{2I - t}{I^2}.$$

Based on the Lemma 5.1 and Equation 5.3, we can compute the freshness of the random-order policy, and the result is

$$\bar{F}(e_i) = \frac{1}{\lambda/f} \left[1 - \left(\frac{1 - e^{-\lambda/f}}{\lambda/f} \right)^2 \right].$$

```
Algorithm 5.4.3 Purely-random synchronization
Input: ElemList = $\{e_1, e_2, \ldots, e_N\}$
Procedure
    [1] While (TRUE)
    [2]     SyncQueue := RandomPermutation(ElemList)
    [3]     e := Dequeue(SyncQueue)
    [4]     Synchronize(e)
```

Figure 5.9: Algorithm of purely-random synchronization policy

Since the above analysis is valid for any element e_i, the freshness of S becomes

$$\bar{F}(S) = \frac{1}{\lambda/f}\left[1 - \left(\frac{1 - e^{-\lambda/f}}{\lambda/f}\right)^2\right].$$

We can compute $\bar{A}(S)$ similarly.

$$\bar{A}(S) = I\left[\frac{1}{3} + \left(\frac{1}{2} - \frac{1}{\lambda/f}\right)^2 - \left(\frac{1 - e^{-\lambda/f}}{(\lambda/f)^2}\right)^2\right]$$

5.4.3 Purely-random policy

Whenever we synchronize an element, we pick an arbitrarily random element under the purely-random policy. Figure 5.9 describes the policy more formally. Note that we pick a completely random element at every synchronization point, since we rebuild SyncQueue before every synchronization.

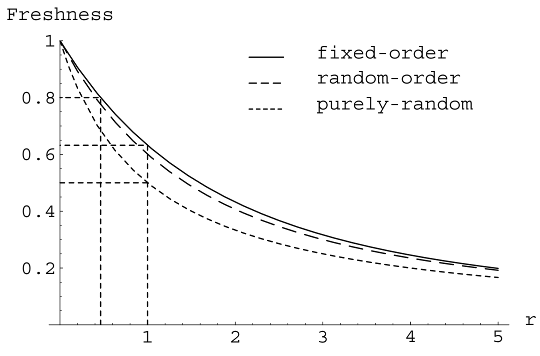
The analysis of purely-random policy is similar to that of random-order policy. Here again, the time between synchronizations of e_i, W, is a random variable with a probability density function $f_W(t)$, and the freshness of e_i becomes

$$\bar{F}(e_i) = \lim_{t\to\infty}\frac{1}{t}\int_0^t F(e_i;t)dt = \frac{\int_0^\infty f_W(s)\left(\int_0^s \mathrm{E}[F(e_i;t)]dt\right)ds}{\int_0^\infty f_W(s)\; s\; ds}.$$

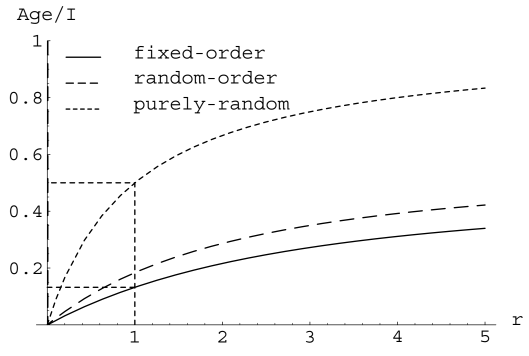
Note that the outer integral is over $(0, \infty)$, since the synchronization interval of e_i may get

policy	*Freshness* $\bar{F}(S)$	*Age* $\bar{A}(S)$
Fixed-order	$\frac{1-e^{-r}}{r}$	$I(\frac{1}{2} - \frac{1}{r} + \frac{1-e^{-r}}{r^2})$
Random-order	$\frac{1}{r}(1 - (\frac{1-e^{-r}}{r})^2)$	$I(\frac{1}{3} + (\frac{1}{2} - \frac{1}{r})^2 - (\frac{1-e^{-r}}{r^2})^2)$
Purely-random	$\frac{1}{1+r}$	$I(\frac{r}{1+r})$

Table 5.2: Freshness and age formula for various synchronization-order policies



(a) Freshness graph over $r = \lambda/f$



(b) Age graph over $r = \lambda/f$

Figure 5.10: Comparison of freshness and age of various synchronization policies

arbitrarily large. From the raw of rare events [TK98], we can prove that $f_W(t)$ is

$$f_W(t) = \begin{cases} fe^{-ft} & t \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

and we get

$$\bar{F}(S) = \frac{1}{1 + \lambda/f} \qquad \bar{A}(S) = I\left(\frac{\lambda/f}{1 + \lambda/f}\right).$$

5.4.4 Comparison of synchronization-order policies

In Table 5.2, we summarize the results in the preceding subsections. In the table, we use r to represent the frequency ratio λ/f, where λ is the frequency at which a real-world element changes and $f(= 1/I)$ is the frequency at which a local element is synchronized. When $r < 1$, we synchronize the elements more often than they change, and when $r > 1$, the elements change more often than we synchronize them.

To help readers interpret the formulas, we show the freshness and the age graphs in Figure 5.10. In the figure, the horizontal axis is the frequency ratio r, and the vertical axis shows the freshness and the age of the local database. Notice that as we synchronize the elements more often than they change ($\lambda \ll f$, thus $r = \lambda/f \to 0$), the freshness approaches 1 and the age approaches 0. Also, when the elements change more frequently than we synchronize them ($r = \lambda/f \to \infty$), the freshness becomes 0, and the age increases. Finally, notice that the freshness is not equal to 1, even if we synchronize the elements as often as they change ($r = 1$). This fact has two reasons. First, an element changes at random points of time, even if it changes at a fixed *average* rate. Therefore, the element may not change between some synchronizations, and it may change more than once between other synchronizations. For this reason, it cannot be always up-to-date. Second, some delay may exist between the change of an element and its synchronization, so some elements may be "temporarily obsolete," decreasing the freshness of the database.

The graphs of Figure 5.10 have many practical implications. For instance, we can answer all of the following questions by looking at the graphs.

- **How can we measure how fresh a local database is?** By measuring how frequently real-world elements change, we can estimate how fresh a local database is. For instance, when the real-world elements change once a day, and when we synchronize the local elements also once a day ($\lambda = f$ or $r = 1$), the freshness of the local database is $(e - 1)/e \approx 0.63$, under the fixed-order policy.

  Note that we derived the equations in Table 5.2 assuming that the real-world elements change at the *same* rate λ. Therefore, the equations may not be true when the real-world elements change at *different* rates. However, we can still interpret λ as the *average* rate at which the whole database changes, and we can use the formulas as approximations. Later in Section 5.5, we derive an exact formula for when the elements change at different rates.

- **How can we guarantee certain freshness of a local database?** From the graph, we can find how frequently we should synchronize local elements in order to achieve certain freshness. For instance, if we want at least 0.8 freshness, the frequency ratio r should be less than 0.46 (fixed-order policy). That is, we should synchronize the local elements at least $1/0.46 \approx 2$ times as frequently as the real-world elements change.

- **Which synchronization-order policy is the best?** The fixed-order policy performs best by both metrics. For instance, when we synchronize the elements as often as they change ($r = 1$), the freshness of the fixed-order policy is $(e-1)/e \approx 0.63$, which is 30% higher than that of the purely-random policy. The difference is more dramatic for age. When $r = 1$, the age of the fixed-order policy is only one fourth of the random-order policy. In general, as the variability in the time between visits increases, the policy gets less effective.

5.5 Resource-allocation policies

In the previous section, we addressed various questions assuming that all elements in the database change at the same rate. But what can we do if the elements change at *different* rates and we know how often each element changes? Is it better to synchronize an element more often when it changes more often? In this section we address this question by analyzing different resource-allocation policies (Item 2 in Section 5.3). For the analysis, we model the real-world database by the *non-uniform* change-frequency model (Section 5.2.3), and we assume the *fixed-order* policy for the synchronization-order policy (Item 3 in Section 5.3), because the fixed-order policy is the best synchronization-order policy. In other words, we assume that the element e_i changes at the frequency λ_i (λ_i's may be different from element to element), and we synchronize e_i at the *fixed interval* $I_i(= 1/f_i$, where f_i is synchronization frequency of $e_i)$. Remember that we synchronize N elements in $I(= 1/f)$ time units. Therefore, the average synchronization frequency ($\frac{1}{N}\sum_{i=1}^{N} f_i$) should be equal to f.

In Sections 5.5.1 and 5.5.2, we start our discussion by comparing the uniform allocation policy with the proportional allocation policy. Surprisingly, the uniform policy turns out to be *always* more effective than the proportional policy. Then in Section 5.5.3 we try to understand why this happens by studying a simple example. Finally in Section 5.5.4 we study how we should allocate resources to the elements to achieve the optimal freshness or age. A reader who is not interested in mathematical details may skip to Section 5.5.3.

5.5.1 Uniform and proportional allocation policy

In this subsection, we first assume that the change frequencies of real-world elements follow the *gamma distribution*, and compare how effective the *proportional* and the *uniform* policies are. In Section 5.5.2 we also prove that the conclusion of this section is valid for *any* distribution.

The gamma distribution is often used to model a random variable whose domain is non-negative numbers. Also, the distribution is known to cover a wide array of distributions. For instance, the exponential and the chi-square distributions are special instances of the gamma distribution, and the gamma distribution is close to the normal distribution when the variance is small. This mathematical property and versatility makes the gamma distribution a desirable one for describing the distribution of change frequencies.

To compute the freshness, we assume that we synchronize element e_i at the *fixed* frequency f_i (Fixed-order policy, Item 3a of Section 5.3). In Section 5.4.1, we showed that the freshness of e_i in this case is

$$\bar{F}(\lambda_i, f_i) = \frac{1 - e^{-\lambda_i/f_i}}{\lambda_i/f_i} \tag{5.4}$$

and the age of e_i is

$$\bar{A}(\lambda_i, f_i) = \frac{1}{f_i}\left(\frac{1}{2} - \frac{1}{\lambda_i/f_i} + \frac{1 - e^{-\lambda_i/f_i}}{(\lambda_i/f_i)^2}\right). \tag{5.5}$$

The gamma distribution $g(x)$ with parameters $\alpha > 0$ and $\mu > 0$ is

$$g(x) = \frac{\mu}{\Gamma(\alpha)}(\mu x)^{\alpha-1}e^{-\mu x} \qquad \text{for} \quad x > 0 \tag{5.6}$$

and the mean and the variance of the distribution are

$$\mathrm{E}[X] = \frac{\alpha}{\mu} \quad \text{and} \quad \mathrm{Var}[X] = \frac{\alpha}{\mu^2}. \tag{5.7}$$

Now let us compute the freshness of S for the uniform allocation policy. By the definition of the uniform allocation policy, $f_i = f$ for any i. Then from Theorem 5.1,

$$\bar{F}(S)_u = \frac{1}{N}\sum_{i=1}^{N}\bar{F}(e_i) = \frac{1}{N}\sum_{i=1}^{N}\bar{F}(\lambda_i, f)$$

where subscript u stands for the uniform allocation policy. When N is large, we can approximate the above average by the weighted integral

$$\bar{F}(S)_u = \frac{1}{N}\sum_{i=1}^{N}\bar{F}(\lambda_i, f) = \int_0^\infty g(\lambda)\bar{F}(\lambda, f)d\lambda \tag{5.8}$$

where $g(\lambda)$ is the gamma distribution. By substituting $g(\lambda)$ and $\bar{F}(\lambda, f)$ using Equation 5.6 and 5.4, we get

$$\bar{F}(S)_u = \frac{\mu f}{1-\alpha}\left((1+\frac{1}{\mu f})^{1-\alpha} - 1\right).$$

It is more intuitive to deal with the *mean* and the *variance* of the distribution than α and μ, so we reformulate the above formula with

$$\lambda = (\text{mean}) = \frac{\alpha}{\mu} \quad \text{and} \quad \delta^2 = \frac{(\text{variance})}{(\text{mean})^2} = \frac{\alpha/\mu^2}{(\alpha/\mu)^2} \qquad \text{(From Equation 5.7).}$$

Then $\bar{F}(S)_u$ becomes

$$\bar{F}(S)_u = \frac{1-(1+r\delta^2)^{1-\frac{1}{\delta^2}}}{r(1-\delta^2)} \quad (r = \lambda/f)$$

.

We can compute the age of the database, $\bar{A}(S)_u$, similarly.

$$\begin{aligned}\bar{A}(S)_u &= \frac{1}{N}\sum_{i=1}^{N}\bar{A}(\lambda_i, f) = \int_0^\infty g(\lambda)\bar{A}(\lambda, f)d\lambda \\ &= \frac{1}{f(1-\delta^2)}\left[\frac{1-\delta^2}{2} - \frac{1}{r} + \frac{1-(1+r\delta^2)^{2-\frac{1}{\delta^2}}}{r^2(1-2\delta^2)}\right]\end{aligned}$$

Now let us compute the freshness of the proportional allocation policy. By the definition of the proportional policy, $\lambda_i/f_i = \lambda/f$ for any i. Then from Equation 5.4 and 5.5, we can derive

$$\bar{F}(\lambda_i, f_i) = \frac{1-e^{-\lambda/f}}{\lambda/f} \qquad \bar{A}(\lambda_i, f_i) = \frac{1}{\lambda_i}\left[\frac{\lambda}{f}\left(\frac{1}{2} - \frac{1}{\lambda/f} + \frac{1-e^{-\lambda/f}}{(\lambda/f)^2}\right)\right].$$

Therefore, $\bar{F}(S)_p$ and $\bar{A}(S)_p$ becomes

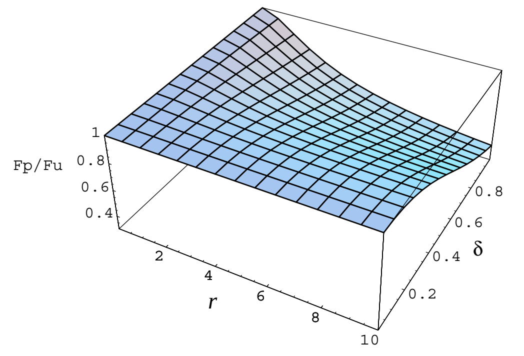
policy	*Freshness* $\bar{F}(S)$	*Age* $\bar{A}(S)$
Uniform allocation	$\frac{1-(1+r\delta^2)^{1-\frac{1}{\delta^2}}}{r(1-\delta^2)}$	$\frac{I}{(1-\delta^2)}[\frac{1-\delta^2}{2} - \frac{1}{r} + \frac{1-(1+r\delta^2)^{2-\frac{1}{\delta^2}}}{r^2(1-2\delta^2)}]$
Proportional allocation	$\frac{1-e^{-r}}{r}$	$\frac{I}{(1-\delta^2)}[\frac{1}{2} - \frac{1}{r} + \frac{1-e^{-r}}{r^2}]$

Table 5.3: Freshness and age formula for various resource-allocation policies

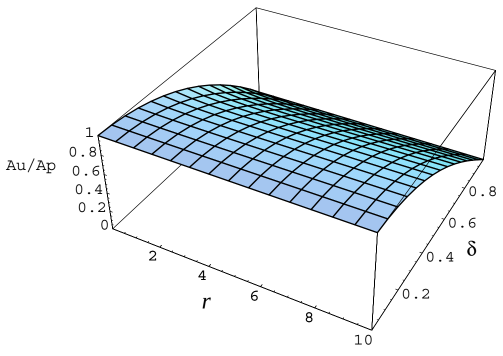
$$
\begin{aligned}
\bar{F}(S)_p &= \frac{1}{N}\sum_{i=1}^{N}\bar{F}(\lambda_i, f_i) = \frac{1-e^{-\lambda/f}}{\lambda/f} \\
\bar{A}(S)_p &= \frac{1}{N}\sum_{i=1}^{N}\bar{A}(\lambda_i, f_i) \\
&= \left[\frac{\lambda}{f}\left(\frac{1}{2} - \frac{1}{\lambda/f} + \frac{1-e^{-\lambda/f}}{(\lambda/f)^2}\right)\right]\left(\frac{1}{N}\sum_{i=1}^{N}\frac{1}{\lambda_i}\right) \\
&= \left[\frac{\lambda}{f}\left(\frac{1}{2} - \frac{1}{\lambda/f} + \frac{1-e^{-\lambda/f}}{(\lambda/f)^2}\right)\right]\int_0^\infty g(\lambda)\,\frac{1}{\lambda}\,d\lambda \\
&= \frac{1}{f(1-\delta^2)}\left[\frac{1}{2} - \frac{1}{\lambda/f} + \frac{1-e^{-\lambda/f}}{(\lambda/f)^2}\right].
\end{aligned}
$$

In Table 5.3 we summarize the results of our analyses. In the table, r represents the frequency ratio λ/f, where λ is the average rate at which elements change (the mean of the gamma distribution), and f is the average rate at which we synchronize them ($1/I$). Also, δ represents the standard deviation of change frequencies (more precisely, $\delta^2 =$(variance)/(mean)^2 of the gamma distribution). Note that the uniform policy is better than the proportional one if $\bar{F}(S)_p < \bar{F}(S)_u$ and $\bar{A}(S)_u < \bar{A}(S)_p$. To compare the two policies, we plot $\bar{F}(S)_p/\bar{F}(S)_u$ and $\bar{A}(S)_u/\bar{A}(S)_p$ graphs in Figure 5.11. When the uniform policy is better, the ratios are below 1 ($\bar{F}(S)_p/\bar{F}(S)_u < 1$ and $\bar{A}(S)_u/\bar{A}(S)_p < 1$), and when the proportional policy is better, the ratios are above 1 ($\bar{F}(S)_p/\bar{F}(S)_u > 1$ and $\bar{A}(S)_u/\bar{A}(S)_p > 1$).

Surprisingly, we can clearly see that the ratios are below 1 for any r and δ values: The uniform policy is always better than the proportional policy! In fact, the uniform policy gets more effective as the elements change at more different frequencies. That is, when the variance of change frequencies is zero ($\delta = 0$), all elements change at the same frequency,



(a) $\bar{F}(S)_p/\bar{F}(S)_u$ graph over r and δ



(b) $\bar{A}(S)_u/\bar{A}(S)_p$ graph over r and δ

Figure 5.11: $\bar{F}(S)_p/\bar{F}(S)_u$ and $\bar{A}(S)_u/\bar{A}(S)_p$ graphs over r and δ

so two policies give the same result ($\bar{F}(S)_p/\bar{F}(S)_u = 1$ and $\bar{A}(S)_u/\bar{A}(S)_p = 1$). But as δ increases (i.e., as the elements change at more different frequencies), $\bar{F}(S)_u$ grows larger than $\bar{F}(S)_p$ ($\bar{F}(S)_p/\bar{F}(S)_u \to 0$) and $\bar{A}(S)_u$ gets smaller than $\bar{A}(S)_p$ ($\bar{A}(S)_u/\bar{A}(S)_p \to 0$). Interestingly, we can observe that the age *ratio* does not change much as r increases, while the freshness *ratio* heavily depends on the r value.

While we showed that the uniform policy is better than the proportional one only for the gamma-distribution assumption, it is in fact a very general conclusion. In the next subsection, we prove that the uniform policy is *always* better than the proportional policy under *any* distribution.

5.5.2 Superiority of the uniform policy over the proportional policy

To prove that the uniform policy is better than the propotional policy under *any* distribution, we first derive the following lemma.

Lemma 5.2 *When $f(x)$ is a convex function,*

$$\frac{1}{n}\sum_{i=1}^{n} f(x_i) \geq f(\frac{1}{n}\sum_{i=1}^{n} x_i) \quad \text{for any } \ {x_i}'\text{s} \ \ (i = 1, 2, \ldots, n).$$

Similarly, when $f(x)$ is concave,

$$\frac{1}{n}\sum_{i=1}^{n} f(x_i) \le f(\frac{1}{n}\sum_{i=1}^{n} x_i) \quad \text{for any } x_i\text{'s } (i = 1, 2, \ldots, n).$$

Proof We prove it by mathematical induction.

1. **Base case:** When $n = 1$, it is obvious that $f(x_1) \ge f(x_1)$ for any x_1.

2. **Induction step:** We assume $\frac{1}{k}\sum_{i=1}^{k} f(x_i) \ge f(\frac{1}{k}\sum_{i=1}^{k} x_i)$ when $k \le n-1$, and we prove it when $k = n$.

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n} f(x_i) &= \frac{1}{n}\left[\frac{n-1}{n-1}\sum_{i=1}^{n-1} f(x_i) + f(x_n)\right] \\
&\ge \frac{1}{n}\left[(n-1)f(\frac{1}{n-1}\sum_{i=1}^{n-1} x_i) + f(x_n)\right] \quad \text{(induction hypothesis)} \\
&= \frac{n-1}{n} f(\frac{1}{n-1}\sum_{i=1}^{n-1} x_i) + \frac{1}{n} f(x_n) \qquad (5.9)
\end{aligned}$$

We assumed $f(x)$ is convex. From the definition of convexity,

$$pf(a) + (1-p)f(b) \ge f(pa + (1-p)b) \quad \text{for any } 0 \le p \le 1. \qquad (5.10)$$

When we set $p = (n-1)/n$, $a = \sum_{i=1}^{n-1} x_i/(n-1)$ and $b = x_n$, Equation 5.10 becomes

$$\frac{n-1}{n} f(\frac{1}{n-1}\sum_{i=1}^{n-1} x_i) + \frac{1}{n} f(x_n) \ge f(\frac{n-1}{n}\frac{1}{n-1}\sum_{i=1}^{n-1} x_i + \frac{1}{n}x_n) = f(\frac{1}{n}\sum_{i=1}^{n} x_i). \qquad (5.11)$$

From Equation 5.9 and 5.11, it is clear that

$$\frac{1}{n}\sum_{i=1}^{n} f(x_i) \ge f(\frac{1}{n}\sum_{i=1}^{n} x_i).$$

The proof for concavity is similar. ■

Now we prove that the freshness of the uniform policy, $\bar{F}(S)_u$, is better than the freshness of the proportional policy, $\bar{F}(S)_p$. The proof is based on the convexity of the function $\bar{F}(\lambda_i, f_i)$ over λ_i. That is, the analytic forms of $\bar{F}(\lambda_i, f_i)$ that we derived for various synchronization-order policies were all convex over λ_i,[1] so we can apply the result of Lemma 5.2 to $\bar{F}(\lambda_i, f_i)$.

Theorem 5.3 *It is always true that* $\quad \bar{F}(S)_u \geq \bar{F}(S)_p$. □

Proof By definition, the uniform policy is $f_i = f$ for any i. Then

$$\bar{F}(S)_u = \frac{1}{N}\sum_{i=1}^{N} \bar{F}(\lambda_i, f_i) = \frac{1}{N}\sum_{i=1}^{N} \bar{F}(\lambda_i, f). \tag{5.12}$$

For the proportional policy, $\lambda_i/f_i = \lambda/f$ for any i, so the freshness of e_i is

$$\bar{F}(\lambda_i, f_i) = \bar{F}(\lambda, f)$$

for any synchronization order policy. Therefore,

$$\bar{F}(S)_p = \frac{1}{N}\sum_{i=1}^{N} \bar{F}(\lambda_i, f_i) = \frac{1}{N}\sum_{i=1}^{N} \bar{F}(\lambda, f) = \bar{F}(\lambda, f). \tag{5.13}$$

Then

$$\begin{aligned}
\bar{F}(S)_u &= \frac{1}{N}\sum_{i=1}^{N} \bar{F}(\lambda_i, f) && \text{(Equation 5.12)} \\
&\geq \bar{F}\left(\frac{1}{N}\sum_{i=1}^{N} \lambda_i, f\right) && \text{(convexity of } \bar{F}) \\
&= \bar{F}(\lambda, f) && \text{(definition of } \lambda) \\
&= \bar{F}(S)_p. && \text{(Equation 5.13)}
\end{aligned}$$

■

Similarly, we can prove that the age of the uniform policy, $\bar{A}(S)_u$, is always less than the age of the proportional policy, $\bar{A}(S)_p$, based on the concavity of $\bar{A}(\lambda_i, f_i)$ over λ_i.

Theorem 5.4 *It is always true that* $\quad \bar{A}(S)_u \leq \bar{A}(S)_p$. □

[1] We can verify that $\bar{F}(\lambda_i, f_i)$ is convex over λ_i by computing $\partial^2 \bar{F}(\lambda_i, f_i)/\partial \lambda_i^2$ for the analytic forms.

Proof From the definition of the uniform and the proportional policies,

$$\bar{A}(S)_u = \frac{1}{N}\sum_{i=1}^{N} \bar{A}(\lambda_i, f) \tag{5.14}$$

$$\bar{A}(S)_p = \frac{1}{N}\sum_{i=1}^{N} \bar{A}(\lambda_i, f_i) = \frac{1}{N}\sum_{i=1}^{N} \frac{\lambda}{\lambda_i}\bar{A}(\lambda, f). \tag{5.15}$$

Then

$$\begin{aligned}
\bar{A}(S)_u &= \frac{1}{N}\sum_{i=1}^{N} \bar{A}(\lambda_i, f) \text{ (Equation 5.14)} \\
&\leq \bar{A}(\frac{1}{N}\sum_{i=1}^{N} \lambda_i, f)\text{(concavity of } \bar{A}\text{)} \\
&= \bar{A}(\lambda, f) \qquad \text{(definition of } \lambda\text{)}
\end{aligned}$$

$$\begin{aligned}
\bar{A}(S)_p &= \lambda\bar{A}(\lambda, f)\left(\frac{1}{N}\sum_{i=1}^{N}\frac{1}{\lambda_i}\right) && \text{(Equation 5.15)} \\
&\geq \lambda\bar{A}(\lambda, f)\frac{1}{\frac{1}{N}\sum_{i=1}^{N}\lambda_i} && \text{(convexity of function } \frac{1}{x}\text{)} \\
&= \lambda\bar{A}(\lambda, f)\frac{1}{\lambda} && \text{(definition of } \lambda\text{)} \\
&= \bar{A}(\lambda, f).
\end{aligned}$$

Therefore, $\bar{A}(S)_u \leq \bar{A}(\lambda, f) \leq \bar{A}(S)_p$. ∎

5.5.3 Two element database

Intuitively, we expected that the proportional policy would be better than the uniform policy, because we allocate more resources to the elements that change more often, which may need more of our attention. But why is it the other way around? In this subsection, we try to understand why we get the counterintuitive result, by studying a very simple example: a database consisting of two elements. The analysis of this simple example will let us understand the result more concretely, and it will reveal some intuitive trends. We

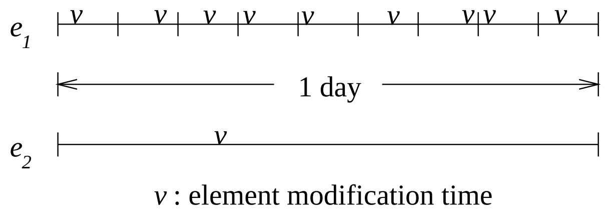


Figure 5.12: A database with two elements with different change frequency

will confirm the trends more precisely when we study the optimal synchronization policy later in Section 5.5.4.

Now we analyze a database consisting of two elements: e_1 and e_2. For the analysis, we assume that e_1 changes at 9 times/day and e_2 changes at once/day. We also assume that our goal is to maximize the freshness of the database averaged over time. In Figure 5.12, we visually illustrate our simple model. For element e_1, one day is split into 9 intervals, and e_1 changes *once and only once* in each interval. However, we do not know exactly when the element changes in one interval. For element e_2, it changes *once and only once* per day, and we do not know when it changes. While this model is not exactly a Poisson process model, we adopt this model due to its simplicity and concreteness.

Now let us assume that we decided to synchronize only *one* element per day. Then what element should we synchronize? Should we synchronize e_1 or should we synchronize e_2? To answer this question, we need to compare how the freshness changes if we pick one element over the other. If the element e_2 changes in the middle of the day and if we synchronize e_2 right after it changed, it will remain up-to-date for the remaining half of the day. Therefore, by synchronizing element e_2 we get 1/2 day "benefit"(or freshness increase). However, the probability that e_2 changes before the middle of the day is 1/2, so the "expected benefit" of synchronizing e_2 is $1/2 \times 1/2$ day $= 1/4$ day. By the same reasoning, if we synchronize e_1 in the middle of an interval, e_1 will remain up-to-date for the remaining half of the interval (1/18 of the day) with probability 1/2. Therefore, the expected benefit is $1/2 \times 1/18$ day $= 1/36$ day. From this crude estimation, we can see that it is more effective to select e_2 for synchronization!

Table 5.4 shows the expected benefits for several other scenarios. The second column shows the total synchronization frequencies $(f_1 + f_2)$ and the third column shows how much of the synchronization is allocated to f_1 and f_2. In the fourth column we estimate the expected benefit, and in the last column we show the f_1 and f_2 values that give the *highest*

row	f_1+f_2	f_1	f_2	benefit	best	
(a)	1	1	0	$\frac{1}{2}\times\frac{1}{18}=\frac{1}{36}$	0	1
(b)		0	1	$\frac{1}{2}\times\frac{1}{2}=\frac{9}{36}$		
(c)	2	2	0	$\frac{1}{2}\times\frac{1}{18}+\frac{1}{2}\times\frac{1}{18}=\frac{2}{36}$	0	2
(d)		1	1	$\frac{1}{2}\times\frac{1}{18}+\frac{1}{2}\times\frac{1}{2}=\frac{10}{36}$		
(e)		0	2	$\frac{1}{3}\times\frac{2}{3}+\frac{1}{3}\times\frac{1}{3}=\frac{12}{36}$		
(f)	5	3	2	$\frac{3}{36}+\frac{12}{36}=\frac{30}{72}$	2	3
(g)		2	3	$\frac{2}{36}+\frac{6}{16}=\frac{31}{72}$		
(h)	10	9	1	$\frac{9}{36}+\frac{1}{4}=\frac{36}{72}$	7	3
(i)		7	3	$\frac{7}{36}+\frac{6}{16}=\frac{41}{72}$		
(j)		5	5	$\frac{5}{36}+\frac{15}{36}=\frac{40}{72}$		

Table 5.4: Estimation of benefits for different choices

expected benefit. To reduce clutter, when $f_1+f_2=5$ and 10, we show only some interesting (f_1,f_2) pairs. Note that since $\lambda_1=9$ and $\lambda_2=1$, row (h) corresponds to the proportional policy ($f_1=9, f_2=1$), and row (j) corresponds to the uniform policy ($f_1=f_2=5$). From the table, we can observe the following interesting trends:

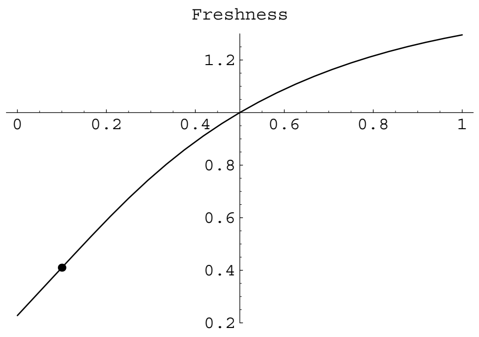
1. **Rows (a)-(e):** When the synchronization frequency (f_1+f_2) is much smaller than the change frequency ($\lambda_1+\lambda_2$), it is better to give up synchronizing the elements that change too fast. In other words, when it is not possible to keep up with everything, it is better to focus on what we can track.

2. **Rows (h)-(j):** Even if the synchronization frequency is relatively large ($f_1+f_2=10$), the uniform allocation policy (row (j)) is more effective than the proportional allocation policy (row (h)). The optimal point (row (i)) is located somewhere between the proportional policy and the uniform policy.

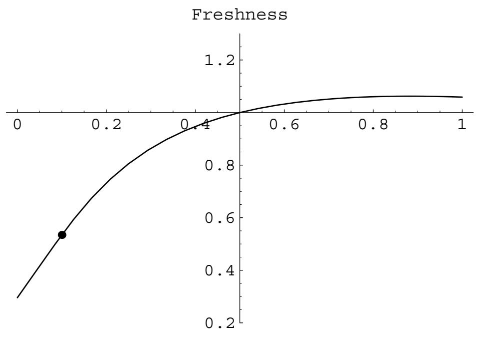
We can verify this trend using our earlier analysis based on a Poisson process. We assume that the changes of e_1 and e_2 are Poisson processes with change frequencies $\lambda_1=9$ and $\lambda_2=1$. To help the discussion, we use $\bar{F}(\lambda_i,f_i)$ to refer to the time average of freshness of e_i when it changes at λ_i and is synchronized at f_i. Then, the freshness of the database is

$$\bar{F}(S)=\frac{1}{2}(\bar{F}(e_1)+\bar{F}(e_2))=\frac{1}{2}(\bar{F}(\lambda_1,f_1)+\bar{F}(\lambda_2,f_2))=\frac{1}{2}(\bar{F}(9,f_1)+\bar{F}(1,f_2)).$$

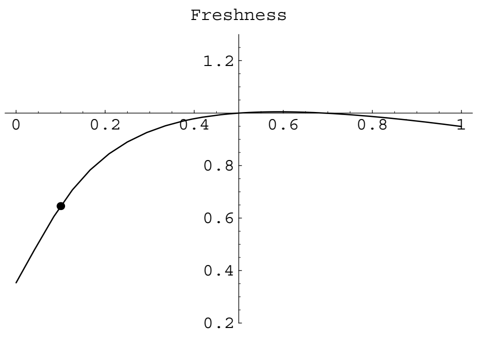
When we fix the value of f_1+f_2, the above equation has only one degree of freedom, and we can plot $\bar{F}(S)$ over, say, f_2. In Figure 5.13, we show a series of graphs obtained this



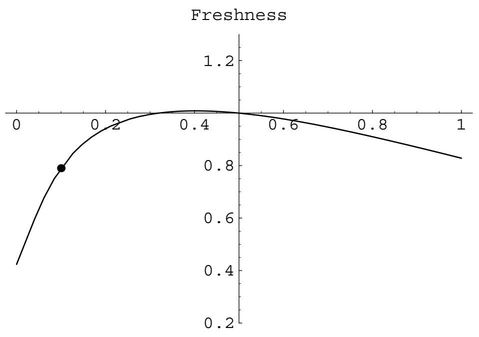
(a) $f_1 + f_2 = 1$



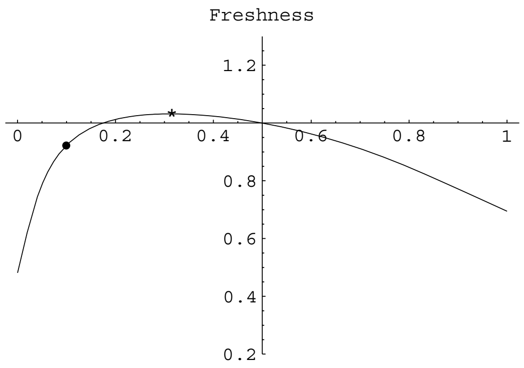
(b) $f_1 + f_2 = 2$



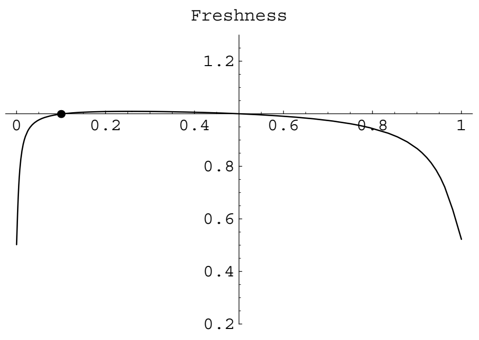
(c) $f_1 + f_2 = 3$



(d) $f_1 + f_2 = 5$



(e) $f_1 + f_2 = 10$



(f) $f_1 + f_2 = 100$

Horizontal axis: fraction of the synchronization resources allocated to e_2

Figure 5.13: Series of freshness graphs for different synchronization frequency constraints. In all of the graphs, $\lambda_1 = 9$ and $\lambda_2 = 1$.

way. The horizontal axis here represents the fraction of the synchronization allocated to e_2. That is, when $x = 0$, we do not synchronize element e_2 at all ($f_2 = 0$), and when $x = 1$ we synchronize only element e_2 ($f_1 = 0$ or $f_2 = f_1 + f_2$). Therefore, the middle point ($x = 0.5$) corresponds to the uniform policy ($f_1 = f_2$), and $x = 0.1$ point corresponds to the proportional policy (Remember that $\lambda_1 = 9$ and $\lambda_2 = 1$). The vertical axis in the graph shows the *normalized* freshness of the database. We normalized the freshness so that $\bar{F}(S) = 1$ at the uniform policy ($x = 0.5$). To compare the uniform and the proportional policies more clearly, we indicate the freshness of the proportional policy by a dot, and the x and the y axes cross at the uniform policy.

From these graphs, we can clearly see that the uniform policy is always better than the proportional policy, since the dots are always below the origin. Also note that when the synchronization frequency is small (graph (a) and (b)), it is better to give up on the element that changes too often (We get the highest freshness when $x = 1$ or $f_1 = 0$). When $f_1 + f_2$ is relatively large (graph (e) and (f)), the optimal point is somewhere between the uniform policy and the proportional policy. For example, the freshness is highest when $x \approx 0.3$ in Figure 5.13(e) (the star in the graph).

5.5.4 The optimal resource-allocation policy

From the previous discussion, we learned that the uniform policy is indeed better than the proportional policy. Also, we learned that the optimal policy is neither the uniform policy nor the proportional policy. For instance, we get the highest freshness when $x \approx 0.3$ for Figure 5.13(e). Then, what is the best way to allocate the resource to elements for a general database S? In this section, we will address this question. More formally, we will study how often we should synchronize individual elements when we know how often they change, in order to maximize the freshness or age. Mathematically, we can formulate our goal as follows:

Problem 5.1 Given λ_i's ($i = 1, 2, \ldots, N$), find the values of f_i's ($i = 1, 2, \ldots, N$) which maximize

$$\bar{F}(S) = \frac{1}{N}\sum_{i=1}^{N}\bar{F}(e_i) = \frac{1}{N}\sum_{i=1}^{N}\bar{F}(\lambda_i, f_i)$$

	e_1	e_2	e_3	e_4	e_5
(a) change frequency	1	2	3	4	5
(b) synchronization frequency (freshness)	1.15	1.36	1.35	1.14	0.00
(c) synchronization frequency (age)	0.84	0.97	1.03	1.07	1.09

Table 5.5: The optimal synchronization frequencies of Example 5.4

when f_i's satisfy the constraints

$$\frac{1}{N}\sum_{i=1}^{N} f_i = f \quad \text{and} \quad f_i \geq 0 \quad (i = 1, 2, \ldots, N) \qquad \square$$

Because we can derive the closed form of $\bar{F}(\lambda_i, f_i)$,[2] we can solve the above problem by the *method of Lagrange multipliers* [Tho69].

Solution The freshness of database S, $\bar{F}(S)$, takes its maximum when all f_i's satisfy the equations

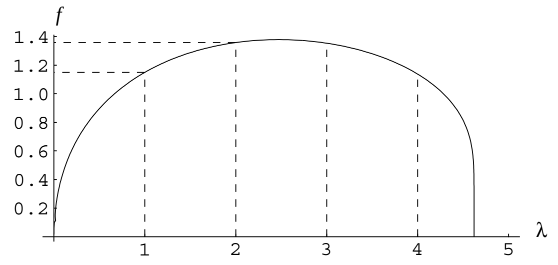
$$\frac{\partial \bar{F}(\lambda_i, f_i)}{\partial f_i} = \mu^3 \quad \text{and} \quad \frac{1}{N}\sum_{i=1}^{N} f_i = f.$$

Notice that we introduced another variable μ in the solution,[4] and the solution consists of $(N+1)$ equations (N equations of $\partial \bar{F}(\lambda_i, f_i)/\partial f_i = \mu$ and one equation of $\frac{1}{N}\sum_{i=1}^{N} f_i = f$) with $(N+1)$ unknown variables $(f_1, \ldots, f_N, \mu)$. We can solve these $(N+1)$ equations for f_i's, since we know the closed form of $\bar{F}(\lambda_i, f_i)$. ■

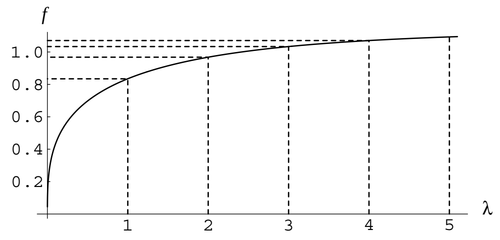
From the solution, we can see that all optimal f_i's satisfy $\partial \bar{F}(\lambda_i, f_i)/\partial f_i = \mu$. That is, all optimal (λ_i, f_i) pairs are on the graph of $\partial \bar{F}(\lambda, f)/\partial f = \mu$. To illustrate the property of the solution, we use the following example.

Example 5.4 The real-world database consists of five elements, which change at the frequencies of 1, 2, ..., 5 (times/day). We list the change frequencies in row (a) of Table 5.5. (We explain the meaning of rows (b) and (c) later, as we continue our discussion.) We decided to synchronize the local database at the rate of 5 elements/day total, but we still need to find out how often we should synchronize each element.

[2]For instance, $\bar{F}(\lambda_i, f_i) = (1 - e^{-\lambda_i/f_i})/(\lambda_i/f_i)$ for the fixed-order policy.
[3]When $\partial \bar{F}(\lambda_i, f_i)/\partial f_i = \mu$ does not have a solution with $f_i \geq 0$, f_i should be equal to zero.
[4]This is a typical artifact of the method of Lagrange multipliers.



(a) change frequency vs. synchronization frequency for freshness optimization



(b) change frequency vs. synchronization frequency for age optimization

Figure 5.14: Solution of the freshness and age optimization problem of Example 5.4

For this example, we can solve Problem 5.1 numerically, and we show the graph of its solution in Figure 5.14(a). The horizontal axis of the graph corresponds to the change frequency of an element, and the vertical axis shows the optimal synchronization frequency of the element with that given change frequency. For instance, the optimal synchronization frequency of e_1 is 1.15 ($f = 1.15$), because the change frequency of element e_1 is 1 ($\lambda = 1$). Similarly from the graph, we can find the optimal synchronization frequencies of other elements, and we list them in row (b) of Table 5.5.

Notice that while e_4 changes twice as often as e_2, we need to synchronize e_4 less frequently than e_2. Furthermore, the synchronization frequency of e_5 is zero, while it changes at the highest rate. This result comes from the shape of Figure 5.14(a). In the graph, when $\lambda > 2.5$, f decreases as λ increases. Therefore, the synchronization frequencies of the elements e_3, e_4 and e_5 gets smaller and smaller. □

While we obtained Figure 5.14(a) by solving Example 5.4, we prove in the next subsection that the shape of the graph is the same for *any* distributions of λ_i's. That is, the optimal graph for *any* database S is *exactly the same* as Figure 5.14(a), except that the graph of S is scaled by a constant factor from Figure 5.14(a). Since the shape of the graph is always the same, the following statement is true in any scenario: *To improve freshness, we should penalize the elements that change too often.*

Similarly, we can compute the optimal *age* solution for Example 5.4, and we show the result in Figure 5.14(b). The axes in this graph are the same as before. Also, we list the optimal synchronization frequencies in row (c) of Table 5.5. Contrary to the freshness, we can observe that we should synchronize an element more often when it changes more often

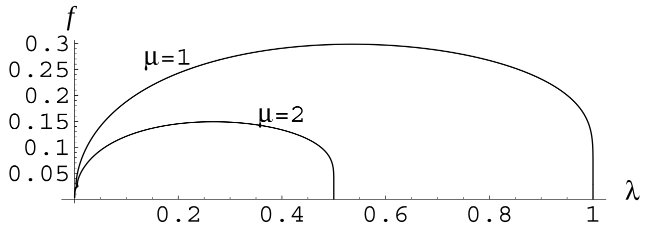


Figure 5.15: $\partial \bar{F}(\lambda, f)/\partial f = \mu$ graph when $\mu = 1$ and $\mu = 2$

$(f_1 < \cdots < f_5)$. However, notice that the difference between the synchronization frequencies is marginal: All f_i's are approximately close to one. In other words, the optimal solution is rather closer to the uniform policy than to the proportional policy. Similarly for age, we can prove that the shape of the optimal age graph is always the same as Figure 5.14(b). Therefore, the trend we observed here is very general and holds for *any* database.

5.5.5 The graph of the equation $\partial \bar{F}(\lambda, f)/\partial f = \mu$

We now study the property of the solution for Problem 5.1. In particular, we prove that the graph $\partial \bar{F}(\lambda, f)/\partial f = \mu$ is only scaled by a constant factor for different μ values. For example, Figure 5.15 shows the graphs of the equation for $\mu = 1$ and 2. The graph clearly shows that the $\mu = 2$ graph is simply a scaled version of the $\mu = 1$ graph by a factor of 2; the $\mu = 1$ graph converges to zero when $\lambda \to 1$, while the $\mu = 2$ graph converges to zero when $\lambda \to 1/2$. Also, the $\mu = 1$ graph takes its maximum at $(0.53, 0.3)$, while the $\mu = 2$ takes its maximum at $(0.53/2, 0.3/2)$. The following theorem proves this scaling property for any μ.

Theorem 5.5 *Let (λ, f) satisfy the equation $\frac{\partial \bar{F}}{\partial f}(\lambda, f) = \mu$. Then $(\mu\lambda, \mu f)$, the point scaled by the factor μ from (λ, f), satisfies the equation $\frac{\partial \bar{F}}{\partial f}(\mu\lambda, \mu f) = 1$.* □

Proof For the fixed-order policy,[5]

$$\frac{\partial \bar{F}}{\partial f}(\lambda, f) = -\frac{e^{-\lambda/f}}{f} + \frac{1 - e^{-\lambda/f}}{\lambda}.$$

[5]We can similarly prove the theorem for other synchronization-order policies.

Then,

$$\begin{aligned}
\frac{\partial \bar{F}}{\partial f}(\mu\lambda, \mu f) &= -\frac{e^{-\mu\lambda/\mu f}}{\mu f} + \frac{1 - e^{-\mu\lambda/\mu f}}{\mu\lambda} \\
&= \frac{1}{\mu}\left(-\frac{e^{-\lambda/f}}{f} + \frac{1 - e^{-\lambda/f}}{\lambda}\right) \\
&= \frac{1}{\mu}\,\frac{\partial \bar{F}}{\partial f}(\lambda, f) \\
&= \frac{1}{\mu}\,\mu & \text{(assumption)} \\
&= 1.
\end{aligned}$$

Therefore, the points $(\mu\lambda, \mu f)$ satisfies the equation $\frac{\partial \bar{F}}{\partial f}(\mu\lambda, \mu f) = 1$. ■

From Theorem 5.5, we can see that the graph of $\partial\bar{F}(\lambda, f)/\partial f = \mu$ for any μ value is essentially a *scaled version* of the graph $\partial\bar{F}(\lambda, f)/\partial f = 1$. Therefore, its shape is *always* the same for *any* value of μ. Since μ is the only constant affected by the distribution of λ_i's, this "scaling property" for different μ values shows that the shape of the graph is *always* the same under any distribution.

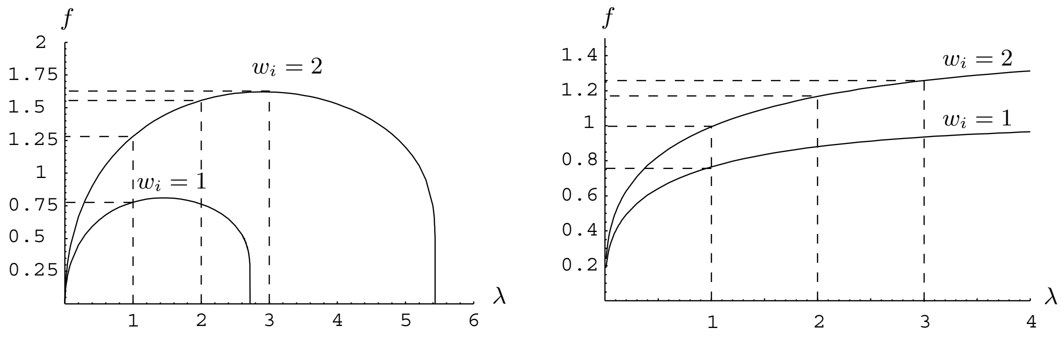
Similarly for age, we can prove that the graphs are always the same except a constant scaling factor.

5.6 Extension: weighted model

So far, we assumed that the freshness and the age of every element is equally important to the users. But what if the elements have different "importance"? For example, if the database S has two elements, e_1 and e_2, and if the users access e_1 twice as often as e_2, how should we synchronize the elements to maximize the freshness of the database *perceived by the users*? Should we refresh e_1 more often than e_2? In this section, we study how we can extend our model to address this scenario.

5.6.1 Weighted freshness metrics

When the users access element e_1 2 times more often than e_2, we may consider that e_1 is twice as important as e_2, because we can make the users see fresh elements twice as often by



(a) change frequency vs. synchronization frequency for freshness optimization

(b) change frequency vs. synchronization frequency for age optimization

Figure 5.16: Solution of the freshness and age optimization problem of Example 5.5

maintaining e_1 up-to-date. To capture this concept of "importance", we may give different "weights" w_i's to element e_i's and define the freshness of a database as follows:

Definition 5.4 (Weighted freshness and age) Let w_i be the weight (or importance) of element e_i. The freshness and the age of a database S is defined as

$$F(S;t) = \frac{1}{\sum_{i=1}^{N} w_i}\left[\sum_{i=1}^{N} w_i F(e_i;t)\right]$$
$$A(S;t) = \frac{1}{\sum_{i=1}^{N} w_i}\left[\sum_{i=1}^{N} w_i A(e_i;t)\right]$$ □

Note that when all w_i's are equal to 1, the above definition reduces to our previous definition. Under this weighted definition, we can analyze the Poisson-process model using a similar technique described in Section 5.5.4. We illustrate the result with the following example.

Example 5.5 We maintain 6 elements in our local database, whose weights and change frequencies are listed in the rows (a) and (b) of Table 5.6. In this example, there exist two classes of elements, the elements with weight 1 (e_{1*}) and the elements with weight 2 (e_{2*}). Each class contains 3 elements (e_{*1}, e_{*2}, e_{*3}) with different change frequencies, and we assume that we synchronize a total of 6 elements every day. Under these parameters, we can analyze the freshness maximization problem, whose result is shown in Figure 5.16(a).

The horizontal axis of the graph represents the change frequency of an element, and the

	e_{11}	e_{12}	e_{13}	e_{21}	e_{22}	e_{23}
(a) weight (w_i)	1			2		
(b) change frequency (times/day)	1	2	3	1	2	3
(c) synchronization frequency (freshness)	0.78	0.76	0.00	1.28	1.56	1.62
(d) synchronization frequency (age)	0.76	0.88	0.94	0.99	1.17	1.26

Table 5.6: The optimal synchronization frequencies of Example 5.5

vertical axis shows the optimal synchronization frequency for the given change frequency. The graph consists of two curves because the elements in different classes follow different curves. The $w_i = 2$ elements (e_{2*}) follow the outer curve, and the $w_i = 1$ elements (e_{1*}) follow the inner curve. For example, the optimal synchronization frequency if e_{11} is 0.78, because its weight is 1 and its change frequency is once a day. We list the optimal synchronization frequencies for all elements in row (c) of Table 5.6. From this result, we can clearly see that we should visit an element more often when its weight is higher. For instance, the synchronization frequency of e_{21} is 1.28, which is higher than 0.78 of e_{11}. (Note that both e_{21} and e_{11} change at the same frequency.)

However, note that the optimal synchronization frequency is not proportional to the weight of an element. For example, the optimal synchronization frequency of e_{23} is infinitely larger than that of e_{13}! In fact, we can *prove* that the weight of an element determines the *scale of the optimal curve* that the element follows. That is, the optimal curve that $w_i = 2$ elements follow is *exactly twice* as large as the curve that $w_i = 1$ elements follow. Mathematically, we can *prove* the following property of the optimal curve under *any* scenario: *When the weight of e_1 is k times as large as that of e_2, the optimal freshness curve that e_1 follows is k times larger than the curve that e_2 follows.*

Similarly, we can analyze the weighted age metric for the example, and we show the result in Figure 5.16(b). Again, we can see that we should visit an element more often when its weight is higher, but not necessarily proportionally more often. In general, we can prove the following property for age: *When the weight of e_1 is k times as large as that of e_2, the optimal age curve that e_1 follows is $\sqrt{k}$ times larger than the curve that e_2 follows.* □

5.7 Improvement from optimal policies

In Chapter 4, we reported how many pages change how often in Figure 4.2. In the figure, the horizontal axis represents the average change interval of pages, and the vertical axis

policy	*Freshness*	*Age*
Proportional	0.12	400 days
Uniform	0.57	5.6 days
Optimal	0.62	4.3 days

Table 5.7: Freshness and age prediction based on the real Web data

shows the fraction of the pages that changed at the given average interval. For instance, we see that about 23% of pages changed more than once a day from the first bar of Figure 4.2.

From this data, we can estimate how much improvement we can get, if we adopt the optimal allocation policy. For the estimation, we assume that we maintain 100 million pages locally and that we synchronize all pages every month.[6] Also based on Figure 4.2, we assume that 23% of pages change every day, 15% of pages change every week, etc. For the pages that did not change in 4 months, we assume that they change every year. While it is a crude approximation, we believe we can get some idea on how effective different policies are.

In Table 5.7, we show the predicted freshness and age for various resource-allocation policies. To compute the numbers, we assumed the fixed-order policy (Item 3a in Section 5.3) as the synchronization-order policy. We can clearly see that the optimal policy is significantly better than any other policies. For instance, the freshness increases from 0.12 to 0.62 (500% increase!), if we use the optimal policy instead of the proportional policy. Also, the age decreases by 23% from the uniform policy to the optimal policy. From these numbers, we can also learn that we need to be very careful when we optimize the policy based on the frequency of changes. For instance, the proportional policy, which people may intuitively prefer, is significantly worse than any other policies: The age of the proportional policy is 100 times worse than that of the optimal policy!

5.8 Related work

In general, we may consider the problem of this chapter as a replicated-data-maintenance problem. A lot of work has been done to maintain replicated-data consistency [BG84, ABGM90, BBC80, BGM95, dCR82, CKLlSM97, KB91, KB94, LLSG92, GL93, PL91, OW00].

[6]Many popular search engines report numbers similar to these.

This body of work studies the tradeoff between consistency and read/write performance [KB94, LLSG92, ABGM90, OW00] and tries to guarantee a certain type of consistency [ABGM90, BGM95, BBC80]. For example, reference [YV00] tries to limit that the number of pending writes that have not been propagated to all replicas and proposes a new distributed protocol. Reference [OW00] guarantees an interval bound on the values of replicated data through the cooperation of the source data.

In most of the existing work, however, researchers have assumed a *push* model, where the sources *notify* the replicated data sites of the updates. In the Web context this push model is not very appropriate, because most of the Web site managers do not inform others of the changes they made. We need to assume a *poll* model where updates are made *independently* and *autonomously* at the sources.

Reference [CLW98] studies how to schedule a Web crawler to improve freshness. The model used for Web pages is similar to the one used in this dissertation; however, the model for the Web crawler and freshness are very different. In particular, the reference assumes that the "importance" or the "weight" of a page is proportional to the change frequency of the page. While this assumption makes the analysis simple, it also prevented the authors from discovering the fundamental trend that we identified in this thesis. We believe the result of this thesis is more general, because we study the impact of the change frequency and the importance of a page separately. We also proposed an *age* metric, which was not studied the the reference.

5.9 Conclusion

In this chapter we studied how a crawler should refresh Web pages to improve their freshness and age. We presented a formal framework, which provides a theoretical foundation for this problem, and we studied the effectiveness of various policies. In our study we identified a potential pitfall (proportional synchronization), and proposed an optimal policy that can improve freshness and age very significantly. Finally, we estimated potential improvement from our policies based on the experimental data described in Chapter 4.

As more and more digital information becomes available on the Web, it will be increasingly important to collect it effectively. A crawler simply cannot refresh all its data constantly, so it must be very careful in deciding what data to poll and check for freshness. The policies we have studied in this chapter can make a significant difference in the "temporal quality" of the data that is collected.

Chapter 6

Change Frequency Estimation

6.1 Introduction

In order to implement the refresh policies in the previous chapter, a crawler needs to *estimate* how often Web pages change, because Web pages are updated autonomously by independent maintainers. For instance, popular news Web sites, such as CNN and NY Times, update their contents whenever there are new developments, and many online stores update the price/availability of their products depending on their inventory and on market conditions. In these cases, the crawler usually does not know exactly when and how often the pages change.

In this chapter we study how a crawler can estimate the change frequencies of Web pages. While this study is motivated by the need of a Web crawler, we believe that our results are very general and our techniques can be applied to other scenarios. For instance, the following applications can use our estimation techniques to improve their effectiveness.

- **Improving Web caching:** A Web cache saves recently accessed Web pages, so that the next access to a page can be served locally. Caching pages reduces the number of remote accesses to minimize access delay and network bandwidth. Typically, a Web cache uses an LRU (least recently used) *page replacement policy*, but it may improve the *cache hit ratio* by estimating how often a page changes. For example, if a page was cached a day ago and if the page changes every hour on average, the system may safely discard that page, because the cached page is most probably obsolete.

- **Improving the update policy of a data warehouse:** A data warehouse maintains a local snapshot, called a *materialized view*, of underlying data sources, which are often autonomous. This materialized view is usually updated during off-peak hours, to minimize the impact on the underlying source data. As the size of the data grows, however, it becomes more difficult to update the view within the limited time window. If we can estimate how often an individual data item (e.g., a row in a table) changes, we may selectively update only the items likely to have changed, and thus incorporate more changes within the same amount of time.

- **Data mining:** In many cases, the frequency of change itself might be useful information. For instance, when a person suddenly accesses his bank account very often, it may signal fraud, and the bank may wish to take an appropriate action.

In this chapter, we study how we can effectively estimate how often a Web page (in general, a data element) changes. We assume that we access an element *repeatedly* through *normal* activities, such as a periodic crawling of Web pages or the users' repeated access to Web pages. From these repeated accesses, we detect changes to the element, and then we estimate its change frequency.

We have motivated the usefulness of "estimating the frequency of change," and how we accomplish the task. However, there exist important challenges in estimating the frequency of change, including the following:

1. **Incomplete change history:** Often, we do not have complete information on how often and when an element changed. For instance, a Web crawler can tell if a page has changed between accesses, but it cannot tell how many times the page changed.

   Example 6.1 A Web crawler accessed a page on a daily basis for 10 days, and it detected 6 changes. From this data, the crawler may naively conclude that its change frequency is $6/10 = 0.6$ times a day. But this estimate can be smaller than the actual change frequency, because the page may have changed more than once between some accesses. Then, what would be the fair estimate for the change frequency? How can the crawler account for the missed changes? □

   Previous work has mainly focused on how to estimate the change frequency given the complete change history [TK98, Win72]. As far as we know, there has been no work on how to estimate the frequency based on incomplete information.

2. **Irregular access interval:** In certain applications, such as a Web cache, we cannot control how often and when a data item is accessed. The access is entirely decided by the user's request pattern, so the access interval can be arbitrary. When we have limited change history and when the access pattern is irregular, it becomes very difficult to estimate the change frequency.

   Example 6.2 In a Web cache, a user accessed a Web page 4 times, at day 1, day 2, day 7 and day 10. In these accesses, the system detected changes at day 2 and day 7. Then what can the system conclude about its change frequency? Does the page change every (10 days)/2 = 5 days on average? □

3. **Difference in available information:** Depending on the application, we may get different levels of information for different data items. For instance, certain Web sites tell us when a page was last-modified, while many Web sites do not provide this information. Depending on the scenario, we may need different "estimators" for the change frequency, to fully exploit the available information.

In this chapter we study how we can estimate the frequency of change when we have *incomplete* change history of a data item. To that end, we first identify various issues and place them into a taxonomy (Section 6.2). Then, for each branch in the taxonomy, we propose an "estimator" and show analytically how good the proposed estimator is (Sections 6.4 through 6.6). In summary, this chapter makes the following contributions:

- We identify the problem of estimating the frequency of change and we present a formal framework to study the problem.

- We propose several estimators that measure the frequency of change much more effectively than existing ones. For the scenario of Example 6.1, for instance, our estimator will predict that the page changes 0.8 times per day (as opposed to the 0.6 we guessed earlier), which reduces the "bias" by 33% on average.

- We present analytical and experimental results that show how precise/effective our proposed estimators are.

6.2 Taxonomy of issues

Before we start discussing how to estimate the change frequency of an element, we first need to clarify what we mean by "change of an element." What do we mean by an "element" and what does "change" mean? To make our discussion concrete, we assume that an element is a Web page and that the change is *any* modification to the page. However, note that the technique that we develop is independent of this assumption. The element can be defined as a whole Web site or a single row in a database table, etc. Also the change may be defined as more than, say, a 30% modification to the page, or as updates to more than 3 columns of the row. Regardless of the definition, we can apply our technique/analysis, as long as we have a clear notion of the element and a precise mechanism to detect changes to the element.

Given a particular definition of an element and change, we assume that we repeatedly access an element to estimate how often the element changes. This access may be performed at a regular interval or at random intervals. Also, we may acquire different levels of information at each access. Based on how we access the element and what information is available, we develop the following taxonomy.

1. **How do we trace the history of an element?** In this chapter we assume that we repeatedly access an element, either actively or passively.

    - **Passive monitoring:** We do not have any control over when and how often we access an element. In a Web cache, for instance, Web pages are accessed only when users access the page. In this case, the challenge is how to analyze the *given* change history to best estimate its change frequency.
    - **Active monitoring:** We actively monitor the changes of an element and can control the access to the element. For instance, a crawler can decide how often and when it will visit a particular page. When we can control the access, another important question is how often we need to access a particular element to best estimate its change frequency. For instance, if an element changes about once a day, it might be unnecessary to access the element every minute, while it might be insufficient to access it every month.

    In addition to the access control, different applications may have different access intervals.

   - **Regular interval:** In certain cases, especially for active monitoring, we may access the element at a regular interval. Obviously, estimating the frequency of change will be easier when the access interval is regular. In this case, (number of detected changes)/(monitoring period) may give us a good estimation of the change frequency.
   - **Random interval:** Especially for passive monitoring, the access intervals might be irregular. In this case, frequency estimation is more challenging.

2. **What information do we have?** Depending on the application, we may have different levels of information regarding the changes of an element.

   - **Complete history of changes:** We know exactly when and how many times the element changed. In this case, estimating the change frequency is relatively straightforward; it is well known that (number of changes)/(monitoring period) gives a "good" estimation of the frequency of change [TK98, Win72]. In this thesis, we do not study this case.
   - **Last date of change:** We know when the element was last modified, but not the complete change history. For instance, when we monitor a bank account which records the last transaction date and its current balance, we can tell when the account was last modified by looking at the transaction date.
   - **Existence of change:** The element that we monitor may not provide any history information and only give us its current status. In this case, we can compute the "signature" of the element at each access and compare these signatures between accesses. By comparing signatures, we can tell whether the element changed or not. However, we cannot tell how many times or when the element changed by this method.

   In Section 6.4, we study how we can estimate the frequency of change, when we only know whether the element changed or not. Then in Section 6.5, we study how we can exploit the "last-modified date" to better estimate the frequency of change.

3. **How do we use estimated frequency?** Different applications may use the frequency of change for different purposes.

- **Estimation of frequency:** In data mining, for instance, we may want to study the correlation between how often a person uses his credit card and how likely it is that the person will default. In this case, it might be important to *estimate* the frequency accurately.
- **Categorization of frequency:** We may only want to classify the elements into several frequency categories. For example, a Web crawler may perform a "small-scale" crawl every week, crawling only the pages that are updated very often. Also, the crawler may perform a "complete" crawl every three months to completely refresh all pages. In this case, the crawler may not be interested in exactly how often a page changes. It may only want to *classify* pages into two categories, the pages to visit every week and the pages to visit every three months.

In Section 6.4 and 6.5, we study the problem of *estimating* the frequency of change, and in Section 6.6, we study the problem of *categorizing* the frequency of change.

6.3 Preliminaries

In this section, we will review some of the basic concepts for the estimation of frequency, to help readers understand our later discussion. A reader familiar with estimation theory may skip this section.

6.3.1 Quality of estimator

In this thesis, again, we assume that an element, or a Web page, follows a Poisson process based on the results in Chapter 4. A Poisson process has an associated parameter λ, which is the *average frequency* that a change occurs. Note that it is also possible that the average change frequency λ itself may change over time. In this chapter we primarily assume λ does not change over time. That is, we adopt a *stationary Poisson process* model. Later in Section 6.7, we also discuss possible options when the process is non-stationary. Also in Section 6.8.1, we study how our proposed estimators perform, when the elements do not necessarily follow a Poisson process.

The goal of this chapter is to estimate the frequency of change λ, from repeated accesses to an element. To estimate the frequency, we need to summarize the observed change history,

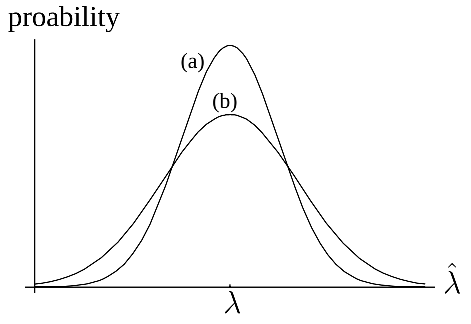


Figure 6.1: Two possible distributions of the estimator $\hat{\lambda}$

or *samples*, as a single number that corresponds to the frequency of change. In Example 6.1, for instance, we summarized the six changes in ten visits as the change frequency of $6/10 = 0.6$/day. We call this summarization procedure the *estimator* of the change frequency. Clearly, there exist multiple ways to summarize the same observed data, which can lead to different change frequencies. In this subsection, we will study how we can compare the effectiveness of various estimators.

An estimator is often expressed as a function of the observed variables. For instance, let X be the number of changes that we detected and T be the total access period. Then, we may use $\hat{\lambda} = X/T$ as the estimator of the change frequency λ as we did in Example 6.1. (We use the notation "hat" to show that we want to measure the parameter underneath it.) Here, note that X is a random variable, which is measured by sampling (or repeated accesses). Therefore, the estimator $\hat{\lambda}$ is also a random variable that follows a certain probability distribution. In Figure 6.1, we show two possible distributions of $\hat{\lambda}$. As we will see, the distribution of $\hat{\lambda}$ determines how effective the estimator $\hat{\lambda}$ is.

1. **Unbiasedness:** Let us assume that the element changes at the average frequency λ, which is shown at the bottom center of Figure 6.1. Intuitively we would like the distribution of $\hat{\lambda}$ to be centered around the value λ. Mathematically, $\hat{\lambda}$ is said to be *unbiased*, when the expected value of $\hat{\lambda}$, $E[\hat{\lambda}]$, is equal to λ.

2. **Efficiency:** In Figure 6.1, it is clear that $\hat{\lambda}$ may take a value other than λ, even if $E[\hat{\lambda}] = \lambda$. For any estimator, the estimated value might be different from the real value λ, due to some statistical variation. Clearly, we want to keep the variation as small as possible. We say that the estimator $\hat{\lambda}_1$ is more *efficient* than the estimator $\hat{\lambda}_2$, if the distribution of $\hat{\lambda}_1$ has smaller variance than that of $\hat{\lambda}_2$. In Figure 6.1, for

instance, the estimator with the distribution (a) is more efficient than the estimator of (b).

3. **Consistency:** Intuitively, we expect that the value of $\hat{\lambda}$ approaches λ, as we increase the sample size. This convergence of $\hat{\lambda}$ to λ can be expressed as follows:

   Let $\hat{\lambda}_n$ be the estimator with sample size n. Then $\hat{\lambda}_n$ is said to be a *consistent* estimator of λ if

$$\lim_{n\to\infty} \Pr\{|\hat{\lambda}_n - \lambda| \leq \epsilon\} = 1 \quad \text{for any positive } \epsilon$$

6.4 Estimation of frequency: existence of change

How can we estimate how often an element changes, when we only know whether the element changed or not between our accesses? Intuitively, we may use X/T (X: the number of detected changes, T: monitoring period) as the estimated frequency of change, as we did in Example 6.1. This estimator has been used in most of the previous work that tried to estimate the change frequency of Web pages [DFK99, WM99, WVS$^+$99].

In Section 6.4.1 we first study how effective this naive estimator X/T is, by analyzing its *bias*, *consistency* and *efficiency*. Then in Section 6.4.2, we will propose a new estimator, which is less intuitive than X/T, but is much more effective.

6.4.1 Intuitive frequency estimator: X/T

To help the discussion, we first define some notation. We assume that we access the element at a regular interval I and that we access the element n times. (Estimating the change frequency for an irregular accesses is discussed later Section 6.4.3.) Also, we use X_i to indicate whether the element changed or not in the ith access. More precisely,

$$X_i = \begin{cases} 1 & \text{if the element changed in } i\text{th access,} \\ 0 & \text{otherwise.} \end{cases}$$

Then, X is defined as the sum of all X_i's, $X = \sum_{i=1}^{n} X_i$, and represents the total number of changes that we detected. We use T to refer to the total time elapsed during our n accesses. Since we access the element every I time units, $T = nI = n/f$, where $f(= 1/I)$ is the frequency at which we access the element. We also assume that the changes of the element follow a Poisson process with rate λ. Then, we can define the frequency ratio $r = \lambda/f$, the ratio of the change frequency to the access frequency. When r is large ($\lambda \gg f$), the element changes more often than we access it, and when r is small ($\lambda \ll f$), we access the element more often than it changes.

Note that our goal is to estimate λ, given X_i's and $T(= n/f)$. However, we may estimate the frequency ratio $r(= \lambda/f)$ first and estimate λ indirectly from r (by multiplying r by f). In the rest of this subsection, we will assume that our estimator is the frequency ratio $\hat{r}$, where

$$\hat{r} = \frac{\hat{\lambda}}{f} = \frac{1}{f}\left(\frac{X}{T}\right) = \frac{X}{n}$$

Note that we need to measure X through an experiment and then use the number X to estimate r.

1. **Is the estimator $\hat{r}$ biased?** As we argued in Example 6.1, the estimated $\hat{r}$ will be smaller than the actual r, because the *detected* number of changes, X, will be smaller than the *actual* number of changes. Furthermore, this bias will grow larger as the element changes more often than we access it (i.e, as $r = \lambda/f$ grows larger), because we miss more changes when the element changes more often.

   We can verify this intuition by computing the expected value of $\hat{r}$, $E[\hat{r}]$, and comparing it with the actual r. To compute $E[\hat{r}]$, we first compute the probability q that the element does not change between accesses [TK98].

   Lemma 6.1 *Let q be the probability that the element does not change during time interval I $(= 1/f)$. Then* $q = \Pr\{X(t+I) - X(t) = 0\} = \dfrac{\lambda^0 e^{-\lambda I}}{0!} = e^{-\lambda I} = e^{-\lambda/f} = e^{-r}$ □

   By definition, X_i is equal to zero when the element does not change between the $(i-1)$th and the ith access. Because the change of the element is a Poisson process, the changes at different accesses are independent, and each X_i takes the value

$$X_i = \begin{cases} 1 & \text{with probability } 1-q \\ 0 & \text{with probability } q \ (= e^{-r}) \end{cases}$$

   independently from other X_i's. Then from the definition of X, X is equal to m, when

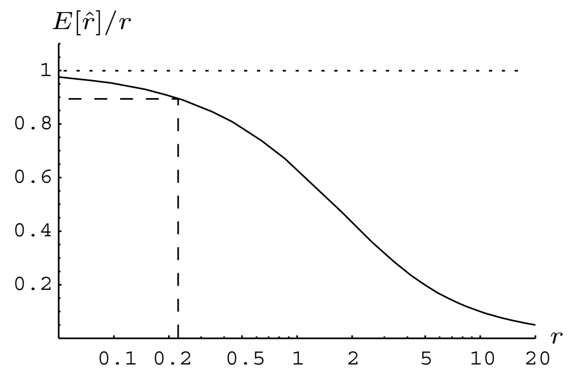


Figure 6.2: Bias of the intuitive estimator $\hat{r} = X/n$

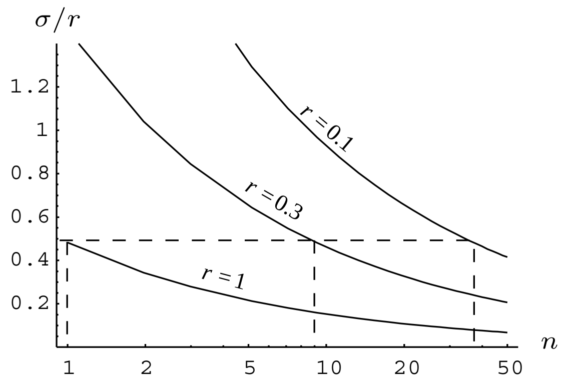


Figure 6.3: Statistical variation of $\hat{r} = X/n$ over n

m X_i's are equal to 1: $\Pr\{X = m\} = \binom{n}{m}(1-q)^m q^{n-m}$. Therefore,

$$E[\hat{r}] = \sum_{m=0}^{n} \frac{m}{n} \Pr\left\{\hat{r} = \frac{m}{n}\right\} = \sum_{m=0}^{n} \frac{m}{n} \Pr\{X = m\} = 1 - e^{-r}$$

Note that for $\hat{r}$ to be unbiased, $E[\hat{r}]$ should be always equal to r. Clearly, $1-e^{-r}$ is not r, and the estimator $\hat{r}$ is biased. In Figure 6.2, we visualize the bias of $\hat{r}$, by plotting $E[\hat{r}]/r$ over r. The horizontal axis is logarithmic because it shows the values more clearly when r is small and large. (In the rest of this chapter, we use a logarithmic scale, whenever convenient.) If $\hat{r}$ is unbiased ($E[\hat{r}] = r$), the graph $E[\hat{r}]/r$ would be equal to 1 for any r (the dotted line), but because the estimated $\hat{r}$ is smaller than the actual r, $E[\hat{r}]/r$ is always less than 1. From the graph, it is clear that $E[\hat{r}]$ is about the same as r ($E[\hat{r}]/r \approx 1$) when r ($= \lambda/f$) is small (i.e., when the element changes less often than we access it), but $E[\hat{r}]$ is significantly smaller than r ($E[\hat{r}]/r \ll 1$), when r is large. Intuitively, this is because we miss more changes as we access the element less often ($r = \lambda/f \gg 1$). From the graph, we can see that the bias is smaller than 10% ($E[\hat{r}]/r > 0.9$) when the frequency ratio r is smaller than 0.21. That is, we should access the element $1/0.21 \approx 5$ times as frequently as it changes, in order to get less than 10% bias.

2. **Is the estimator $\hat{r}$ consistent?** The estimator $\hat{r} = X/n$ is not consistent, because the bias of $\hat{r}$ does not decrease even if we increase the sample size n; the difference between r and $E[\hat{r}]$ ($E[\hat{r}]/r = (1-e^{-r})/r$) remains the same independently of the size

of n.

This result coincides with our intuition; $\hat{r}$ is biased because we miss some changes. Even if we access the element for a longer period, we still miss a certain fraction of changes, if we access the element at the same frequency.

3. **How efficient is the estimator?** To evaluate the efficiency of $\hat{r}$, let us compute its variance.

$$V[\hat{r}] = E[\hat{r}^2] - E[\hat{r}]^2 = e^{-r}(1 - e^{-r})/n$$

Then, the standard deviation of $\hat{r}$ is $\sigma = \sqrt{V[\hat{r}]} = \sqrt{e^{-r}(1 - e^{-r})/n}$

Remember that the standard deviation tells us how clustered the distribution of $\hat{r}$ is around $E[\hat{r}]$; Even if $E[\hat{r}] \approx r$, the estimator $\hat{r}$ may take a value other than r, because our sampling process (or access to the element) inherently induces some statistical variation.

From the basic statistics theory, we know that $\hat{r}$ takes a value in the interval $(E[\hat{r}] - 2\sigma, E[\hat{r}]+2\sigma)$ with 95% probability, assuming $\hat{r}$ follows the normal distribution [WMS97]. In most applications, we want to make this confidence interval (whose length is proportional to σ) small compared to the actual frequency ratio r. Therefore, we want to reduce the ratio of the confidence interval to the frequency ratio, σ/r, as much as we can. In Figure 6.3, we show how this ratio changes over the sample size n by plotting its graph. Clearly, the statistical variation σ/r decreases as n increases; While we *cannot* decrease the *bias* of $\hat{r}$ by increasing the sample size, we *can* minimize the *statistical variation* (or the confidence interval) with more samples.

Also note that when r is small, we need a larger sample size n to get the same variation σ/r. For example, to make $\sigma/r = 0.5$, n should be 1 when $r = 1$, while n should be 9 when $r = 0.3$. We explain what this result implies by the following example.

Example 6.3 A crawler wants to estimate the change frequency of a Web page by visiting the page 10 times, and it needs to decide on the access frequency.

Intuitively, the crawler should not visit the page too slowly, because the crawler misses many changes and the estimated change frequency is biased. But at the same time, the crawler should not visit the page too often, because the statistical variation σ/r can be large and the estimated change frequency may be inaccurate.

For example, let us assume that the actual change frequency of the page is, say, once every week ($\lambda = 1/\text{week}$), and the crawler accesses the page once every two weeks ($f = 1/2$ weeks). Then the bias of the estimated change frequency is 57%! (When $r = 2$, $E[\hat{r}]/r \approx 0.43$.) On the other hand, if the crawler revisits the page every day ($f = 1/\text{day}$), then the statistical variation is large and the 95% confidence interval is 1.5 times as large as the actual frequency! (When $r = 1/7$, the 95% confidence interval, $2\sigma/r$ is 1.5.) In the next subsection, we will try to identify the best revisit frequency that estimates the change frequency most accurately. □

6.4.2 Improved estimator: $-\log(\bar{X}/n)$

While the estimator X/T is known to be quite effective when we have a *complete* change history of an element [TK98, Win72], our analysis showed that it is less than desirable when we have an *incomplete* change history. The estimator is biased and we cannot reduce the bias by increasing the sample size. In this subsection, we propose another estimator $-\log(\bar{X}/n)$, which has more desirable properties.

Intuitively, we can derive our new estimator from Lemma 6.1.[1] In the lemma, we computed the probability q that the element does not change at each access: $q = e^{-\lambda/f} = e^{-r}$. By rearranging this equation, we get

$$r = -\log q$$

Note that we can measure the probability q in the above equation by an experiment; For example, if we accessed an element 100 times, and if the element did not change in 70 accesses, we can reasonably infer that the probability q is 70/100. Then from the equation, we can estimate that the frequency ratio $r = -\log(70/100) = 0.36$. Note that this estimated frequency ratio 0.36 is slightly larger than the 0.30 (= 30/100) that the previous X/n estimates. This increase is because our new estimator accounts for the changes that we may have missed between some accesses. Also note that we can estimate the value of q more accurately by increasing the sample size. Therefore, we can estimate r more precisely and *decrease the bias* by increasing the sample size! (We verify this claim later.) This property has a significant implication in practice. If we use the estimator X/n, we can

[1]We can derive the new estimator using *the method of maximum likelihood estimator* [WMS97], but we show a more intuitive derivation here.

reduce the bias only by adjusting the *access frequency* f (or by adjusting r), which might not be possible for certain applications. However, if we use the estimator $-\log q$, we can reduce the bias to the desirable level, simply by increasing the *number of accesses* to the element. For this reason, we believe our new estimator can be useful for a wider range of applications than X/n is.

To define the new estimator more formally, let $\bar{X}$ be the number of accesses where the element did not change ($\bar{X} = n - X$). Then, our new estimator is

$$\hat{\lambda}/f = -\log(\bar{X}/n) \quad \text{or} \quad \hat{r} = -\log(\bar{X}/n)$$

While intuitively attractive, the estimator $-\log(\bar{X}/n)$ has a mathematical singularity. When the element changes in all our accesses (i.e., $\bar{X} = 0$), the estimator produces infinity, because $-\log(0/n) = \infty$. This singularity makes the estimator technically unappealing, because the expected value of the estimator, $E[\hat{r}]$, is now infinity due to this singularity. (In other words, $\hat{r}$ is biased to infinity!)

Intuitively, we can avoid the singularity if we increase $\bar{X}$ slightly when $\bar{X} = 0$, so that the logarithmic function does not get 0 even when $\bar{X} = 0$. In general, we may avoid the singularity if we add small numbers a and b (> 0) to the numerator and the denominator of the estimator, so that the estimator is $-\log(\frac{\bar{X}+a}{n+b})$. Note that when $\bar{X} = 0$, $-\log(\frac{\bar{X}+a}{n+b}) = -\log(\frac{a}{n+b}) \neq \infty$ if $a > 0$.

Then what value should we use for a and b? To answer this question, we use the fact that we want the expected value, $E[\hat{r}]$, to be as close to r as possible. As we will show shortly, the expected value of $\hat{r}$ is

$$E[\hat{r}] = \mathrm{E}\left[-\log\left(\frac{\bar{X}+a}{n+b}\right)\right] = -\sum_{i=0}^{n} \log\left(\frac{i+a}{n+b}\right)\binom{n}{i}(1-e^{-r})^{n-i}(e^{-r})^{i}$$

which can be approximated to

$$E[\hat{r}] \approx \left[-\log\left(\frac{n+a}{n+b}\right)\right] + \left[n\log\left(\frac{n+a}{n-1+b}\right)\right] r + \ldots$$

by Taylor expansion [Tho69]. Note that we can make the above equation to $E[\hat{r}] \approx r + \ldots$, by setting the constant term $-\log(\frac{n+a}{n+b}) = 0$, and the factor of the r term, $n\log(\frac{n+a}{n-1+b}) = 1$.

From the equation $-\log(\frac{n+a}{n+b}) = 0$, we get $a = b$, and from $n\log(\frac{n+a}{n-1+a}) = 1$, we get the

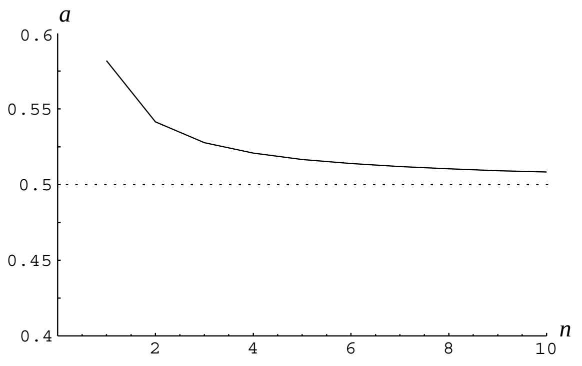


Figure 6.4: The a values which satisfy the equation $n\log(\frac{n+a}{n-1+a}) = 1$

graph of Figure 6.4. In the graph, the horizontal axis shows the value of n and the vertical axis shows the value of a which satisfies the equation $\log(\frac{n+a}{n-1+a}) = 1$ for a given n. We can see that the value of a converges to 0.5 as n increases and that a is close to 0.5 even when n is small. Therefore, we can conclude that we can minimize the bias by setting $a = b = 0.5$.

In summary, we can avoid this singularity by adding a small constant, 0.5, to $\bar{X}$ and n:

$$\hat{r} = -\log\left(\frac{\bar{X}+0.5}{n+0.5}\right)$$

In the rest of this subsection, we will study the properties of this modified estimator $\hat{r} = -\log(\frac{\bar{X}+0.5}{n+0.5})$

1. **Is the estimator unbiased?** To see whether the estimator is biased, let us compute the expected value of $\hat{r}$. From the definition of $\bar{X}$,

   $$\Pr\{\bar{X} = i\} = \Pr\{X = n-i\} = \binom{n}{i}(1-q)^{n-i}q^i$$

   Then,

   $$E[\hat{r}] = \mathrm{E}\left[-\log\left(\frac{\bar{X}+0.5}{n+0.5}\right)\right] = -\sum_{i=0}^{n}\log\left(\frac{i+0.5}{n+0.5}\right)\binom{n}{i}(1-e^{-r})^{n-i}(e^{-r})^i \quad (6.1)$$

   We cannot obtain a closed-form expression in this case. Thus we study its property by numerically evaluating the expression and plotting the results. In Figure 6.5 we show the graph of $E[\hat{r}]/r$ over r for several n values. For comparison, we also show the graph of the previous estimator X/n, in the figure.

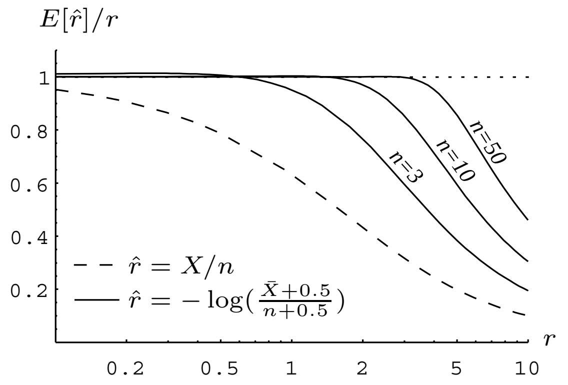


Figure 6.5: Bias of the estimator $-\log(\frac{\bar{X}+0.5}{n+0.5})$

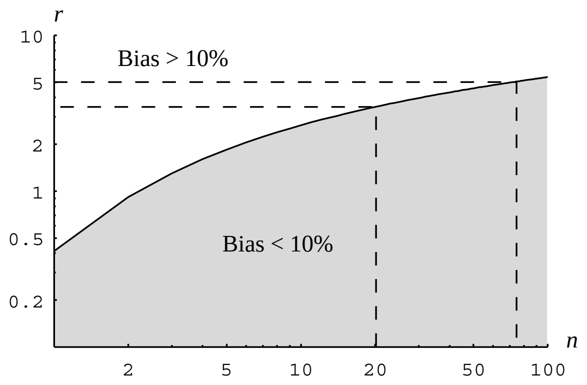


Figure 6.6: The region where the estimator $-\log(\frac{\bar{X}+0.5}{n+0.5})$ is less than 10% biased

From the graph, we can see that our new estimator $-\log(\frac{\bar{X}+0.5}{n+0.5})$ is much better than X/n. While X/n is heavily biased for most r, $-\log(\frac{\bar{X}+0.5}{n+0.5})$ is not heavily biased until $r > 1$ for $n \geq 3$. Clearly, the new estimator becomes less biased as we increase the sample size n. For instance, when $n = 3$, $\hat{r}$ shows bias if $r > 1$, but when $n = 50$, it is not heavily biased until $r > 3$.

Given that the estimator becomes less biased as the sample size grows, we may ask how large the sample size should be in order to get an *unbiased* result. For instance, what sample size gives us less than 10% bias? Mathematically, this can be formulated as follows: Find the region of n and r, where

$$\left| \frac{E[\hat{r}] - r}{r} \right| \leq 0.1$$

is satisfied. From the formula of Equation 6.1, we can numerically compute the region of n and r where the above condition is met, and we show the result in Figure 6.6. In the figure, the gray area is where the bias is less than 10%. Note that the unbiased region grows larger as we increase the sample size n. When $n = 20$, $\hat{r}$ is unbiased when $r < 3.5$, but when $n = 80$, $\hat{r}$ is unbiased when $r < 5$. We illustrate how we can use these graphs to select the revisit frequency when we discuss the efficiency of the new estimator.

2. **How efficient is the estimator?** As we discussed in Section 6.4.1, $\hat{r}$ may take a value other than r even if $E[\hat{r}] \approx r$, and the value of σ/r tells us how large this

statistical variation can be.

To plot the σ/r graph of $-\log(\frac{\bar{X}+0.5}{n+0.5})$, we first compute the variance of $\hat{r}$

$$\begin{aligned}V[\hat{r}] &= E[\hat{r}^2] - E[\hat{r}]^2 \\ &= \sum_{i=0}^{n}\left(\log\left(\frac{i+0.5}{n+0.5}\right)\right)^2\binom{n}{i}(e^{-r})^i(1-e^{-r})^{n-i} \\ &\quad -\left(\sum_{i=0}^{n}\log\left(\frac{i+0.5}{n+0.5}\right)\binom{n}{i}(e^{-r})^i(1-e^{-r})^{n-i}\right)^2\end{aligned} \tag{6.2}$$

From this formula, we can numerically compute $\sigma/r = \sqrt{V[\hat{r}]}/r$ for various r and n values and we show the result in Figure 6.7. As expected, the statistical variation σ/r gets smaller as the sample size n increases. For instance, σ/r is 0.4 for $r = 1.5$ when $n = 10$, but σ/r is 0.2 for the same r value when $n = 40$.

Also note that the statistical variation σ/r takes its minimum at $r \approx 1.5$ within the unbiased region of r. For instance, when $n = 20$, the estimator is practically unbiased when $r < 2$ (the bias is less than 0.1% in this region) and within this range, σ/r is minimum when $r \approx 1.35$. For other values of n, we can similarly see that σ/r takes its minimum when $r \approx 1.5$. We can use this result to decide on the revisit frequency for an element.

Example 6.4 A crawler wants to estimate the change frequency of a Web page by visiting it 10 times. While the crawler does not know exactly how often that particular page changes, say many pages within the same domain are known to change roughly once every week. Based on this information, the crawler wants to decide how often to access that page.

Because the statistical variation (thus the confidence interval) is smallest when $r \approx 1.5$ and because the current guess for the change frequency is once every week, the optimal revisit frequency for that page is 7 days $\times$ 1.5 $\approx$ once every 10 days. Under these parameters, the estimated change frequency is less than 0.3% biased and the estimated frequency may be different from the actual frequency by up to 35% with 75% probability. We believe that this confidence interval will be more than adequate for most crawling and caching applications.

In certain cases, however, the crawler may learn that its initial guess for the change

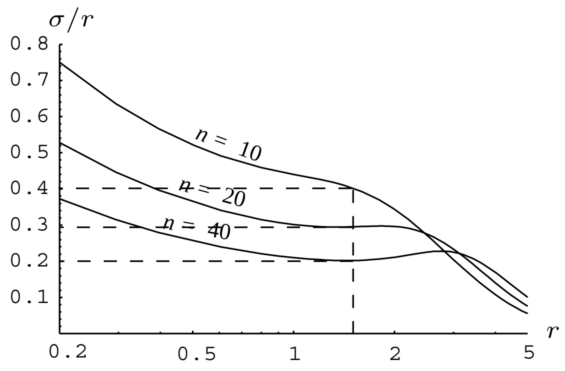


Figure 6.7: The graph of σ/r for the estimator $-\log(\frac{\bar{X}+0.5}{n+0.5})$

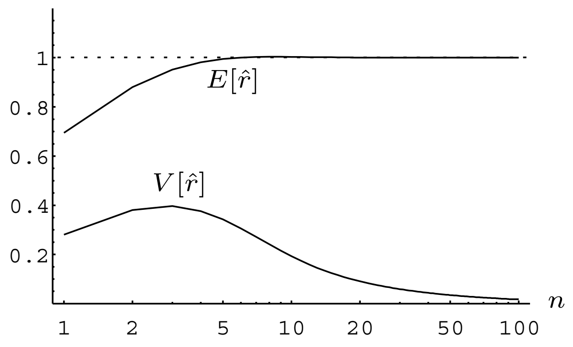


Figure 6.8: The graphs of $E[\hat{r}]$ and $V[\hat{r}]$ over n, when $r = 1$

frequency may be quite different from the actual change frequency, and the crawler may want to adjust the access frequency in the subsequent visits. We briefly discuss this adaptive policy later. □

3. **Is the estimator consistent?** We can prove that the estimator $\hat{r}$ is consistent, by showing that $\lim_{n\to\infty} E[\hat{r}] = r$ and $\lim_{n\to\infty} V[\hat{r}] = 0$ for any r [WMS97]. Although it is not easy to formally prove, we believe our estimator $\hat{r}$ is indeed consistent. In Figure 6.5, $E[\hat{r}]/r$ gets close to 1 as $n \to \infty$ for any r, and in Figure 6.7, σ/r (thus $V[\hat{r}]$) approaches zero as $n \to \infty$ for any r. As an empirical evidence, we show the graphs of $E[\hat{r}]$ and $V[\hat{r}]$ over n when $r = 1$ in Figure 6.8. $E[\hat{r}]$ clearly approaches 1 and $V[\hat{r}]$ approaches zero.

6.4.3 Irregular access interval

When we access an element at irregular intervals, it becomes more complicated to estimate its change frequency. For example, assume that we detected a change when we accessed an element after 1 hour and we detected another change when we accessed the element after 10 hours. While all changes are considered equal when we access an element at regular intervals, in this case the first change "carries more information" than the second, because if the element changes more than once every hour, we will definitely detect a change when we accessed the element after 10 hours.

In order to obtain an estimator for the irregular case, we can use a technique, called *maximum likelihood estimator* [WMS97]. Informally, the maximum likelihood estimator

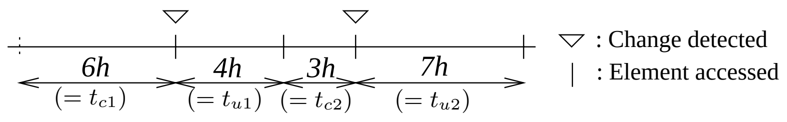


Figure 6.9: An example of irregular accesses

computes which λ value has the highest probability of producing the observed set of events, and use this value as the estimated λ value. Using this method for the irregular access case, we obtain (derivation not given here) the following equation:

$$\sum_{i=1}^{m} \frac{t_{ci}}{e^{\lambda t_{ci}} - 1} = \sum_{j=1}^{n-m} t_{uj} \tag{6.3}$$

Here, t_{ci} represents the interval in which we detected the ith change, and t_{uj} represents the jth interval in which we did not detect a change. Also, m represents the total number of changes we detected, from n accesses. Note that all variables in Equation 6.3 (except λ) can be measured by an experiment. Therefore, we can compute the estimated frequency by solving this equation for λ. Also note that all access intervals, t_{ci}'s and t_{uj}'s, take part in the equation. It is because depending on the access interval, the detected change or non-change carries different levels of information. We illustrate how we can use the above estimator by the following example.

Example 6.5 We accessed an element 4 times in 20 hours (Figure 6.9), in which we detected 2 changes (the first and the third accesses). Therefore, the two changed intervals are $t_{c1} = 6h$, $t_{c2} = 3h$ and the two unchanged intervals are $t_{u1} = 4h$, $t_{u2} = 7h$. Then by solving Equation 6.3 using these numbers, we can estimate that $\lambda = 2.67$ changes/20 hours. Note that the estimated λ is slightly larger than 2 changes/20 hours, which is what we actually observed. This result is because the estimator takes "missed" changes into account. □

In this thesis we do not formally analyze the bias and the efficiency of the above estimator, because the analysis requires additional assumption on how we access the element. However, we believe the proposed estimator is "good" for three reasons:

1. The estimated λ has the highest probability of generating the observed changes.

2. When the access to the element follows a Poisson process, the estimator is consistent. That is, as we access the element more, the estimated λ converges to the actual λ.

3. When the access interval is always the same, the estimator reduces to the one in Section 6.4.2.

6.5 Estimation of frequency: last date of change

When the last-modification date of an element is available, how can we use it to estimate change frequency? For example, assume that a page changed 10 hours before our first access and 20 hours before our second access. Then what will be a fair guess for its change frequency? Would it be once every 15 hours? Note that in this scenario we cannot apply standard statistical techniques (e.g., the maximal-likelihood estimator), because the observed last-modified dates might be correlated: If the page did not change between two accesses, the last-modification date in the first access would be the same as the modification date in the second access. In this section, we propose a new estimator which can use the last-modified date for frequency estimation.

6.5.1 Initial estimator

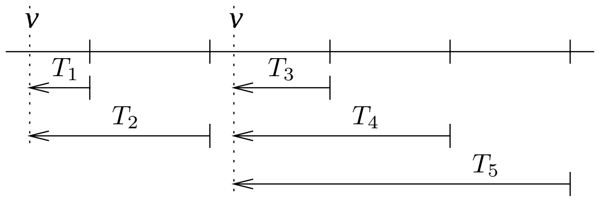
The final estimator that we propose is relatively complex, so we obtain the estimator step by step, instead of presenting the estimator in its final form.

We can derive the initial version of our estimator based on the following well-known lemma [WMS97]:

Lemma 6.2 *Let T be the time to the previous event in a Poisson process with rate λ. Then the expected value of T is $E[T] = 1/\lambda$* □

That is, in a Poisson process the expected time to the last change is $1/\lambda$. Therefore, if we define T_i as the time from the last change at the ith access, $E[T_i]$ is equal to $1/\lambda$. When we accessed the element n times, the sum of all T_i's, $\mathsf{T} = \sum_{i=1}^{n} T_i$, is $E[\mathsf{T}] = \sum_{i=1}^{n} E[T_i] = n/\lambda$. From this equation, we suspect that if we use n/T as our estimator, we may get an unbiased estimator $E[n/\mathsf{T}] = \lambda$. Note that T in this equation is a number that needs to be measured by repeated accesses.

While intuitively appealing, this estimator has a serious problem because the element may not change between some accesses. In Figure 6.10, for example, the element is accessed 5 times but it changed only twice. If we apply the above estimator naively to this example, n will be 5 and T will be $T_1 + \cdots + T_5$. Therefore, this naive estimator practically considers that



v : The element changes
| : The element is accessed

Figure 6.10: Problems with the estimator based on last modified date

```
Init()
   N = 0;  /* number of accesses */
   X = 0;  /* number of changes detected */
   T = 0;  /* sum of time to change */

Update(Ti, Ii)
   N = N + 1;
   /* Has the element changed? */
   If (Ti < Ii) then
      /* The element has changed. */
      X = X + 1;
      T = T + Ti;
   else
      /* The element has not changed */
      T = T + Ii;

Estimate()
   return X/T;
```

Figure 6.11: The estimator using last-modified dates

the element changed 5 times with the last modified dates of $T_1, T_2, \ldots, T_5$. This estimation clearly does not match with the actual changes of the element, and thus leads to bias.[2] Intuitively, we may get a better result if we divide the actual number of changes, 2, by the sum of T_2 and T_5, the final last-modified dates for the two changes. Based on this intuition, we modify the naive estimator to the one shown in Figure 6.11.

The new estimator consists of three functions, `Init()`, `Update()` and `Estimate()`, and it maintains three global variables `N`, `X`, and `T`. Informally, `N` represents the number of accesses to the element, `X` represents the number of detected changes, and `T` represents the sum of

[2]We can verify the bias by computing $E[n/T]$ when $\lambda \ll f$.

the time to the previous change at each access. (We do not use the variable `N` in the current version of the estimator, but we will need it later.) Initially, the `Init()` function is called to set all variables to zero. Then whenever the element is accessed, the `Update()` function is called, which increases `N` by one and updates `X` and `T` values based on the detected change. The argument `Ti` to `Update()` is the time to the previous change in the ith access and the argument `Ii` is the interval between the accesses. If the element has changed between the $(i-1)$th access and the ith access, `Ti` will be smaller than the access interval `Ii`. Note that the `Update()` function increases `X` by one, only when the element has changed (i.e., when `Ti` $<$ `Ii`). Also note that the function increases `T` by `Ii`, not by `Ti`, when the element has not changed. By updating `X` and `T` in this way, this algorithm implements the estimator that we intend. Also note that the estimator of Figure 6.11 predicts the change frequency λ directly. In contrast, the estimator of Section 6.4 predicts the change frequency by estimating the frequency ratio r.

6.5.2 Bias-reduced estimator

In this section, we analyze the bias of the estimator described in Figure 6.11. This analysis will show that the estimator has significant bias when `N` is small. By studying this bias carefully, we will then propose an improved version of the estimator that practically eliminates the bias.

For our analysis, we assume that we access the element at a regular interval $I(= 1/f)$, and we compute the bias of the *frequency ratio* $\hat{r} = \hat{\lambda}/f$, where $\hat{\lambda}$ is the estimator described in Figure 6.11. This assumption makes our the analysis manageable and it also reveals the the core problem of the estimator. While the *analysis* is based on regular access cases, we emphasize that we can still use our final estimator when access is irregular.

The following lemma gives us the basic formula for the analysis of the bias.

Lemma 6.3 *The bias of the estimator $\hat{r} = \hat{\lambda}/f$ ($\hat{\lambda}$ is the estimator of Figure 6.11) is:*

$$\frac{E[\hat{r}]}{r} = \sum_{k=0}^{n} \left[\frac{\Pr\{X = k\}}{r} \int_{(n-k)I}^{nI} \left(\frac{k}{ft}\right) \Pr\{T = t \mid X = k\}\, dt \right] \tag{6.4}$$

Here, $\Pr\{X = k\}$ *is the probability that the variable* `X` *takes a value k, and* $\Pr\{T = t \mid X = k\}$ *is the probability that the variable* `T` *takes a value t when* `X` $= k$. *We assume we access the element n times.* □

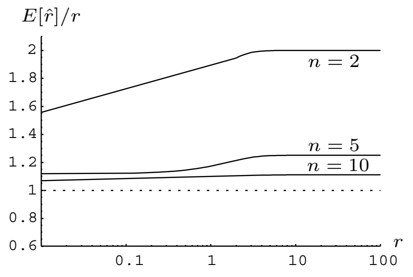


Figure 6.12: Bias of the estimator in Figure 6.11

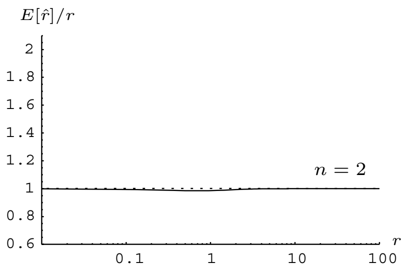


Figure 6.13: Bias of the estimator with the new `Estimate()` function

Proof The proof is straightforward. We can compute the expected value by summing all possible $\frac{X}{T}$ values multiplied by their probabilities. In the proof of Theorem 6.1, we will show how we can compute $\Pr\{X = k\}$ and $\Pr\{T = t \mid X = k\}$. ■

For our later discussion, it is important to understand that each term in the summation of Equation 6.4 corresponds to the case when `X` $= k$. Since `X` may take $0, 1, \ldots, n$, we are considering all possible `X` values by varying k from 0 to n.

In Figure 6.12, we show the bias of the estimator by plotting the graph of $E[\hat{r}]/r$. We computed this graph analytically[3] and we verified its correctness by simulations. Remember that $E[\hat{r}] = r$ (or $E[\hat{r}]/r = 1$) when the estimator is unbiased (the dotted line). The solid line shows the actual graphs of the estimator. From the graph, we can see that the estimator has significant bias when n is small, while the bias decreases as n increases. For instance, when $n = 2$, the estimated r is *twice* as large as the actual r ($E[\hat{r}]/r \approx 2$) for large r values.

To study the bias more carefully, we compute the closed form of the bias in the following theorem. This theorem will give us an important idea on how we can eliminate the bias.

Theorem 6.1 *When $r \to \infty$, the only remaining term in the summation of Equation 6.4 is when $k = n$, and $E[\hat{r}]/r$ becomes $\frac{n}{n-1}$.*

$$\lim_{r\to\infty} \frac{E[\hat{r}]}{r} = \lim_{r\to\infty} \frac{\Pr\{X = n\}}{r} \int_0^{nI} \left(\frac{n}{ft}\right) \Pr\{T = t \mid X = n\}\, dt = \frac{n}{n-1}$$

Also, when $r \to 0$, the only remaining term in the equation is when $k = 1$, and $E[\hat{r}]/r$

[3]We used the analytical forms of $\Pr\{X = k\}$ and $\Pr\{T = t \mid X = k\}$, which are computed in the proof of Theorem 6.1.

becomes $n \log(\frac{n}{n-1})$.

$$\lim_{r\to 0} \frac{E[\hat{r}]}{r} = \lim_{r\to 0} \frac{\Pr\{X=1\}}{r} \int_{(n-1)I}^{nI} \left(\frac{1}{ft}\right) \Pr\{\mathcal{T}=t \mid X=1\}\, dt = n \log\left(\frac{n}{n-1}\right) \qquad \square$$

Proof We first show how we can compute $\Pr\{X = k\}$ and $\Pr\{\mathcal{T} = t \mid X = k\}$. The variable X is equal to k when the element changed in k accesses. Since the element may change at each access with probability $1 - e^{-r}$ ($r = \lambda/f$, r: frequency ratio),

$$\Pr\{X = k\} = \binom{n}{k}(1-e^{-r})^k(e^{-r})^{n-k}$$

Now we compute the probability that $\mathcal{T} = t$ ($0 \le t \le kI$) when $X = k$. If we use t_i to represent the *time added to* $\mathcal{T}$ *at the ith access*, $\mathcal{T}$ can be expressed as

$$\mathcal{T} = \sum_{i=1}^{n} t_i$$

Since the element did not change in $(n-k)$ accesses when $X = k$, we added $(n-k)$ times of I to $\mathcal{T}$. Then

$$\mathcal{T} = (n-k)I + \sum_{i=1}^{k} t_{c_i}$$

where c_i is the ith access in which the element changed. Without losing generality, we can assume that the element changed only in the first k accesses. Then

$$\mathcal{T} = (n-k)I + \sum_{i=1}^{k} t_i \tag{6.5}$$

In those changed accesses, the probability that $t_i = t$ is

$$\Pr\{t_i = t\} = \begin{cases} \dfrac{\lambda e^{-\lambda t}}{1-e^{-r}} & \text{if } 0 \le t \le I \\ 0 & \text{otherwise} \end{cases} \tag{6.6}$$

because the element follows a Poisson process. From Equations 6.5 and 6.6, we can compute the $\Pr\{\mathcal{T} = t \mid X = k\}$ by repeatedly applying the *methods of transformations* [WMS97].

For example, when $k = 3$

$$\Pr\{\mathcal{T} = (n-k)I + t \mid X = k\} = \begin{cases} \dfrac{(\lambda t)^2 \lambda e^{-\lambda t}}{2(1-e^{-r})^3} & \text{for } 0 \le t < I \\ \dfrac{[6\lambda t - 2(\lambda t)^2 - 3]\ \lambda e^{-\lambda t}}{2(1-e^{-r})^3} & \text{for } I \le t < 2I \\ \dfrac{(\lambda t - 3)^2 \lambda e^{-\lambda t}}{2(1-e^{-r})^3} & \text{for } 2I \le t < 3I \\ 0 & \text{otherwise} \end{cases}$$

Now we study the limit values of Equation 6.4. When $r \to \infty$,

$$\lim_{r\to\infty} \frac{\Pr\{X = k\}}{r} = \begin{cases} 0 & \text{if} \quad k = 0, 1, \ldots, n-1 \\ 1 & \text{if} \quad k = n \end{cases}$$

Also when $r \to 0$,

$$\lim_{r\to 0} \frac{\Pr\{X = k\}}{r} = \begin{cases} n & \text{if} \quad k = 1 \\ 0 & \text{if} \quad k = 2, \ldots, n \end{cases}$$

Then

$$\begin{aligned} \lim_{r\to\infty} \frac{E[\hat{r}]}{r} &= \lim_{r\to\infty} \sum_{k=0}^{n} \left(\frac{\Pr\{X = k\}}{r} \int_{(n-k)I}^{nI} \left(\frac{k}{ft}\right) \Pr\{\mathcal{T} = t \mid X = k\}\, dt \right) \\ &= \lim_{r\to\infty} \int_0^{nI} \left(\frac{n}{ft}\right) \Pr\{\mathcal{T} = t \mid X = n\}\, dt \\ &= \lim_{r\to\infty} \int_0^{nI} \left(\frac{n}{ft}\right) \frac{(\lambda t)^{n-1} \lambda e^{-\lambda t}}{(n-1)!(1-e^{-r})^n}\, dt \\ &= \frac{n}{n-1} \end{aligned}$$

$$\begin{aligned} \lim_{r\to 0} \frac{E[\hat{r}]}{r} &= \lim_{r\to 0} \sum_{k=0}^{n} \left(\frac{\Pr\{X = k\}}{r} \int_{(n-k)I}^{nI} \left(\frac{k}{ft}\right) \Pr\{\mathcal{T} = t \mid X = k\}\, dt \right) \\ &= n \lim_{r\to 0} \int_{(n-1)I}^{nI} \left(\frac{1}{ft}\right) \Pr\{\mathcal{T} = t \mid X = 1\}\, dt \\ &= n \lim_{r\to 0} \int_{(n-1)I}^{nI} \left(\frac{1}{ft}\right) \frac{e^{(n-1)I} \lambda e^{-\lambda t}}{1 - e^{-r}}\, dt \\ &= n \log\left(\frac{n}{n-1}\right) \end{aligned}$$

■

```
Estimate()
   X' = (X-1) - X/(N*log(1-X/N));
   return X'/T;
```

Figure 6.14: New `Estimate()` function that reduces the bias

Informally, we may explain the meaning of the theorem as follows:

> When r is very large (i.e., when the element changes much more often than we access it), X will be n with high probability, and the bias of the estimator is $\frac{n}{n-1}$. When r is very small (i.e., when we access the element much more often than it changes), X will be either 0 or 1 with high probability, and the bias is $n\log(n/(n-1))$ when $X = 1$.

We can use this result to design an estimator that eliminates the bias. Assume that $X = n$ after n accesses. Then it strongly indicates that r is very large, in which case the bias is $\frac{n}{n-1}$. To avoid this bias, we may divide the original estimator X/T by $\frac{n}{n-1}$ and use $\frac{X}{T}/\frac{n}{n-1} = \frac{n-1}{T}$ as our new estimator in this case. That is, when $X = n$ we may want to use $(X-1)/T$ as our estimator, instead of X/T. Also, assume that $X = 1$ after n accesses. Then it strongly indicates that r is very small, in which case the bias is $n\log(n/(n-1))$. To avoid this bias, we may use $\left(\frac{X}{T}\right) / \; n\log(n/(n-1)) = \frac{1}{T}\frac{X}{n\log(n/(n-X))}$ as our estimator when $X = 1$. In general, if we use the estimator X'/T where

$$X' = (X-1) - \frac{X}{n\log(1 - X/n)}$$

we can avoid the bias both when $X = n$ and $X = 1$: $X' = n - 1$ when $X = n$, and $X' = \frac{1}{n\log(n/(n-1))}$ when $X = 1$.[4]

In Figure 6.14, we show a new `Estimate()` function that incorporates this idea. The new function first computes X' and uses this value to predict λ.

To show that our new estimator is practically unbiased, we plot the graph of $E[\hat{r}]/r$ for the new `Estimate()` function in Figure 6.13. The axes in the graph are the same as in Figure 6.12. Clearly, the estimator is practically unbiased. Even when $n = 2$, $E[\hat{r}]/r$ is very close to 1 (the bias is less than 2% for any r value.). We show the graph only for $n = 2$,

[4]The function $(X-1) - X/(n\log(1 - X/n))$ is not defined when $X = 0$ and $X = n$. However, we can use $\lim_{X\to 0}[(X-1) - X/(n\log(1-X/n))] = 0$ and $\lim_{X\to n}[(X-1) - X/(n\log(1-X/n))] = n-1$ as its value when $X = 0$ and $X = n$, respectively. That is, we assume $X' = 0$ when $X = 0$, and $X' = n-1$ when $X = n$.

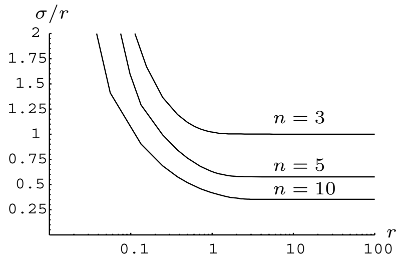


Figure 6.15: Statistical variation of the new estimator over r

because the graphs for other n values essentially overlap with that of $n = 2$.

While we derived the new `Estimate()` based on the analysis of regular access cases, note that the new `Estimate()` function does not require that access be regular. In fact, through multiple simulations, we have experimentally verified that the new function still gives negligible bias even when access is irregular. We illustrate the usage of this new estimator through the following example.

Example 6.6 A crawler wants to estimate the change frequency of a page by visiting it 5 times. However, the crawler cannot access the page more than once every month, because the site administrator does not allow more frequent crawls. Fortunately, the site provides the last modified date whenever the crawler accesses the page.

To show the improvement, let us assume that the page changes, say, once every week and we crawl the page once every month. Then, without the last modified date, the bias is 43% on average ($E[\hat{r}]/r \approx 0.57$), while we can practically eliminate the bias when we use the last modified date. (The bias is less than 0.1%.) $\square$

Finally in Figure 6.15, we show the statistical variation σ/r of the new estimator, for various n. The horizontal axis in the graph is the frequency ratio r, and the vertical axis is the statistical variation σ/r. We can see that as n increases, the variation (or the standard deviation) gets smaller.

6.6 Categorization of frequency: Bayesian inference

We have studied how to *estimate* the change frequency given its change history. But for certain applications, we may only want to categorize elements into several classes based on their change frequencies.

Example 6.7 A crawler completely recrawls the web once every month and partially updates a small subset of the pages once every week. Therefore, the crawler does not particularly care whether an element changes every week or every 10 days, but it is mainly interested in whether it needs to crawl a page either every week or every month. That is, it only wants to classify pages into two categories based on their change history. □

For this example, we may still use the estimators of previous sections and classify pages by some threshold frequency. For example, we may classify a page into the every month category if its *estimated* frequency is lower than once every 15 days, and otherwise categorize the page into the every week category. In this section, however, we will study an alternative approach, which is based on the Bayesian decision theory. While the machinery that we use in this section has been long used in the statistics community, it has not been applied to the *incomplete* change history case. After a brief description of the estimator, we will study the effectiveness of this method and the implications when the change histories are incomplete.

To help our discussion, let us assume that we want to categorize a web page (p_1) into two classes, the pages that change every week (C_W) and the pages that change every month (C_M). To trace which category p_1 belongs to, we maintain two probabilities $P\{p_1 \in C_W\}$ (the probability that p_1 belongs to C_W) and $P\{p_1 \in C_M\}$ (the probability that p_1 belongs to C_M). As we access p_1 and detect changes, we update these two probabilities based on the detected changes. Then at each point of time, if $P\{p_1 \in C_W\} > P\{p_1 \in C_M\}$, we consider p_1 belongs to C_W, and otherwise we consider p_1 belongs to C_M. (While we use two categories in our discussion, the technique can be generalized to more than two categories.)

Initially we do not have any information on how often p_1 changes, so we start with fair values $P\{p_1 \in C_W\} = 0.5$ and $P\{p_1 \in C_M\} = 0.5$. Now let us assume we first accessed p_1 after 5 days and we learned that p_1 had changed. Then how should we update $P\{p_1 \in C_W\}$ and $P\{p_1 \in C_M\}$? Intuitively, we need to increase $P\{p_1 \in C_W\}$ and decrease $P\{p_1 \in C_M\}$, because p_1 had changed in less than a week. But how much should we increase $P\{p_1 \in C_W\}$? We can use the Bayesian theorem to answer this question. Mathematically, we want to reevaluate $P\{p_1 \in C_W\}$ and $P\{p_1 \in C_M\}$ given the event E, where E represents the change of p_1. That is, we want to compute $P\{p_1 \in C_W \mid E\}$ and $P\{p_1 \in C_M \mid E\}$. According to the Bayesian theorem,

$$\begin{aligned} P\{p_1 \in C_W \,|\, E\} &= \frac{P\{(p_1 \in C_W) \wedge E\}}{P\{E\}} = \frac{P\{(p_1 \in C_W) \wedge E\}}{P\{E \wedge (p_1 \in C_W)\} + P\{E \wedge (p_1 \in C_M)\}} \\ &= \frac{P\{E \,|\, p_1 \in C_W\} P\{p_1 \in C_W\}}{P\{E \,|\, p_1 \in C_W\} P\{p_1 \in C_W\} + P\{E \,|\, p_1 \in C_M\} P\{p_1 \in C_M\}} \end{aligned} \tag{6.7}$$

In the equation, we can compute $P\{E \,|\, p_1 \in C_W\}$ (the probability that p_1 changes in 5 days, when its change frequency is a week) and $P\{E \,|\, p_1 \in C_M\}$ (the probability that p_1 changes in 5 days, when its change frequency is a month) based on the Poisson process assumption. Also we previously assumed that $P\{p_1 \in C_W\} = P\{p_1 \in C_M\} = 0.5$. Then,

$$\begin{aligned} P\{p_1 \in C_W \,|\, E\} &= \frac{(1 - e^{-5/7})0.5}{(1 - e^{-5/7})0.5 + (1 - e^{-5/30})0.5} \approx 0.77 \\ P\{p_1 \in C_M \,|\, E\} &= \frac{(1 - e^{-5/30})0.5}{(1 - e^{-5/7})0.5 + (1 - e^{-5/30})0.5} \approx 0.23 \end{aligned}$$

That is, p_1 now belongs to C_W with probability 0.77 and p_1 belongs to C_M with probability 0.23! Note that these new probabilities, 0.77 and 0.23, coincide with our intuition. $P\{p_1 \in C_W\}$ has indeed increased and $P\{p_1 \in C_M\}$ has indeed decreased.

For the next access, we can repeat the above process. If we detect another change after 5 days, we can update $P\{p_1 \in C_W \,|\, E\}$ and $P\{p_1 \in C_M \,|\, E\}$ by using Equation 6.7, but now with $P\{p_1 \in C_W\} = 0.77$ and $P\{p_1 \in C_M\} = 0.23$. After this step, $P\{p_1 \in C_W\}$ increases to 0.92 and $P\{p_1 \in C_M\}$ becomes 0.08.

Note that we do not need to set an arbitrary threshold to categorize elements under this estimator. If we want to use the previous estimators in Section 6.4, we need to set a threshold to classify pages, which can be quite arbitrary. By using the Bayesian estimator, we can avoid setting this arbitrary threshold, because the estimator itself naturally classifies pages.

In Figure 6.16 we show how accurate the Bayesian estimator is. In the graph, we show the probability that a page is classified into C_M when its change frequency is λ (the horizontal axis) for various n values. We obtained the graph analytically assuming that we access the page every 10 days. From the graph we can see that the estimator classifies the page quite accurately. For instance, when $\lambda \leq \frac{1}{\text{month}}$ the estimator places the page in C_M with more than 80% probability for $n = 3$. Also, when $\lambda \geq \frac{1}{\text{week}}$ it places the page in C_W with more than 80% probability for $n = 3$ ($P\{p_1 \in C_M\} < 0.2$, so $P\{p_1 \in C_W\} > 0.8$).

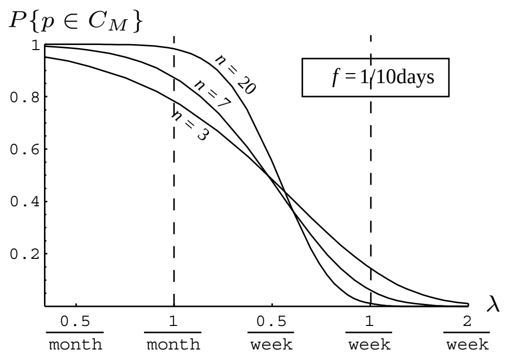


Figure 6.16: The accuracy of the Bayesian estimator

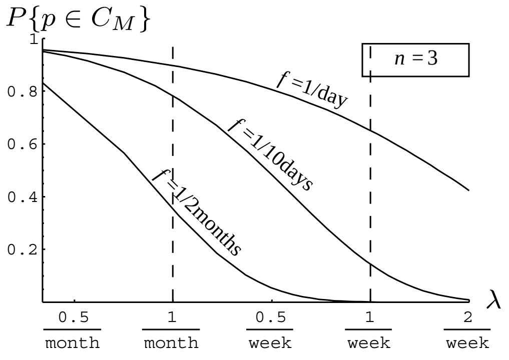


Figure 6.17: The accuracy of the Bayesian estimator for various access frequencies

Clearly the estimator categorizes the page more accurately as the sample size increases. When $n = 10$, the estimator categorizes the page correctly with more than 95% probability.

While the Bayesian estimator is quite accurate and can handle the irregular access case, we can get a more accurate result by carefully deciding on how often we access the page. To illustrate this issue, we show in Figure 6.17 the accuracy graph for various access frequencies. From the graph, we can see that the estimator is much more accurate when the access frequency lies between categorization frequencies, $\frac{1}{\text{month}}$ and $\frac{1}{\text{week}}$ ($f = 1/10$ days), than when it lies outside of this range ($f = 1/\text{day}$ or $f = 1/2$ months). For example, when $f = 1/\text{day}$, the estimator places the page in C_M with high probability even when $\lambda > \frac{1}{\text{week}}$. Intuitively, we can explain this phenomenon as follows: When we access the page much more often than it changes, we do not detect any change to the page at all in the first several accesses. Then, since the page is less likely to change when $p_1 \in C_M$ than when $p_1 \in C_W$, the Bayesian estimator increases $P\{p_1 \in C_M\}$ and decreases $P\{p_1 \in C_W\}$, although this increase/decrease is small. Therefore, the estimator tends to place the page in C_M until n becomes large. While this bias disappears after we access the page many times (more specifically, when $n \approx f/\lambda$ or larger), we can avoid this bias in the first place by selecting an appropriate revisit frequency if we can.

6.7 Dynamic estimation

So far, we have assumed that we pick an estimator *before* we start an experiment and use the method throughout the experiment. But in certain cases, we may need to dynamically adjust our estimation method, depending on what we detect during the experiment. In this section, we briefly discuss when we may need this dynamic estimation technique and what we can do in that situation.

1. **Adaptive scheme:** Even if we initially decide on a certain access frequency, we may want to adjust it during the experiment, when the estimated change frequency is very different from our initial guess. Then exactly when and how much should we adjust the access frequency?

   Example 6.8 Initially, we guessed that a page changes once every week and started visiting the page every 10 days. In the first 4 accesses, however, we detected 4 changes, which signals that the page may change much more frequently than we initially guessed.

   In this scenario, should we increase the access frequency immediately or should we wait a bit longer until we collect more evidence? When we access the page less often than it changes, we need a large sample size to get an unbiased result, so it might be good to adjust the access frequency immediately. On the other hand, it is also possible that the page indeed changes once every week on average, but it changed in the first 4 accesses by pure luck. Then when should we adjust the change frequency to get the optimal result?

   Note that dynamic adjustment is not a big issue when the last modified date is available. In Section 6.5, we showed that the bias is practically negligible independent of the access frequency (Figure 6.13) and that the statistical variation gets smaller as we access the page less frequently (Figure 6.15). Therefore, it is always good to access the page as slowly as we can. In this case, the only constraint will be how early we need to estimate the change frequency. □

   In signal processing, similar problems have been identified and carefully studied [OS75, Mah89, TL98]. For example, when we want to reconstruct a signal, the signal should be sampled at a certain frequency, while the optimal sampling frequency depends on the frequency of the signal. Since the frequency of the signal is *unknown* before it is sampled, we need to adjust the sampling frequency based on the previous sampling result. We may use the mathematical tools and principles developed in this context to address our problem.

2. **Changing λ:** Throughout this chapter, we assumed that the change frequency λ of an element is stationary (i.e., does not change). This assumption may not be valid in certain cases, and we may need to test whether λ changes or not.

   If λ changes very slowly, we may be able to detect the change of λ and use the estimated λ to improve, say, Web crawling. For example, if λ changes once every month, we may estimate λ in the first few days of every month and use the estimated λ for the rest of the month. Then by comparing the λ's for each month, we may also compute how much λ increases/decreases every month. However, when λ changes very rapidly, it will be difficult and impractical to estimate λ and use the estimated λ to improve other applications.

   There has been work on how we can verify whether a Poisson process is stationary and, if not, how we can estimate the change rate of λ [YC96, Can72]. While this work assumes that the complete change history of an element is available, we believe the basic principles still can be used for our scenario.

6.8 Experiments

In this thesis, we have mainly assumed that an element changes by a Poisson process with a change rate λ. In this section we present additional experiments confirming that our estimator is better than the naive one even when page changes do not follow a Poisson process (Section 6.8.1). Also, we show that our estimator works well on real Web data (Section 6.8.2) and can improve the effectiveness of a Web crawler very significantly (Section 6.8.3).

6.8.1 Non-Poisson model

Although our results in Chapter 4 strongly suggest a Poisson change distribution (that we have used so far), it is interesting to study what happens to our estimators if the distribution were "not quite Poisson." One simple way to model deviations from Poisson is to change the exponential inter-arrival distribution to a more general *gamma distribution*.

A gamma distribution has two parameters, α and λ. When $\alpha = 1$, the gamma distribution becomes an exponential distribution with change rate λ, and we have a Poisson change process. If $\alpha < 1$, small change intervals become more frequent. As α grows beyond 1, small intervals become rare, and the distribution starts approximating a normal one. Thus, we can capture a wide range of behaviors by varying the α around its initial value of 1.

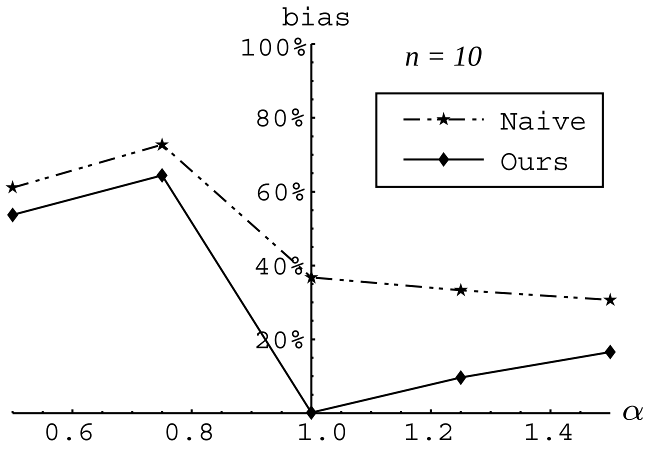


Figure 6.18: Bias of the naive and our estimators for a gamma distribution

Through simulations, we experimentally measured the biases of the naive estimator (Section 6.4.1) and our proposed estimator (Section 6.4.2), and we show the result in Figure 6.18. This simulation was done on synthetic data, because the real Web data mostly followed an exponential distribution. Also, we let the element change once every second on average, and we accessed the element every second. This simulation was conducted for various α values (the horizontal axis).

From the graph, we can see that the bias of our new estimator is consistently smaller than that of the naive estimator. When $\alpha = 1$, or when the element follows a Poisson process, the bias of our estimator is 0, while the bias of the naive estimator is around 37%. Even when α is not equal to 1, our estimator still has smaller bias than that of the naive estimator. This graph suggests that our estimator is significantly better than the naive one, even if the change distribution is not quite a Poisson distribution.

6.8.2 Effectiveness of estimators for real Web data

In this subsection, we compare the effectiveness of our proposed estimator (Section 6.4.2) and that of the naive estimator (Section 6.4.1), using the real Web data described in Chapter 4.

To compare their effectiveness, we need the *actual* change frequency of a page, so that we can see how far apart each estimator is from this actual value. Clearly, it is not possible to obtain the *actual* change frequency, because we do not know exactly how many times each page changed (i.e., we may have missed some changes). To address this problem, we used the following method: First, we identified the pages for which we monitored "most" of the changes. That is, we selected only the pages that changed *less than once in three days*, because we would probably have missed many changes, if a page changed more often. Also, we filtered out the pages that changed *less than 3 times* during our monitoring period, because we may have not monitored the page long enough, if it changed less often.[5] Then for the selected pages, we assumed that we did *not* miss any of their changes and thus we can estimate their actual frequencies by X/T (X: number of changes detected, T: monitoring period). We refer to this value as a *projected change frequency.* After this selection, we ran a *simulated* crawler on those pages which visited each page only *once every week.* Therefore, the crawler had *less* change information than our original dataset. Based on this limited information, the crawler estimated change frequency and we compared the estimates to the projected change frequency.

We emphasize that the simulated crawler did not actually crawl pages. Instead, the simulated crawler was run on the change data collected for Chapter 4, so the projected change frequency and the crawler's estimated change frequency are based on the same dataset (The crawler simply had less information than our dataset). Therefore, we believe that an estimator is better when it is closer to the projected change frequency.

From this comparison, we could observe the following:

- For 83% of pages, our proposed estimator is closer to the projected change frequency than the naive one. The naive estimator was "better" for less than 17% pages.
- Assuming that the projected change frequency is the actual change frequency, our estimator showed about 15% bias on average over all pages, while the naive estimator showed more than 35% bias. Clearly, this result shows that our proposed estimator is significantly more effective than the naive one. We can decrease the bias by one half, if we use our estimator!

In Figure 6.19, we show more detailed results from this experiment. The horizontal axis in the graph shows the ratio of the *estimated* change frequency to the *projected* change frequency (r_λ) and the vertical axis shows the fraction of pages that had the given ratio. Therefore, for the pages with $r_\lambda < 1$, the estimate was smaller than the projected frequency, and for the pages with $r_\lambda > 1$, the estimate was larger than the projected frequency. Assuming that the projected frequency is the actual frequency, a better estimator is the

[5]We also used less (and more) stringent range for the selection, and the results were similar.

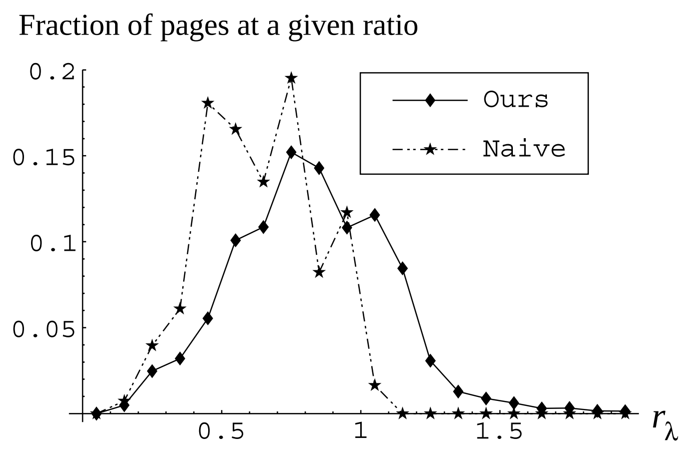


Figure 6.19: Comparison of the naive estimator and ours

one centered around $r_\lambda = 1$. From the graph, we can clearly see that our proposed estimator is better than the naive one. The distribution of the naive estimator is skewed to the left quite significantly, while the distribution of our estimator is more centered around 1.

6.8.3 Application to a Web crawler

Our results show that our proposed estimator gives a more accurate change frequency than the naive one. Now we study how much improvement an application may achieve by using our estimator. To illustrate this point, we consider a Web crawler as an example.

A typical Web crawler revisits all pages at the *same* frequency, regardless of how often they change. Instead, a crawler may revisit pages at different frequencies depending on their estimated change frequency. In this subsection, we compare the following three revisit policies:

- **Uniform policy:** A crawler revisits all pages at the frequency of once every week. The crawler does not try to estimate how often a page changes.
- **Naive policy:** In the first 5 visits, a crawler visits each page at the frequency of once every week. After the 5 visits, the crawler estimates change frequencies of pages using the *naive estimator* (Section 6.4.1). Based on these estimates, the crawler adjusts revisit frequencies for the remaining visits.
- **Our policy:** The crawler uses our proposed estimator (Section 6.4.2) to estimate change frequency. Other than this fact, the crawler uses the same policy as the naive

policy	*Changes detected*	*Percentage*
Uniform	$2,147,589$	100%
Naive	$4,145,581$	193%
Ours	$4,892,116$	228%

Table 6.1: Total number of changes detected for each policy

policy.

Using each policy, we ran a *simulated* crawler on the change history data described in Chapter 4. In the experiments, the crawler adjusted its revisit frequencies (for the naive and our policies), so that the *average* revisit frequency *over all pages* was equal to once a week under any policy. That is, the crawler used the same total download/revisit resources, but allocated these resources differently under different policies. Since we have the change history of 720,000 pages for about 3 months,[6] and since the simulated crawler visited pages once every week on average, the crawler visited pages $720,000 \times 13$ weeks $\approx 9,260,000$ times in total.

Out of these 9.2M visits, we counted how many times the crawler detected changes, and we report the results in Figure 6.1. The second column shows the total number of changes detected under each policy, and the third column shows the percentage improvement over the uniform policy. Note that the best policy is the one that detected the highest number of changes, because the crawler visited pages the same number of times in total. That is, we use the *total number of changes detected* as our quality metric. From these results, we can observe the following:

- A crawler can significantly improve its effectiveness by adjusting its revisit frequency. For example, the crawler detected 2 times more changes when it used the naive policy than the uniform policy.

- Our proposed estimator makes the crawler much more effective than the naive one. Compared to the naive policy, our policy detected 35% more changes!

[6]While we monitored pages for 4 months, some pages were deleted during our experiment, so each page was monitored for 3 months on average

6.9 Related work

The problem of estimating change frequency has long been studied in the statistics community [TK98, Win72, MS75, Can72]. However, most of the existing work assumes that the complete change history is known, which is not true in many practical scenarios.

Researchers have experimentally estimated the change frequency of pages [WVS$^+$99, DFK99, WM99]. Note that most of the work used a naive estimator, which is significantly worse than the estimators that we propose in this chapter. We believe their work can substantially benefit by using our estimators.

One notable exception is [BC00], which uses last-modified dates to estimate the *distribution* of change frequencies over a *set* of pages. However, since their analysis predicts the *distribution* of change frequencies, not the change frequency of an *individual* page, the method is not appropriate for the scenarios in this thesis.

Some of the tools that we used in this chapter (e.g., the maximum likelihood estimator in Section 6.4.3 and the Bayesian inference method in Section 6.6) have already been developed by the statistics community. However, in certain cases, straightforward application of standard statistical techniques results in estimators with undesirable properties (e.g., highly-biased estimators of Sections 6.4.1 and 6.5.1). By analyzing these estimators carefully, we designed estimators with much less bias and with high consistency.

6.10 Conclusion

In this chapter, we studied the problem of estimating the change frequency of a Web page when a crawler does *not* have the complete change history of the page. We proposed new estimators that compute the change frequency reasonably well even with the incomplete change history. Also, we analytically showed how effective the proposed estimators are, discussing the practical implications of the various choices. In most cases, our new estimators are significantly better than existing ones and can estimate change frequency much more accurately. Clearly, accurate estimates can lead to better decisions for many applications, including Web crawlers.

Chapter 7

Crawler Architecture

7.1 Introduction

In the preceding chapters, we studied 1) how a crawler can discover and download important pages early 2) how we can parallelize the crawling process and 3) how the crawler should refresh downloaded pages. In this chapter, we study some of the remaining issues for a crawler and propose a crawler *architecture*:

- In Section 7.2 we identify some of the remaining design choices for a crawler and we *quantify* the impact of the choices using the experimental data of Chapter 4.
- In Section 7.3 we propose an architecture for a crawler, which maintains only "important" pages and adjusts revisit frequency for pages depending on how often they change.

7.2 Crawler design issues

The results of Chapter 4 showed us how Web pages change over time. Based on these results, we further 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 *freshness* metric described in Chapter 5. That is, we use the fraction of "up-to-date" pages in the local collection as the metric for how up-to-date the collection is. For example, if the crawler maintains 100 pages and if 70 out of 100 local pages are the same as the actual pages on the Web, the freshness of the collection is 0.7.

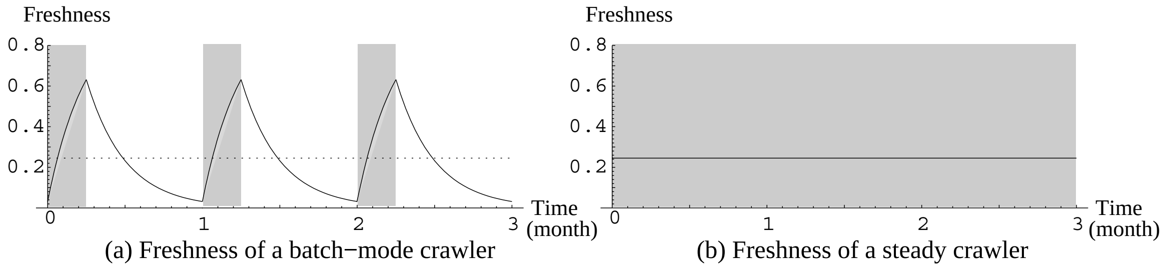


Figure 7.1: Freshness evolution of a batch-mode/steady crawler

(In Chapter 5, we also discussed 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 7.1(a). In the figure, the horizontal axis represents time and the gray 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. The derivation is similar to the one in Chapter 5.) 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 gray regions). Note that the freshness is not equal to 1 even at the end of each crawl (the right ends of gray regions), because some pages have already changed during the crawl.

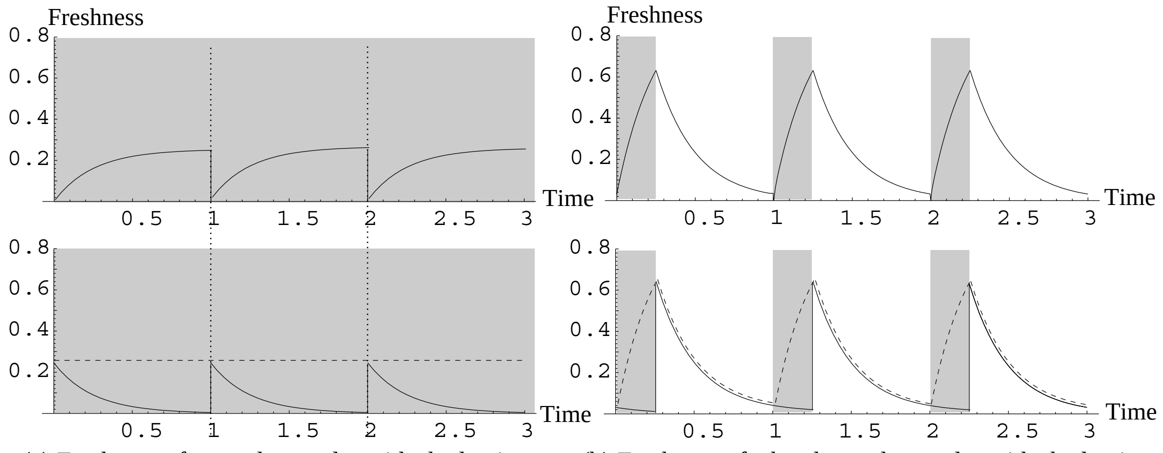
Also note that the freshness of the collection decreases exponentially in the white region. This trend is consistent with the experimental result of Figure 4.5.

- **Steady crawler:** A *steady crawler* runs continuously without any pause (Figure 7.1(b)). In the figure, the entire area is gray, 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 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 one of our early crawler prototypes ran at a very high speed, it once crashed the central router for the Stanford network. After that incident, Stanford network managers have closely monitored our crawling activity to ensure it runs at a reasonable speed. Also, the Web masters 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



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

Figure 7.2: Freshness of the crawler's and the current collection

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 7.2. 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 7.2(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 7.2(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 pages. To compare how freshness is affected by shadowing,

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

Table 7.1: Freshness of the collection for various choices

we show the freshness of the current collection *without shadowing* as a dashed line in Figure 7.2(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 7.2(b), we show the freshness of a *batch-mode* crawler when the collection is shadowed. The solid line in Figure 7.2(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 gray 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 gray 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 7.2(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 (gray regions), the freshness of the *in-place update* crawler is higher than that of the *shadowing* crawler, because new pages are immediately available to users with the in-place update crawler.

In Table 7.1 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 Chapter 4. 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 7.1 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 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.

   - **Uniform frequency:** The crawler revisits Web pages at the same frequency, regardless of how often they change. We believe this uniform-frequency policy matches well with a batch-mode crawler, since a batch-mode crawler commonly revisits all pages in the collection in every crawl.
   - **Variable frequency:** In Chapter 4 we showed that Web pages change at widely different frequencies, and in Chapter 5 we showed that a crawler may increase the freshness of its collection by 10%–23% if it adjusts page revisit frequencies based on how often pages change. Note that the variable-frequency policy is well suited for a *steady* crawler with *in-place updates*. Since a steady crawler visits pages continuously, it can adjust the revisit frequency with arbitrary granularity and thus increase the freshness of the collection.

We summarize the discussion of this section in Figure 7.3. As we have argued, there exist two “reasonable” combinations of options, which have different advantages. The crawler

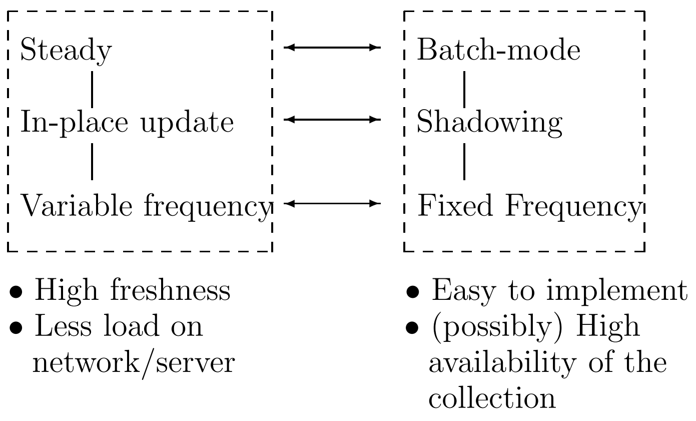


Figure 7.3: Two possible crawlers and their advantages

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. We refer to a crawler with the properties of the left-side as an *incremental crawler*, because it can continuously and incrementally update its collection of pages. In the next section, we discuss how we can implement an effective incremental crawler.

7.3 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.

7.3.1 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.

```
Algorithm 7.3.1 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 7.4: Conceptual operational model of an incremental crawler

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 4.4 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.

7.3.2 Operational model of an incremental crawler

In Figure 7.4 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 7.3.3, 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 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 and discard decision as a *refinement decision*.

Note that this refinement decision should be based on the "importance" of pages. To measure importance, the crawler may use various importance metrics listed in Chapter 2. 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 at 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 an *update decision*.

7.3.3 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 7.5. The architecture consists of three major modules (`RankingModule`, `UpdateModule` and `CrawlModule`) and three data structures (AllUrls, CollUrls and Collection). The lines and

[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.

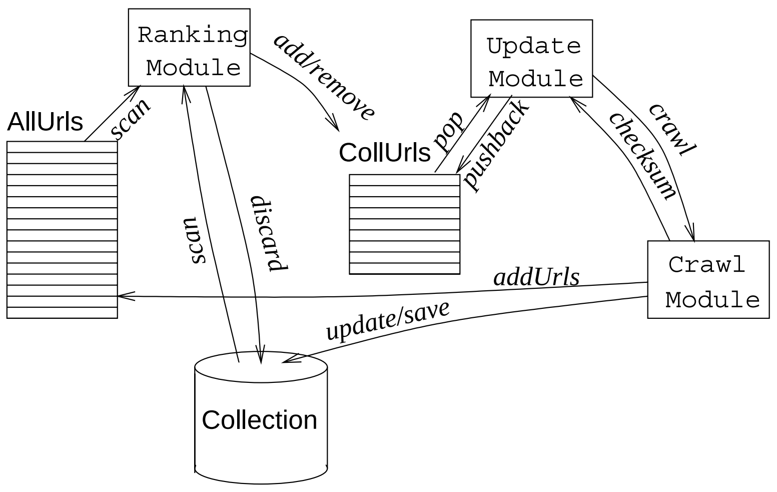


Figure 7.5: Architecture of the incremental crawler

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 7.4. 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 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 which is *not* in CollUrls turns out to be more important than a page within CollUrls, the `RankingModule` schedules for replacement the least-important page with the 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 least-important page(s) 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

[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.

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 Chapter 6, we proposed two estimators that the `UpdateModule` can use for frequency estimation.

The first estimator (described in Section 6.4) is based on the Poisson process model for Web page change. To implement this estimator, the `UpdateModule` has to record how many changes to a page it detected for, say, the last 6 months. Then it uses this number to get the confidence interval for the page change frequency.

The second estimator (described in Section 6.6) is based on the Bayesian estimation method. Informally, the goal of the second estimator is 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 Chapter 6.

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 the 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 a *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 more often than other pages.[3] To implement this policy, the `UpdateModule` also needs to record the "importance" of each page.

[3]This topic was discussed in more detail in Section 5.6.

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.

Separating the update decision (`UpdateModule`) from the refinement decision (`Ranking-Module`) is crucial for performance reasons. For example, to visit 100 million pages every month,[4] 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.

7.4 Conclusion

In this chapter we studied the architecture for an effective Web crawler. Using the experimental results in Chapter 4, we compared various design choices for a crawler and possible trade-offs. We then proposed an architecture for a crawler, which combines the best strategies identified.

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

Chapter 8

Conclusions and Future Work

As the Web grows larger and its contents become more diverse, the role of a Web crawler becomes even more important. In this dissertation we studied how we can implement an effective Web crawler that can discover and identify important pages early, retrieve the pages promptly in parallel, and maintain the retrieved pages fresh.

In Chapter 2, we started by discussing various definitions for the importance of a page, and we showed that a crawler can retrieve important pages significantly earlier by employing simple selection algorithms. In short, the PageRank ordering metric is very effective when the crawler considers highly-linked pages important. To find pages related to a particular topic, it may use anchor text and the distance from relevant pages.

In Chapter 3, we addressed the problem of crawler parallelization. Our goal was to minimize the overhead from the coordination of crawling processes while maximizing the downloads of highly-important pages. Our results indicate that when we run 4 or fewer crawling processes in parallel, a firewall-mode crawler is a good option, but for 5 or more processes, an exchange-mode crawler is a good option. For an exchange-mode crawler, we can minimize the coordination overhead by using the batch communication technique and by replicating 10,000 to 100,000 popular URLs in each crawling process.

In Chapter 4 we studied how the Web changes over time through an experiment conducted on 720,000 Web pages for 4 months. This experiment provided us with various statistics on how often Web pages change and how long they stay on the Web. We also observed that a Poisson process is a good model to describe Web page changes.

In Chapter 5 we compared various page refresh policies based on the Poisson model. Our analysis showed that the proportional policy, which is intuitively appealing, does not necessarily result in high freshness and that we therefore need to be very careful in adjusting a page revisit frequency based on page change frequency. We also showed that we can improve freshness very significantly by adopting our optimal refresh policy.

In Chapter 6, we explained how a crawler can estimate the change frequency of a page when it has a limited change history of the page. Depending on the availability of change information, we proposed several estimators appropriate for each scenario. We also showed, through theoretical analyses and experimental simulations, that our estimators can predict the change frequency much more accurately than existing ones.

Finally, in Chapter 7 we described a crawler architecture which can employ the techniques described in this dissertation. We also discussed some of the remaining issues for a crawler design and implementation.

As is clear from our discussion, some of our techniques are not limited to a Web crawler but can also be applied in other contexts. In particular, the algorithms that we described in Chapters 5 and 6 can be applied to any other applications that need to maintain a local copy of independently updated data sources.

The work in this dissertation resulted in the Stanford WebBase crawler, which currently maintains 130 million pages downloaded from the Web. The WebBase crawler consists of 20,000 lines of C/C++ code, and it can download 100 million Web pages in less than two weeks.[1] The pages downloaded by the WebBase crawler are being actively used by various researchers within and outside of Stanford.

8.1 Future work

We now briefly discuss potential areas for future work. In Chapter 2, we assumed that all Web pages can be reached by following the link structure of the Web. As more and more pages are dynamically generated, however, some pages are "hidden" behind a query interface [Ber01, KSS97, RGM01, IGS01]. That is, some pages are reachable only when the user issues *keyword queries* to a query interface. For these pages, the crawler cannot simply follow links but has to figure out the keywords to be issued. While this task is clearly challenging, the crawler may get some help from the "context" of the pages. For example, the crawler may examine the pages "surrounding" a query interface and guess that the pages are related to the US weather. Based on this guess, the crawler may issue

[1]We achieved this rate by running 4 crawling processes in parallel.

(city, state) pairs to the query interface and retrieve pages. Initial steps have been taken by Raghavan et al. [RGM01] and by Ipeirotis et al. [IGS01], but we believe a further study is necessary to address this problem.

In Chapter 2, we studied how a crawler can download "important" pages early by using an appropriate ordering metric. However, our study was based mainly on experiments rather than theoretical proofs. One interesting research direction would be to identify a *link-structure model* of the Web and design an optimal ordering metric based on that model. While we assumed a hypothetical ideal crawler in Chapter 2 and evaluated various ordering metrics against it, the ideal crawler is practically unfeasible in most cases. A Web link model and its formal analysis could show us the optimal crawler *in practice* and how well various ordering metrics perform compared to this optimal crawler.

In Chapter 5, we showed that it is very difficult to maintain a page up-to-date if the page changes too often. As more and more pages are dynamically generated, frequent changes may become a serious problem, and we may need to take one of the following approaches:

- *Separation of dynamic content:* In certain cases, the content of a page may change only in a particular section. For example, a product-related page on `Amazon.com` may be often updated in price, but not in the description of the product. When the changes of a page are focused on a particular section, it might be useful to separate the dynamic content from the static content and to refresh them separately. Incidentally, recent HTML standard proposes to separate the *style* of a page from its actual content, but more granular level of separation may be necessary.

- *Server-side push:* A major challenge of Chapter 5 was that the crawler does not know how often a page changes. Therefore the crawler often refreshes a page even when the page has not changed, wasting its limited resources. Clearly, if a Web server is willing to *push* changes to the Web crawler, this challenge can be addressed. To realize this "server-side push", we need to study how much overhead it may impose on the server and how we can minimize it. Also, we need to develop a mechanism by which a crawler can *subscribe* to the changes that it is interested in. In reference [BCGM00], we studied how much benefit a server and a crawler may get when the server publishes the list of its Web pages and their modification dates, so that the crawler can make a better crawling decision based on the information. In references [OW01, GL93, DRD99, PL91], various researchers proposed similar ideas that the data source and its caches cooperate to limit the difference between the data source and the caches, although the contexts are not exactly the same.

In Chapter 5, we assumed that Web pages are updated *independently* without any correlation among them. A certain set of pages, however, may be highly correlated and updated together. For example, consider a set of pages related to the "Clinton & Monica Lewinsky" relationship. The updates to these pages may be highly correlated, because most of the pages will be updated whenever there is new discovery on that topic. In this case, we may improve the freshness of the downloaded pages by using a *sampling* technique: when we want to refresh the pages, we first check only a couple of "sample" pages for change and refresh the remainder only if the sample pages have changed.

In this dissertation, we simply defined *any* change to a page as a change. For certain applications, however, we may need a different definition of change. For example, we may define a change as the change in the stock price on a Web page, in case of a stock-price monitoring system. We may also define the "freshness" of a price as how close the cached price is to the the current price. Under this new definition of change and freshness, it will be interesting to see how the optimal algorithm in Chapter 5 performs and how we should modify the algorithm to accommodate the difference.

Bibliography

[AAGY01] Charu C. Aggarwal, Fatima Al-Garawi, , and Philip S. Yu. Intelligent crawling on the world wide web with arbitrary predicates. In *Proceedings of the Tenth International World-Wide Web Conference*, Hong Kong, May 2001.

[ABGM90] Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, 15(3):359–384, September 1990.

[ADN$^+$95] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

[BA99] Albert-Laszlo Barabasi and Reka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, October 1999.

[BB99] Krishna Bharat and Andrei Broder. Mirror, mirror on the web: A study of host pairs with replicated content. In *Proceedings of the Eighth International World-Wide Web Conference*, Toronto, Canada, May 1999.

[BBC80] Philip Bernstein, Barbara Blaustein, and Edmund Clarke. Fast maintenance of semantic integrity assertions using redundant aggregate data. In *Proceedings of the Sixth International Conference on Very Large Databases*, pages 126–136, Montreal, Canada, October 1980.

[BBM$^+$97] Michael Baentsch, L. Baum, Georg Molter, S. Rothkugel, and P. Sturm. World Wide Web caching: The application-level view of the internet. *IEEE Communications Magazine*, 35(6):170–178, June 1997.

[BC00] Brian E. Brewington and George Cybenko. How dynamic is the web. In *Proceedings of the Ninth International World-Wide Web Conference*, Amsterdam, Netherlands, May 2000.

[BCGM00] Onn Brandman, Junghoo Cho, and Hector Garcia-Molina. Crawler-friendly servers. In *Proceedings of the Workshop on Performance and Architecture of Web Servers (PAWS)*, Santa Clara, California, June 2000.

[Ber01] Michael K. Bergman. The Deep Web: Surfacing Hidden Value. BrightPlanet White Paper, available at `http://www.brightplanet.com/deepcontent/tutorials/DeepWeb/`, June 2001.

[BG84] Philip Bernstein and Nathan Goodman. The failure and recovery problem for replicated distributed databases. *ACM Transactions on Database Systems*, 9(4):596–615, December 1984.

[BGM95] Daniel Barbara and Hector Garcia-Molina. The demarcation protocol: A technique for maintaining linear arithmetic constraints in distributed database systems. In *Proceedings of the Internationl Conference on Extending Database Technology*, pages 373–388, Vienna, Austria, 1995.

[BKM$^+$00] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web: experiments and models. In *Proceedings of the Ninth International World-Wide Web Conference*, Amsterdam, Netherlands, May 2000.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International World-Wide Web Conference*, Brisbane, Australia, April 1998.

[Bur98] Mike Burner. Crawling towards eterneity: Building an archive of the world wide web. *Web Techniques Magazine*, 2(5), May 1998.

[BYBCW00] Ziv Bar-Yossef, Alexander Berg, Steve Chien, and Jittat Fakcharoenphol Dror Weitz. Approximating aggregate queries about web pages via random walks. In *Proceedings of the Twenty-sixth International Conference on Very Large Databases*, Cairo, Egypt, September 2000.

[Can72] G.C. Canavos. A bayesian approach to parameter and reliability estimation in the Poisson distribution. *IEEE Transactions on Reliability*, R21(1):52–56, February 1972.

[CGM00a] Junghoo Cho and Hector Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proceedings of the Twenty-sixth International Conference on Very Large Databases*, Cairo, Egypt, September 2000.

[CGM00b] Junghoo Cho and Hector Garcia-Molina. Synchronizing a database to improve freshness. In *Proceedings of the International Conference on Management of Data*, Dellas, Texas, May 2000.

[CGM01] Junghoo Cho and Hector Garcia-Molina. Estimating frequency of change. Technical report, Stanford Database Group, 2001. `http://dbpubs.stanford.edu/pub/2000-4`.

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

[CKLlSM97] Latha S. Colby, Akira Kawaguchi, Daniel F. Lieuwen, and Inderpa l Singh Mumick. Supporting multiple view maintenance policies. In *Proceedings of the International Conference on Management of Data*, pages 405–416, Tuscon, Arizona, May 1997.

[CLW98] Edward G. Coffman, Jr., Zhen Liu, and Richard R. Weber. Optimal robot scheduling for web search engines. *Journal of Scheduling*, 1(1):15–29, June 1998.

[CvdBD99a] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Distributed hypertext resource discovery through examples. In *Proceedings of the Twenty-fifth International Conference on Very Large Databases*, Edinburgh, Scotland, UK, September 1999.

[CvdBD99b] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused crawling: A new approach to topic-specific web resource discovery. In *Proceedings* *of the Eighth International World-Wide Web Conference*, Toronto, Canada, May 1999.

[DCL$^+$00] Michelangelo Diligenti, Frans Coetzee, Steve Lawrence, C. Lee Giles, and Marco Gori. Focused crawling using context graphs. In *Proceedings of the Twenty-sixth International Conference on Very Large Databases*, pages 527–534, Cairo, Egypt, September 2000.

[dCR82] Osvaldo Sergio Farhat de Carvalho and Gerard Roucairol. On the distribution of an assertion. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 121–131, Ottawa, Canada, 1982.

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

[DRD99] Lyman Do, Prabhu Ram, and Pamela Drew. The need for distributed asynchronous transactions. In *Proceedings of the International Conference on Management of Data*, pages 534–535, Philadelphia, Pennsylvania, June 1999.

[Eic94] David Eichmann. The RBSE spider: Balancing effective search against web load. In *Proceedings of the First World-Wide Web Conference*, Geneva, Switzerland, May 1994.

[GL93] Richard A. Golding and Darrell D.E. Long. Modeling replica divergence in a weak-consistency protocol for global-sc ale distributed data bases. Technical report UCSC-CRL-93-09, Computer and Information Sciences Board, University of California, Santa Cruz, 1993.

[Goo] Google Inc. `http://www.google.com`.

[GS96] James Gwertzman and Margo Seltzer. World-Wide Web cache consistency. In *Proceedings of USENIX 1996 Annual Technical Conference*, San Diego, California, January 1996.

[HGMW$^+$95] Joachim Hammer, Hector Garcia-Molina, Jennifer Widom, Wilburt J. Labio, and Yue Zhuge. The Stanford data warehousing project. *IEEE Data Engineering Bulletin*, 18(2):41–48, June 1995.

[Hir76] Daniel S. Hirschberg. Parallel algorithms for the transitive closure and the connected component problem. In *Proceedings of the 8th Annual ACM Symposium on the Theory of Computing*, pages 55–57, Hershey, Pennsylvania, 1976.

[HJ$^+$98] Michael Hersovicia, Michal Jacovia, Yoelle S. Maareka Dan Pellegb, Menachem Shtalhaima, and Sigalit Ura. The shark-search algorithm - an application: tailored web site mapping. In *Proceedings of the Seventh International World-Wide Web Conference*, Brisbane, Australia, April 1998.

[HN99] Allan Heydon and Marc Najork. Mercator: A scalable, extensible web crawler. In *Proceedings of the Eighth International World-Wide Web Conference*, pages 219–229, Toronto, Canada, May 1999.

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffery D. Ullman. Implementing data cubes efficiently. In *Proceedings of the International Conference on Management of Data*, Montreal, Canada, June 1996.

[IGS01] Panagiotis G. Ipeirotis, Luis Gravano, and Mehran Sahami. Probe, count, and classify: Categorizing hidden-web databases. In *Proceedings of the International Conference on Management of Data*, Santa Barbara, California, May 2001.

[Kah97] Brewster Kahle. Archiving the internet. *Scientific American*, March 1997.

[KB91] Narayanan Krishnakumar and Arthur Bernstein. Bounded ignorance in replicated systems. In *Proceedings of the 10th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 63–74, Denver, Colorado, May 1991.

[KB94] Narayanan Krishnakumar and Arthur Bernstein. Bounded ignorance: A technique for increasing concurrency in a repli cated system. *ACM Transactions on Database Systems*, 19(4):586–625, December 1994.

[Kos95] Martijn Koster. Robots in the web: threat or treat? *ConneXions*, 4(4), April 1995.

[KSS97] Henry Kautz, Bart Selman, and Mehul Shah. The hidden web. *AI Magazine*, 18(2):27–36, Summer 1997.

[LG98] Steve Lawrence and C. Lee Giles. Searching the World Wide Web. *Science*, 280(5360):98–100, April 1998.

[LG99] Steve Lawrence and C. Lee Giles. Accessibility of information on the web. *Nature*, 400(6740):107–109, July 1999.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–357, November 1989.

[LLSG92] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

[Mah89] S.M. Mahmud. High precision phase measurement using adaptive sampling. *IEEE Transactions on Instrumentation and Measurement*, 38(5):954–960, October 1989.

[Mar97] Massimo Marchiori. The quest for correct information on the web: Hyper search engines. In *Proceedings of the Sixth International World-Wide Web Conference*, pages 265–276, Santa Clara, California, April 1997.

[McB94] Oliver A. McBryan. GENVL and WWWW: Tools for taming the web. In *Proceedings of the First World-Wide Web Conference*, Geneva, Switzerland, May 1994.

[MDP$^+$00] Dejan S. MiloJicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, September 2000.

[Mil98] Eric Miller. An introduction to the resource description framework. *D-Lib Magazine*, May 1998. `http://www.dlib.org/dlib/may98/miller/05miller.html`.

[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, August 1984.

[MNRS99] Andrew McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Building domain-specific search engines with machine learning techniques. In *AAAI Spring Symposium on Intelligent Agents in Cyberspace 1999*, Stanford, CA, March 1999.

[MPR01] Filippo Menczer, Gautam Pant, and Miguel E. Ruiz. Evaluating topic-driven web crawlers. In *Proceedings of the Twenty-Fourth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, New Orleans, LA, September 2001.

[MS75] P.N. Misra and H.W. Sorenson. Parameter estimation in Poisson processes. *IEEE Transactions on Information Theory*, IT-21(1):87–90, January 1975.

[Muk00] Sougata Mukherjea. Wtms: A system for collecting and analysing topic-specific web information. In *Proceedings of the Ninth International World-Wide Web Conference*, Amsterdam, Netherlands, May 2000.

[NS82] David Nassimi and Sartaj Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *Journal of the ACM*, 29:642–667, July 1982.

[NW01] Marc Najork and Janet L. Wiener. Breadth-first search crawling yields high-quality pages. In *Proceedings of the Tenth International World-Wide Web Conference*, Hong Kong, May 2001.

[OS75] Alan V. Oppenheim and Ronald W. Schafer. *Digital Signal Processing.* Prentice Hall, first edition, 1975.

[OV99] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, second edition, 1999.

[OW00] Chris Olston and Jennifer Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Proceedings of the Twenty-sixth International Conference on Very Large Databases*, Cairo, Egypt, September 2000.

[OW01] Chris Olston and Jennifer Widom. Best-effort cache synchronization with source cooperation. Technical report, Stanford Database Group, 2001. `http://dbpubs.stanford.edu/pub/2001-43`.

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

[Pin94] Brian Pinkerton. Finding what people want: Experiences with the web crawler. In *Proceedings of the Second World-Wide Web Conference*, Chicago, Illinois, October 1994.

[PL91] Calton Pu and Avraham Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings of the International Conference on Management of Data*, pages 377–386, Denver, Colorado, May 1991.

[PP97] James Pitkow and Peter Pirolli. Life, death, and lawfulness on the electronic frontier. In *Proceedings of the Conference on Human Factors in Computing Systems CHI'97*, pages 383–390, Atlanta, Georgia, March 1997.

[PPR96] Peter Pirolli, James Pitkow, and Ramana Rao. Silk from a sow's ear: Extracting usable structures from the web. In *Proceedings of the Conference on Human Factors in Computing Systems CHI'96*, pages 118–125, Vancouver, British Columbia, Canada, April 1996.

[QD84] Michael J. Quinn and Narsingh Deo. Parallel graph algorithms. *ACM Computing Surveys*, 16(3):319–348, September 1984.

[RGM01] Sriram Raghavan and Hector Garcia-Molina. Crawling the hidden web. In *Proceedings of the Twenty-seventh International Conference on Very Large Databases*, Rome, Italy, September 2001.

[Rob] Robots exclusion protocol. `http://info.webcrawler.com/mak/projects/robots/exclusion.html`.

[Sal83] Gerard Salton. *Introduction to modern information retrieval.* McGraw-Hill, first edition, 1983.

[SKK$^+$90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[Tho69] George B. Thomas, Jr. *Calculus and analytic geometry.* Addison-Wesley, 4th edition, 1969.

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

[TL98] P.W.M. Tsang and W.T. Lee. An adaptive decimation and interpolation scheme for low complexity image compression. *Signal Processing*, 65(3):391–401, March 1998.

[TR85] Andrew S. Tanenbaum and Robbert Van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4), December 1985.

[Win72] Robert L. Winkler. *An Introduction to Bayesian Inference and Decision.* Holt, Rinehart and Winston, Inc, first edition, 1972.

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

[WMS97] Dennis D. Wackerly, William Mendenhall, and Richard L. Scheaffer. *Mathematical Statistics With Applications.* PWS Publishing, 5th edition, 1997.

[WVS$^+$99] Alec Wolman, Geoffrey M. Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry M. Levy. On the scale and performance of cooperative web proxy caching. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 16–31, December 1999.

[YBS99] Haobo Yu, Lee Breslau, and Scott Shenker. A scalable Web cache consistency architecture. In *Proceedings of the 1999 ACM SIGCOMM Conference*, Cambridge, Massachusetts, August 1999.

[YC96] S. Yacout and Yusuo Chang. Using control charts for parameter estimation of a homogeneous poisson process. In *Proceedings of the 19th International Conference on Computers and Industrial Engineering*, Kyongju, Korea, October 1996.

[YLYL95] Budi Yuwono, Savio L. Lam, Jerry H. Ying, and Dik L. Lee. A world wide web resource discovery system. In *Proceedings of the Fourth International World-Wide Web Conference*, Darmstadt, Germany, April 1995.

[YV00] Haifeng Yu and Amin Vahdat. Efficient numerical error bounding for replicated network services. In *Proceedings of the Twenty-sixth International Conference on Very Large Databases*, pages 123–133, Cairo, Egypt, September 2000.

[ZGMHW95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *Proceedings of the International Conference on Management of Data*, San Jose, California, May 1995.

[Zip49] George Kingsley Zipf. *Human Behaviour and the Principle of Least Effort: an Introduction to Human Ecology.* Addison-Wesley, first edition, 1949.