Searching the Web

Arvind Arasu Junghoo Cho Hector Garcia-Molina Andreas Paepcke Sriram Raghavan
Computer Science Department, Stanford University
{arvinda,cho,hector,paepcke,rsram}@cs.stanford.edu

Abstract

We offer an overview of current Web search engine design. After introducing a generic search engine architecture, we examine each engine component in turn. We cover crawling, local Web page storage, indexing, and the use of link analysis for boosting search performance. The most common design and implementation techniques for each of these components are presented. We draw for this presentation from the literature, and from our own experimental search engine testbed. Emphasis is on introducing the fundamental concepts, and the results of several performance analyses we conducted to compare different designs.

Keywords: Search engine, crawling, indexing, link analysis, PageRank, HITS, hubs, authorities, information retrieval.

1 Introduction

The plentiful content of the World-Wide Web is useful to millions. Some simply browse the Web through entry points such as Yahoo!. But many information seekers use a *search engine* to begin their Web activity. In this case, users submit a query, typically a list of keywords, and receive a list of Web pages that may be relevant, typically pages that contain the keywords. In this paper we discuss the challenges in building good search engines, and describe some of the techniques that are useful.

Many of the search engines use well-known information retrieval (IR) algorithms and techniques [55, 28]. However, IR algorithms were developed for relatively small and coherent collections such as newspaper articles or book catalogs in a (physical) library. The Web, on the other hand, is massive, much less coherent, changes more rapidly, and is spread over geographically distributed computers. This requires new techniques, or extensions to the old ones, to deal with the gathering of the information, to make index structures scalable and efficiently updateable, and to improve the discriminating ability of search engines. For the last item, discriminating ability, it is possible to exploit the linkage among Web pages to better identify the truly relevant pages.

There is no question that the Web is huge and challenging to deal with. Several studies have estimated the size of the Web [4, 42, 41, 6], and while they report slightly different numbers, most of

them agree that over a billion pages are available. Given that the average size of a Web page is around 5–10K bytes, just the textual data amounts to at least tens of terabytes. The growth rate of the Web is even more dramatic. According to [41, 42], 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, the existing pages are continuously updated [52, 58, 24, 17]. For example, in our own study of over half a million pages over 4 months [17], we found that about 23% of pages changed daily. In the `.com` domain 40% of the pages changed daily, and the half-life of pages is about 10 days (in 10 days half of the pages are gone, i.e., their URLs are no longer valid). In [17], we also report that a Poisson process is a good model for Web page changes. Later in Section 2, we will show how some of these results can be used to improve search engine quality.

In addition to size and rapid change, the interlinked nature of the Web sets it apart from many other collections. Several studies aim to understand how the Web's linkage is structured and how that structure can be modeled [11, 5, 2, 36, 17]. One recent study, for example, suggests that the link structure of the Web is somewhat like a "bow-tie" [11]. That is, about 28% of the pages constitute a strongly connected core (the center of the bow tie). About 22% form one of the tie's loops: these are pages that can be reached from the core but not vice versa. The other loop consists of 22% of the pages that can reach the core, but cannot be reached from it. (The remaining nodes can neither reach the core nor can be reached from the core.)

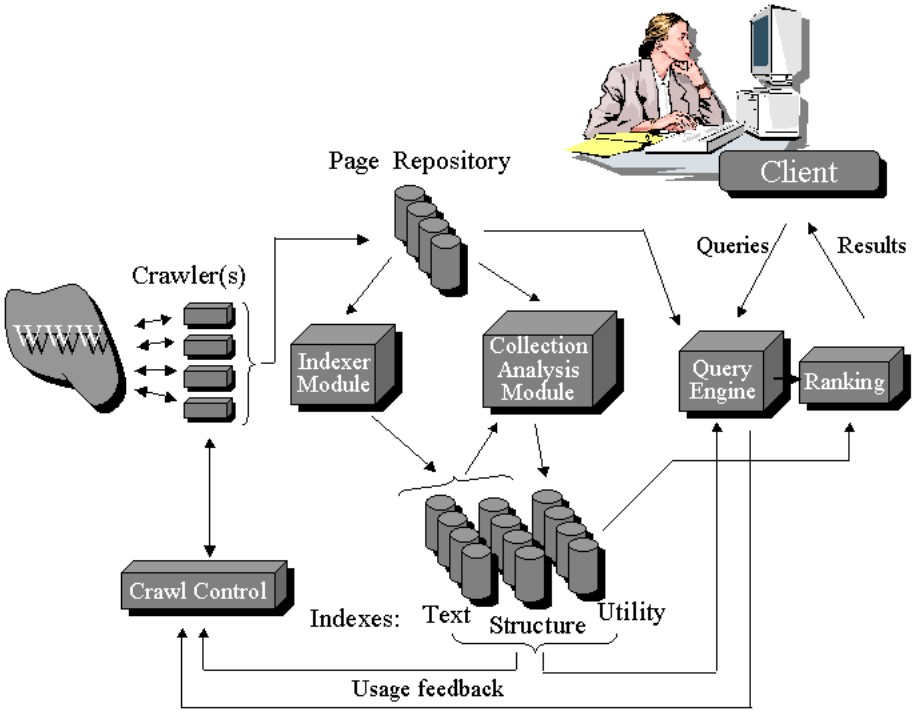

Figure 1: General search engine architecture

Before we describe search engine techniques, it is useful to understand how a Web search engine is

typically put together. Figure 1 shows such an engine schematically. Every engine relies on a *crawler* module to provide the grist for its operation (shown on the left in Figure 1). Crawlers are small programs that 'browse' the Web on the search engine's behalf, similarly to how a human user would follow links to reach different pages. The programs are given a starting set of URLs, whose pages they retrieve from the Web. The crawlers extract URLs appearing in the retrieved pages, and give this information to the *crawler control* module. This module determines what links to visit next, and feeds the links to visit back to the crawlers. (Some of the functionality of the crawler control module may be implemented by the crawlers themselves.) The crawlers also pass the retrieved pages into a *page repository*. Crawlers continue visiting the Web, until local resources, such as storage, are exhausted.

This basic algorithm is modified in many variations that give search engines different levels of coverage or topic bias. For example, crawlers in one engine might be biased to visit as many sites as possible, leaving out the pages that are buried deeply within each site. The crawlers in other engines might specialize on sites in one specific domain, such as governmental pages. The crawl control module is responsible for directing the crawling operation.

Once the search engine has been through at least one complete crawling cycle, the crawl control module may be informed by several indexes that were created during the earlier crawl(s). The crawl control module may, for example, use a previous crawl's link graph (the *structure index* in Figure 1) to decide which links the crawlers should explore, and which links they should ignore. Crawl control may also use feedback from usage patterns to guide the crawling process (connection between the query engine and the crawl control module in Figure 1). Section 2 will examine crawling operations in more detail.

The *indexer* module extracts all the words from each page, and records the URL where each word occurred. The result is a generally very large "lookup table" that can provide all the URLs that point to pages where a given word occurs (the *text index* in Figure 1). The table is of course limited to the pages that were covered in the crawling process. As mentioned earlier, text indexing of the Web poses special difficulties, due to its size, and its rapid rate of change. In addition to these quantitative challenges, the Web calls for some special, less common kinds of indexes. For example, the indexing module may also create a structure index, which reflects the links between pages. Such indexes would not be appropriate for traditional text collections that do not contain links. The *collection analysis module* is responsible for creating a variety of other indexes.

The *utility index* in Figure 1 is created by the collection analysis module. For example, utility indexes may provide access to pages of a given length, pages of a certain "importance," or pages with some number of images in them. The collection analysis module may use the text and structure indexes when creating utility indexes. Section 4 will examine indexing in more detail.

During a crawling and indexing run, search engines must store the pages they retrieve from the Web. The page repository in Figure 1 represents this—possibly temporary—collection. Sometimes search engines maintain a cache of the pages they have visited beyond the time required to build the

index. This cache allows them to serve out result pages very quickly, in addition to providing basic search facilities. Some systems, such as the Internet Archive [37], have aimed to maintain a very large number of pages for permanent archival purposes. Storage at such a scale again requires special consideration. Section 3 will examine these storage-related issues.

The *query engine* module is responsible for receiving and filling search requests from users. The engine relies heavily on the indexes, and sometimes on the page repository. Because of the Web's size, and the fact that users typically only enter one or two keywords, result sets are usually very large. The *ranking* module therefore has the task of sorting the results such that results near the top are the most likely ones to be what the user is looking for. The query module is of special interest, because traditional information retrieval (IR) techniques have run into selectivity problems when applied without modification to Web searching: Most traditional techniques rely on measuring the similarity of query texts with texts in a collection's documents. The tiny queries over vast collections that are typical for Web search engines prevent such similarity based approaches from filtering sufficient numbers of irrelevant pages out of search results. Section 5 will introduce search algorithms that take advantage of the Web's interlinked nature. When deployed in conjunction with the traditional IR techniques, these algorithms significantly improve retrieval precision in Web search scenarios.

In the rest of this article we will describe in more detail the search engine components we have presented. We will also illustrate some of the specific challenges that arise in each case, and some of the techniques that have been developed. Our paper is not intended to provide a complete survey of techniques. As a matter of fact, the examples we will use to illustrate will be drawn mainly from our own work since it is what we know best.

In addition to research at universities and open laboratories, many "dot-com" companies have worked on search engines. Unfortunately, many of the techniques used by dot-coms, and especially the resulting performance, are kept private behind company walls, or are 'disclosed' in patents whose language only lawyers can comprehend and appreciate. We therefore believe that the overview of problems and techniques we provide here can be of use.

2 Crawling Web pages

The crawler module (Figure 1) retrieves pages from the Web for later analysis by the indexing module. As discussed in the introduction, a crawler module typically starts off with an initial set of URLs S_0. Roughly, 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. Given the enormous size and the change rate of the Web, many issues arise, 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 [42, 6]. Given this fact, it is important for the crawler to carefully select the pages and to visit "important" pages first by prioritizing the URLs in the queue properly, 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 [18, 58], the crawler needs to carefully decide what page to revisit and what page to skip, because this decision may significantly impact the "freshness" of the downloaded collection. 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 [39]. 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 resource. Also, after this retrieval the page needs to be transferred through the network, which is another resource shared by multiple organizations. The crawler should minimize its impact on these resources [54]. Otherwise, the administrators of the Web site or a particular network may complain and sometimes 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 [10, 18]. 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, and the adopted crawling policy should be strictly enforced. The coordination can incur significant communication overhead, limiting the number of simultaneous crawlers.

In the rest of this section we discuss the first two issues, page selection and page refresh, in more detail. We do not discuss load or parallelization issues, mainly because much less research has been done on those topics.

2.1 Page selection

As we argued, the crawler may want to download "important" pages first, so that the downloaded collection is of high quality. There are three questions that need to be addressed: the meaning of "importance," how a crawler operates, and how a crawler "guesses" good pages to visit. We discuss these questions in turn, using our own work to illustrate some of the possible techniques.

2.1.1 Importance metrics

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

1. *Interest Driven.* The goal is to obtain pages "of interest" to a particular user or set of users. So important pages are those that match the interest of users. One particular way to define this notion is through what we call a *driving query*. Given a query Q, the importance of page P is defined to be the "textual similarity" [55] between P and Q. More formally, we compute textual similarity by first viewing each document (P or Q) as an m-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. Then we define the *similarity* between P and Q as a *cosine product* between the P and Q vectors [55]. Assuming that query Q represents the user's interest, this metric shows how "relevant" P is. We use $IS(P)$ to refer to this particular importance metric.

   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., 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 $IS(P)$, which can be computed only after the entire Web has been crawled.

   Reference [16] presents another interest-driven approach based on a hierarchy of topics. Interest is defined by a topic, and the crawler tries to guess the topic of pages that will be crawled (by analyzing the link structure that leads to the candidate pages).

2. *Popularity Driven.* Page importance depends on how "popular" a page is. For instance, one way to define popularity is to use a page's *backlink count*. (We use the term *backlink* for links that point *to* a given page.) Thus a Web page P's backlinks are the set of all links on pages other than P, which point to P. When using backlinks as a popularity metric, the importance value of 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 in bibliometrics 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. (The estimate may be inaccurate early on in a crawl.) Later in Section 5 we define a similar yet more sophisticated metric, called PageRank $IR(P)$, that can also be used as a popularity measure.

3. *Location Driven.* 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 of more interest than other URLs. Another location metric that is sometimes used considers URLs with fewer slashes more useful than those with more slashes. Location driven metrics can be considered a special case of interest driven ones, but we list them separately because they are often easy to evaluate. In particular, all the location metrics we have mentioned here are local 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 $IC(P) = k_1 \cdot IS(P) + k_2 \cdot IB(P) + k_3 \cdot IL(P)$, for some constants k_1, k_2, k_3 and query Q. This combines the similarity metric, the backlink metric and the location metric.

2.1.2 Crawler models

Our goal is to design a crawler that if possible visits high importance pages before lower ranked ones, for a certain importance metric. Of course, the crawler will only have *estimated* importance values (e.g., $IB'(P)$) available. Based on these estimates, the crawler will have to guess the high importance pages to fetch next. For example, we may define the *quality metric* of a crawler in one of the following two ways:

- **Crawl & Stop:** Under this model, the crawler C starts at its initial page P_0 and stops after visiting K pages. (K is a *fixed number* determined by the number of pages that the crawler can download in one crawl.) At this point a perfect 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 ($\leq K$) pages with rank higher than or equal to that of R_K. (Note that we need to know the *exact* rank of *all* pages in order to obtain the value M. Clearly, this estimation may not be possible until we download all pages and obtain the global image of the Web. Later in Section 2.1.4, we restrict the entire Web to the pages in the Stanford domain and estimate the ranks of pages based on this assumption.) Then we define the performance of the crawler C to be $P_{CS}(C) = (M \cdot 100)/K$. The performance of the ideal crawler is of course 100%. A crawler that somehow manages to visit pages entirely at random, and may revisit pages, would have a performance of $(K \cdot 100)/T$, 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 importance higher than G is considered hot. Let us assume that the total number of hot pages is H. Again, we assume that we know the ranks of all pages and thus can to obtain the value H. The performance of the crawler, $P_{ST}(C)$, is the percentage of the H hot pages that have been visited when the crawler stops. If $K < H$, then an ideal crawler will have performance $(K \cdot 100)/H$. If $K \geq H$, then the ideal crawler has 100% performance. A purely random crawler that revisits pages is expected to visit $(H/T) \cdot K$ hot pages when it stops. Thus, its performance is $(K \cdot 100)/T$. Only if the random crawler visits all T pages, is its performance expected to be 100%.

2.1.3 Ordering metrics

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

The ordering 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 an $IB'(P)$ as the ordering metric, where P is the page u points to. However, it might also make sense to consider an $IR'(P)$ (the PageRank metric, Section 5), even if our importance metric is the simpler citation count. In the next subsection we show why this may be the case.

Location metrics can be used directly for ordering, since the URL of P directly gives the $IL(P)$ value. However, for similarity metrics, it is much harder to devise an ordering metric, since we have not seen P yet. 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 is $IS(A)$ (for some query Q), where A is the anchor text of the URL u. Reference [23] proposes an approach like this, where not just the anchor text, but all the text of a page (and "near" pages) is considered for $IS(P)$.

2.1.4 Case Study

To illustrate that it is possible to download important pages significantly earlier when we adopt an appropriate ordering metric, we show some results from an experiment we conducted. To keep our experiment manageable, we defined the entire Web to be the 225,000 Stanford University Web pages that were downloaded by our Stanford WebBase crawler. That is, we assumed that all pages outside Stanford have "zero importance value," and that links to pages outside Stanford or from pages outside

Stanford do not count in page importance computations. Note that since the pages were downloaded by our crawler, they are all reachable from the Stanford homepage.

Under this assumption, we experimentally measured the performance of various ordering metrics for the importance metric $IB(P)$, and we show the result in Figure 2. In this graph, we assumed the Crawl & Stop model with Threshold, with threshold value $G = 100$. That is, pages with more than 100 backlinks were considered "hot," and we measured how many hot pages were downloaded when the crawler had visited x% of the Stanford pages. Under this definition, the total number of hot pages was 1,400, which was about 0.8% of all Stanford pages. The horizontal axis represents the percentage of pages downloaded from the Stanford domain, and the vertical axis shows the percentage of *hot* pages downloaded.

In the experiment, the crawler started at the Stanford homepage (`http://www.stanford.edu`) and, in three different experimental conditions, selected the next page visit either by the ordering metric $IR'(P)$ (PageRank), by $IB'(P)$ (backlink), or by following links breadth-first (breadth). The straight line in the graph shows the expected performance of a random crawler.

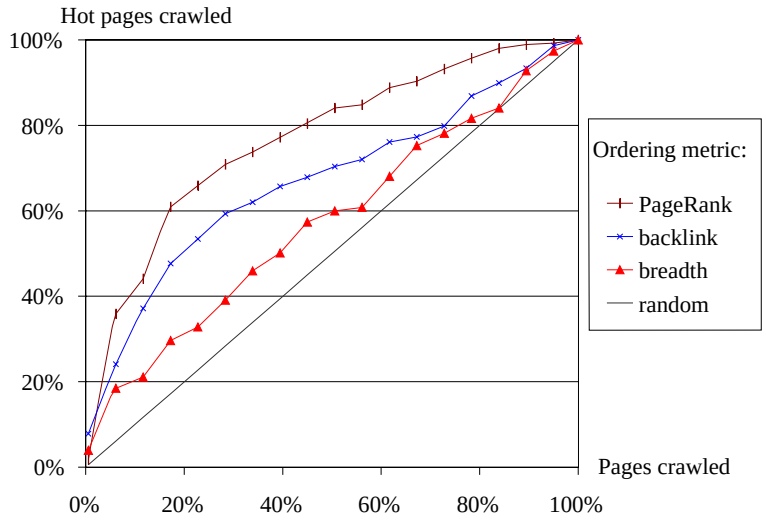

Figure 2: The performance of various ordering metrics for $IB(P)$; $G = 100$

From the graph, we can clearly see that an appropriate ordering metric can significantly improve the performance of the crawler. For example, when the crawler used $IB'(P)$ (backlink) as its ordering metric, the crawler downloaded more than 50% of hot pages, when it visited less than 20% of the entire Web. This is a significant improvement compared to a random crawler or a breadth first crawler, which downloaded less than 30% of hot pages at the same point. One interesting result of this experiment is that the PageRank ordering metric, $IR'(R)$, shows better performance than the backlink ordering metric $IB'(R)$, even when the importance metric is $IB(R)$. This is because of the inheritance property of the PageRank metric, which can help avoid downloading "locally popular" pages before "globally popular," but "locally unpopular" pages. In additional experiments [20] (not described here) we study

other metrics, and also observe that the right ordering metric can significantly improve the crawler performance.

2.2 Page refresh

Once the crawler has selected and downloaded "important" pages, it has to periodically refresh the downloaded pages, so that the pages are maintained up-to-date. Clearly there exist multiple ways to update the pages, and different strategies will result in different "freshness" of the pages. For example, consider the following two strategies:

1. **Uniform refresh policy:** The crawler revisits all pages at the same frequency f, regardless of how often they change.

2. **Proportional refresh policy:** The crawler revisits a page proportionally more often, as it changes more often. More precisely, assume that λ_i is the change frequency of a page e_i, and that f_i is the crawler's revisit frequency for e_i. Then the frequency ratio λ_i/f_i is the same for any i. For example, if page e_1 changes 10 times more often than page e_2, the crawler revisits e_1 10 times more often than e_2.

   Note that the crawler needs to estimate λ_i's for each page, in order to implement this policy. This estimation can be based on the change history of a page that the crawler can collect [17]. For example, if a crawler visited/downloaded a page p_1 every day for a month, and it detected 10 changes, the crawler may reasonable estimate that λ_1 is one change every 3 days. For more detailed discussion on λ estimation, see [17].

Between these two strategies, which one will give us higher "freshness?" Also, can we come up with an even better strategy? To answer these questions, we need to understand how Web pages change over time and what we mean by "freshness" of pages. In the next two subsections, we go over possible answers to these questions and compare various refresh strategies.

2.2.1 Freshness metric

Intuitively, we consider a collection of pages "fresher" when the collection has more up-to-date pages. For instance, consider two collections, A and B, containing the same 20 web pages. Then if A maintains 10 pages up-to-date on average, and if B has maintains 15 up-to-date pages, we consider B to be fresher than A. Also, we have a notion of "age:" even if all pages are obsolete, we consider collection A "more current" than B, if A was refreshed 1 day ago, and B was refreshed 1 year ago. Based on this intuitive notion, we have found the following definitions of *freshness* and *age* to be useful. (Incidentally, [21] has a slightly different definition of freshness, but it leads to results that are analogous to ours.) In the following discussion, we refer to the pages on the Web that the crawler monitors as the *real-world pages* and their local copies as the *local pages*.

1. **Freshness:** Let $S = \{e_1, \ldots, e_N\}$ be the local collection of N pages. Then we define the *freshness* of the collection as follows.

   Definition 1 The *freshness* of a local page e_i at time t is

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

   (By up-to-date we mean that the content of a local page equals that of its real-world counterpart.) Then, the *freshness* of the local collection S at time t is

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

   The *freshness* is the fraction of the local collection that is up-to-date. For instance, $F(S; t)$ will be one if all local pages are up-to-date, and $F(S; t)$ will be zero if all local pages are out-of-date.

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

   Definition 2 The *age* of the local page 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 collection S is

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

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

Obviously, the freshness (and age) of the local collection 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. Because of this possible fluctuation, we now compute the average freshness over a long period of time and use this value as the "representative" freshness of a collection.

Definition 3 We define the time average of freshness of page e_i, $\bar{F}(e_i)$, and the time average of freshness of collection 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. □

Mathematically, the time average of freshness or age may not exist, because we take a limit over time. However, the above definition approximates the fact that we take the average of freshness over a long period of time. Later on, we will see that the time average *does* exist when pages change by a reasonable process and when the crawler periodically revisits each page.

2.2.2 Refresh strategy

In comparing the page refresh strategies, it is important to note that crawlers can download/update only a limited number of pages within a certain period, because crawlers have limited resources. For example, many search engines report that their crawlers typically download several hundred pages per second. (Our own crawler, which we call the WebBase crawler, typically runs at the rate of 50–100 pages per second.) Depending on the page refresh strategy, this limited *page download resource* will be allocated to different pages in different ways. For example, the proportional refresh policy will allocate this download resource proportionally to the page change rate.

To illustrate the issues, consider a very simple example. Suppose that the crawler maintains a collection of two pages: e_1 and e_2. Page e_1 changes 9 times per day and e_2 changes once a day. Our goal is to maximize the freshness of the database averaged over time. In Figure 3, we illustrate our simple model. For page 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 page changes within an interval. Page e_2 changes *once and only once* per day, but we do not know precisely when it changes.

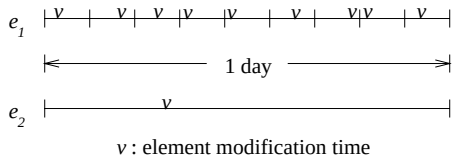

Figure 3: A database with two pages with different change frequencies

Because our crawler is a tiny one, assume that we can refresh *one* page per day. Then what page should it refresh? Should the crawler refresh e_1 or should it refresh e_2? To answer this question, we need to compare how the freshness changes if we pick one page over the other. If page e_2 changes in the middle of the day and if we refresh e_2 right after the change, it will remain up-to-date for the remaining half of the day. Therefore, by refreshing page 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 refreshing e_2 is $1/2 \times 1/2$ day $= 1/4$ day. By the same reasoning, if we refresh 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 refresh!

Of course, in practice, we do not know for sure that pages will change in a given interval. Furthermore, we may also want to worry about the age of data. (In our example, if we always visit e_2, the age of e_1 will grow indefinitely.)

In [19], we have studied a more realistic scenario, using the Poisson process model. In particular, we can mathematically prove that the uniform policy is *always* superior or equal to the proportional one, for any number of pages, change frequencies, and refresh rates, and for both the freshness and the age

metrics, when page changes follow Poisson processes. For a detailed proof, see [19].

In [19] we also show how to obtain the optimal refresh policy (better than uniform or any other), assuming page changes follow a Poisson process and their change frequencies are static (i.e., do not change over time). To illustrate, in Figure 4 we show the refresh frequencies that *maximizes* the freshness value for a simple scenario. In this scenario, the crawler maintains 5 pages with change rates, $1, 2, \ldots, 5$ (times/day), respectively, and the crawler can download 5 pages per day. The graph in Figure 4 shows the needed refresh frequency of a page (vertical axis) as a function of its change frequency (horizontal axis), in order to maximize the freshness of the 5 page collection. For instance, the optimal revisit frequency for the page that changes once a day is 1.15 times/day. Notice that the graph does not monotonically increase over change frequency, and thus we need to refresh pages less often if the pages change too often. The pages with change frequency larger than 2.5 times/day should be refreshed less often than the ones with change frequency 2 times/day. When a certain page changes too often, and if we cannot maintain it up-to-date under our resource constraint, it is in fact better to focus our resource on the pages that we can keep track of.

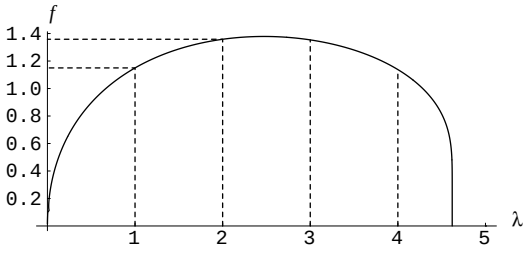

Figure 4: change frequency vs. refresh frequency for freshness optimization

Figure 4 is for a particular 5-page scenario, but in [19] we prove that the shape of the graph is the same for *any* distribution of change frequencies under the Poisson process model. That is, the optimal graph for *any* collection of pages S is *exactly the same* as Figure 4, except that the graph of S is scaled by a constant factor from Figure 4. Thus, no matter what the scenario, pages that change too frequently (relative to the available resources) should be penalized and not visited very frequently.

We can obtain the optimal refresh policy for the age metric, as described in [19].

2.3 Conclusion

In this section, we discussed the challenges that a crawler encounters when it downloads large collections of pages from the Web. In particular, we studied how a crawler should *select* and *refresh* the pages that it retrieves and maintains.

There are, of course, still many open issues. For example, it is not clear how a crawler and a Web site can negotiate/agree on a right crawling policy, so that the crawler does not interfere with the primary operation of the site, while downloading the pages on the site. Also, existing work on crawler

parallelization is either ad hoc or quite preliminary, so we believe this issue needs to be carefully studied. Finally, some of the information on the Web is now "hidden" behind a search interface, where a query must be submitted or a form filled out. Current crawlers cannot generate queries or fill out forms, so they cannot visit the "dynamic" content. This problem will get worse over time, as more and more sites generate their Web pages from databases.

3 Storage

The *page repository* in Figure 1 is a scalable storage system for managing large collections of Web pages. As shown in the figure, the repository needs to perform two basic functions. First, it must provide an interface for the crawler to store pages. Second, it must provide an efficient access API that the indexer and collection analysis modules can use to retrieve the pages. In the rest of the section, we present some key issues and techniques for such a repository.

3.1 Challenges

A repository manages a large collection of "data objects", namely, Web pages. In that sense, it is conceptually quite similar to other systems that store and manage data objects (e.g., file systems and database systems). However, a Web repository does not have to provide a lot of the functionality that the other systems provide (e.g., transactions, logging, directory structure) and instead, can be targeted to address the following key challenges:

Scalability: It must be possible to seamlessly distribute the repository across a cluster of computers and disks, in order to cope with the size of the Web (see Section 1).

Dual access modes: The repository must support two different access modes equally efficiently. *Random access* is used to quickly retrieve a specific Web page, given the page's unique identifier. *Streaming access* is used to receive the entire collection, or some significant subset, as a stream of pages. Random access is used by the query engine to serve out cached copies to the end-user. Streaming access is used by the indexer and analysis modules to process and analyze pages in bulk.

Large bulk updates: Since the Web changes rapidly (see Section 1), the repository needs to handle a high rate of modifications. As new versions of Web pages are received from the crawler, the space occupied by old versions must be reclaimed[1] through space compaction and reorganization. In addition, excessive conflicts between the update process and the applications accessing pages must be avoided.

[1]Some repositories might maintain a temporal history of Web pages by storing multiple versions for each page. We do not consider this here.

Obsolete pages: In most file or data systems, objects are explicitly deleted when no longer needed. However, when a Web page is removed from a Web site, the repository is not notified. Thus, the repository must have a mechanism for detecting and removing obsolete pages.

3.2 Designing a distributed Web repository

Let us consider a generic Web repository that is designed to function over a cluster of interconnected *storage nodes*. There are three key issues that affect the characteristics and performance of such a repository:

- Page distribution across nodes
- Physical page organization within a node
- Update strategy

3.2.1 Page distribution policies

Pages can be assigned to nodes using a number of different policies. For example, with a *uniform distribution* policy, all nodes are treated identically. A given page can be assigned to any of the nodes in the system, independent of it's identifier. Nodes will store portions of the collection proportionate to their storage capacities. In contrast, with a *hash distribution policy*, allocation of pages to nodes would depend on the page identifiers. In this case, a page identifier would be hashed to yield a node identifier and the page would be allocated to the corresponding node. Various other policies are also possible. Reference [35] contains qualitative and quantitative comparisons of the uniform and hash distribution policies in the context of a Web repository.

3.2.2 Physical page organization methods

Within a single node, there are three possible operations that could be executed: *page addition/insertion*, *high-speed streaming*, and *random page access*. Physical page organization at each node is a key factor that determines how well a node supports each of these operations.

There are several options for page organization. For example, a hash-based organization treats a disk (or disks) as a set of hash buckets, each of which is small enough to fit in memory. Pages are assigned to hash buckets depending on their page identifier. Since page additions are common, a log-structured organization may also be advantageous. In this case, the entire disk is treated as a large contiguous log to which incoming pages are appended. Random access is supported using a separate B-tree index that maps page identifiers to physical locations on disk. One can also devise a hybrid hashed-log organization, where storage is divided into large sequential "extents," as opposed to buckets that fit in memory. Pages are hashed into extents, and each extent is organized like a log-structured

file. In [35] we compare these strategies in detail, and Table 1 summarizes the relative performance. Overall, a log-based scheme works very well, except if many random-access requests are expected.

	Log-structured	*Hash-based*	*Hashed-log*
Streaming performance	++	−	+
Random access performance	+−	++	+−
Page addition	++	−	+

Table 1: Comparing page organization methods

3.2.3 Update strategies

Since updates are generated by the crawler, the design of the update strategy for a Web repository depends on the characteristics of the crawler. In particular, there are at least two ways in which a crawler may be structured:

***Batch-mode or Steady crawler*:** A batch-mode crawler is periodically executed, say once every month, and allowed to crawl for a certain amount of time (or until a targeted set of pages have been crawled) and then stopped. With such a crawler, the repository receives updates only for a certain number of days every month. In contrast, a steady crawler runs without any pause, continuously supplying updates and new pages to the repository.

***Partial or Complete crawls*:** A batch-mode crawler may be configured to either perform a complete crawl every time it is run, or recrawl only a specific set of pages or sites. In the first case, pages from the new crawl completely replace the old collection of pages already existing in the repository. In the second case, the new collection is created by applying the updates of the partial crawl to the existing collection. Note that this distinction between partial and complete crawls does not make sense for steady crawlers.

Depending on these two factors, the repository can choose to implement either in-place update or shadowing. With in-place updates, pages received from the crawler are directly integrated into the repository's existing collection, possibly replacing older versions. With shadowing, pages from a crawl are stored separate from the existing collection and updates are applied in a separate step. As shown in Figure 5, the *read nodes* contain the existing collection and are used to service all random and streaming access requests. The *update nodes* store the set of pages retrieved during the latest crawl.

The most attractive characteristic of shadowing is the complete separation between updates and read accesses. A single storage node does not ever have to concurrently handle page addition and page retrieval. This avoids conflicts, leading to improved performance and a simpler implementation. On the downside, since there is a delay between the time a page is retrieved by the crawler and the time the page is available for access, shadowing may decrease collection freshness.

In our experience, a batch-mode crawler generating complete crawls matches well with a shadowing repository. Analogously, a steady crawler can be better matched with a repository that uses in-place updates. Furthermore, shadowing has a greater negative impact on freshness with a steady crawler than with a batch-mode crawler. For additional details, see [18].

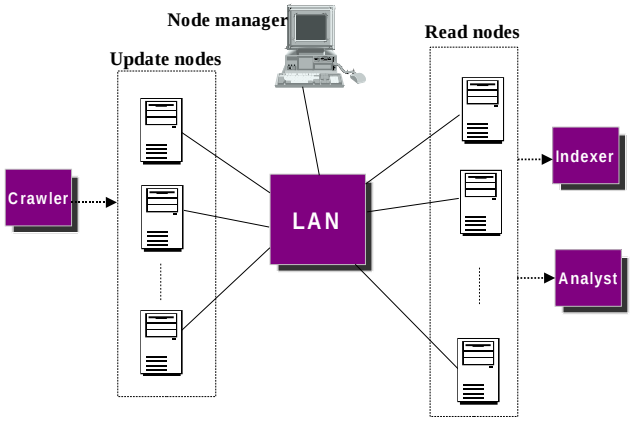

Figure 5: WebBase repository architecture

3.3 The Stanford WebBase repository

To illustrate how all the repository factors fit in, we briefly describe the Stanford WebBase repository, and the choices that were made. The WebBase repository is a distributed storage system that works in conjunction with the Stanford WebCrawler. The repository operates over a cluster of storage nodes connected by a high-speed communication network (see Figure 5). The repository employs a *node manager* to monitor the individual storage nodes and collect status information (such as free space, load, fragmentation, and number of outstanding access requests). The node manager uses this information to control the operations of the repository and schedule update and access requests on each node.

Each page in the repository is assigned a unique identifier, derived by computing a signature (e.g., checksum or cyclic redundancy check) of the URL associated with the page. However, a given URL can have multiple text string representations. For example, `http://WWW.STANFORD.EDU:80/` and `http://www.stanford.edu` represent the same Web page but would give rise to different signatures. To avoid this problem, the URL is first *normalized* to yield a canonical representation [35]. The identifier is computed as a signature of this normalized URL.

Since the Stanford WebCrawler is a batch-mode crawler, the WebBase repository employs the shadowing technique. It can be configured to use different page organization methods and distribution policies on the read nodes and update nodes. Hence, the update nodes can be tuned for optimal page addition performance and the read nodes, for optimal read performance.

3.4 Conclusion

The page repository is an important component of the overall Web search architecture. It must efficiently support the different access patterns of the query engine (random access) and the indexer modules (streaming access). It must also employ an update strategy that is tuned to the characteristics of the crawler.

In this section, we focused on the basic functionality required of a Web repository. However, there are a number of other features that might be useful for specific applications. Below, we suggest possible enhancements:

Multiple media types: So far, we have assumed that Web repositories store only plain text or HTML pages. However, with the growth of non-text content on the Web, it is becoming increasingly important to store, index, and search over image, audio, and video collections.

Advanced streaming: In our discussion of streaming access, we have assumed that the streaming order was not important. This is sufficient for building most basic indexes including the text and structure indexes of Figure 1. However, for more complicated indexes, the ability to retrieve a specific subset (e.g., pages in the ".edu" domain) and/or in a specified order (e.g., in increasing order by citation rank) may be useful.

4 Indexing

The *indexer* and *collection analysis* modules in Figure 1 build a variety of indexes on the collected pages. The indexer module builds two basic indexes: a text (or content) index and a structure (or link index). Using these two indexes and the pages in the repository, the *collection analysis* module builds a variety of other useful indexes. Below, we will present a short description of each type of index, concentrating on their structure and use.

Link index: To build a link index, the crawled portion of the Web is modeled as a graph with nodes and edges. Each node in the graph is a Web page and a directed edge from node A to node B represents a hypertext link in page A that points to page B. An index on the link structure must be a scalable and efficient representation of this graph.

Often, the most common structural information used by search algorithms [8, 38] is *neighborhood information*, i.e., given a page P, retrieve the set of pages pointed to by P (outward links) or the set of pages pointing to P (incoming links). Disk-based adjacency list representations [1] of the original Web graph and of the inverted Web graph[2] can efficiently provide access to such neighborhood information. Other structural properties of the Web graph can be easily derived from the basic information stored in

[2]In the inverted Web graph, the direction of the hypertext links are reversed.

these adjacency lists. For example, the notion of *sibling pages* is often used as the basis for retrieving pages "related" to a given page (see Section 5). Such sibling information can be easily derived from the pair of adjacency list structures described above.

Small graphs of hundreds or even thousands of nodes can be efficiently represented by any one of a variety of well-known data structures [1]. However, doing the same for a graph with several million nodes and edges is an engineering challenge. In [7], Bharath et. al. describe the *Connectivity Server*, a system to scalably deliver linkage information for all pages retrieved and indexed by the Altavista search engine.

Text index: Even though link-based techniques are used to enhance the quality and relevance of search results, text-based retrieval (i.e., searching for pages containing some keywords) continues to be the primary method for identifying pages relevant to a query. Indexes to support such text-based retrieval can be implemented using any of the access methods traditionally used to search over text document collections. Examples include *suffix arrays* [43], *inverted files or inverted indexes* [55, 59], and *signature files* [27]. Inverted indexes have traditionally been the index structure of choice on the Web. We will discuss inverted indexes in detail later on in the section.

Utility indexes: The number and type of utility indexes built by the collection analysis module depends on the features of the query engine and the type of information used by the ranking module. For example, a query engine that allows searches to be restricted to a specific site or domain (e.g., `www.stanford.edu`) would benefit from a *site index* that maps each domain name to a list of pages belonging to that domain. Similarly, using neighborhood information from the link index, an iterative algorithm (see Section 5) can easily compute and store the PageRank [8] associated with each page in the repository. Such an index would be used at query time to aid in the ranking of search results.

For the rest of this section, we will focus our attention on text indexes. In particular, we will address the problem of quickly and efficiently building inverted indexes over Web-scale collections.

4.1 Structure of an inverted index

An inverted index over a collection of Web pages consists of a set of *inverted lists*, one for each word (or *index term*). The inverted list for a term is a sorted list of *locations* where the term appears in the collection. In the simplest case, a *location* will consist of a page identifier and the position of the term in the page. However, search algorithms often make use of additional information about the occurrence of terms in a Web page. For example, terms occurring in bold face (within `<B>` tags), in section headings (within `<H1>` or `<H2>`tags), or as anchor text might be weighted differently in the ranking algorithms. To accommodate this, an additional *payload* field is added to the location entries. The payload field

encodes whatever additional information needs to be maintained about each term occurrence. Given an index term w, and a corresponding location l, we refer to the pair (w, l) as a *posting* for w.

In addition to the inverted lists, most text-indexes also maintain what is known as a *lexicon*. A lexicon lists all the terms occurring in the index along with some term-level statistics (e.g., total number of documents in which a term occurs) that are used by the ranking algorithms [55, 59].

4.2 Challenges

Conceptually, building an inverted index involves processing each page to extract postings, sorting the postings first on index terms and then on locations, and finally writing out the sorted postings as a collection of inverted lists on disk. For relatively small and static collections, as in the environments traditionally targeted by information retrieval (IR) systems, index build times are not very critical. However, when dealing with Web-scale collections, naive index build schemes become unmanageable and require huge resources, often taking days to complete. As a measure of comparison with traditional IR systems, our 40 million page WebBase repository (Section 3.3) represents only about 4% of the *publicly indexable Web* but is already larger than the 100 GB *very large TREC-7 collection* [34], the benchmark for large IR systems.

In addition, since content on the Web changes rapidly (see Section 1), periodic crawling and rebuilding of the index is necessary to maintain "freshness." Index rebuilds become necessary because most incremental index update techniques perform poorly when confronted with the huge wholesale changes commonly observed between successive crawls of the Web [46].

Finally, storage formats for the inverted index must be carefully designed. A small compressed index improves query performance by allowing large portions of the index to be cached in memory. However, there is a tradeoff between this performance gain and the corresponding decompression overhead at query time [47, 49, 59]. Achieving the right balance becomes extremely challenging when dealing with Web-scale collections.

4.3 Index partitioning

Building a Web-scale inverted index requires a highly scalable and distributed text-indexing architecture. In such an environment, there are two basic strategies for partitioning the inverted index across a collection of nodes [44, 53, 56].

In the *local inverted file (IF_L)* organization [53], each node is responsible for a disjoint subset of pages in the collection. A search query would be broadcast to all the nodes, each of which would return disjoint lists of page identifiers containing the search terms.

Global inverted file (IF_G) organization [53] partitions on index terms so that each query server stores inverted lists only for a subset of the terms in the collection. For example, in a system with two query servers A and B, A could store the inverted lists for all index terms that begin with characters in the

ranges *[a-q]* whereas B could store the inverted lists for the remaining index terms. Therefore, a search query that asks for pages containing the term *"process"* would only involve A.

Reference [45] describes certain important characteristics of the IF_L strategy, such as resilience to node failures and reduced network load, that make this organization ideal for the Web search environment. Performance studies in [56] also indicate that IF_L organization uses system resources effectively and provides good query throughput in most cases.

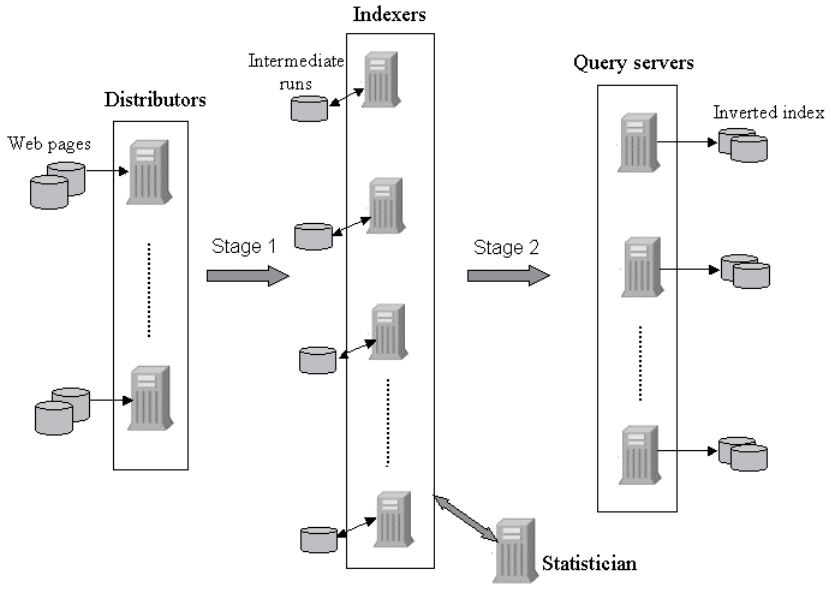

Figure 6: WebBase indexing architecture

4.4 WebBase Text-Indexing System

Our experience in building a text index for our WebBase repository serves to illustrate the problems faced in building a massive index. Indeed, our index was built to facilitate tests with different solutions, so we were able to obtain experimental results that show some of the tradeoffs. In the rest of this subsection we provide an overview of the WebBase index and the techniques utilized.

4.4.1 System overview

Our indexing system operates on a shared-nothing architecture consisting of a collection of nodes connected by a local area network (Figure 6). We identify three types of nodes in the system.[3] The *distributors* are part of the WebBase repository (Section 3) and store the collection of Web pages to be indexed. The *indexers* execute the core of the index building engine. The final inverted index is partitioned across the *query servers* using the IF_L strategy discussed in Section 4.3.

[3]We shall discuss the statistician later, in Section 4.4.3.

The WebBase indexing system builds the inverted index in two stages. In the first stage, each distributor node runs a *distributor process* that disseminates the pages to the indexers using the streaming access mode provided by the repository. Each indexer receives a mutually disjoint subset of pages and their associated identifiers. The indexers parse and extract postings from the pages, sort the postings in memory, and flush them to intermediate structures (*sorted runs*) on disk.

In the second stage, these intermediate structures are merged together to create one or more inverted files and their associated lexicons. An `(inverted file, lexicon)` pair is generated by merging a subset of the sorted runs. Each pair is transferred to one or more query servers depending on the degree of index replication.

4.4.2 Parallelizing the index-builder

The core of our indexing engine is the *index-builder* process that executes on each indexer. We demonstrate below that this process can be effectively parallelized by structuring it as a *software pipeline*.

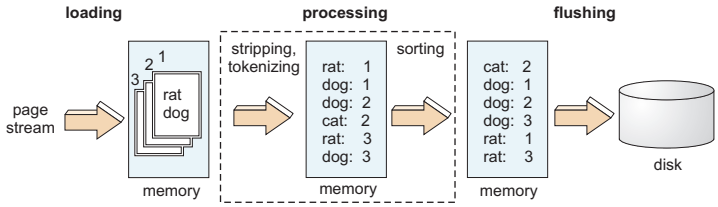

Figure 7: Logical phases

The input to the index-builder is a sequence of Web pages and their associated identifiers. The output of the index-builder is a set of *sorted runs*, each containing postings extracted from a subset of the pages. The process of generating these sorted runs can logically be split into three phases as illustrated in Figure 7. We refer to these phases as *loading*, *processing*, and *flushing*. During the loading phase, some number of pages are read from the input stream and stored in memory. The processing phase involves two steps. First, the pages are parsed, tokenized into individual terms, and stored as a set of postings in a memory buffer. In the second step, the postings are sorted in-place, first by term, and then by location. During the flushing phase, the sorted postings in the memory buffer are saved on disk as a sorted run. These three phases are executed repeatedly until the entire input stream of pages has been consumed.

Loading, processing and flushing tend to use disjoint sets of system resources. Processing is obviously CPU-intensive, whereas flushing primarily exerts secondary storage, and loading can be done directly from the network or a separate disk. Therefore, indexing performance can be improved by executing these three phases concurrently (see Figure 8). Since the execution order of loading, processing and flushing is fixed, these three phases together form a *software pipeline*.

The goal of our pipelining technique is to design an execution schedule for the different indexing

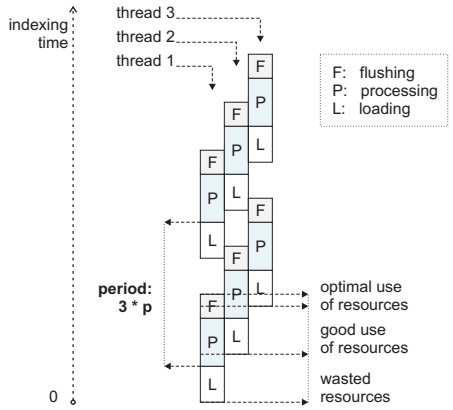

Figure 8: Multi-threaded execution of index-builder

phases that will result in minimal overall running time. Our problem differs from a typical *job scheduling* problem [12] in that we can vary the sizes of the incoming *jobs*, i.e., in every loading phase we can choose the number of pages to load.

Consider an index-builder that uses N executions of the pipeline to process the entire collection of pages and generate N sorted runs. By an *execution of the pipeline*, we refer to the sequence of three phases — loading, processing, and flushing — that transform some set of pages into a sorted run. Let B_i, $i = 1 \ldots N$, be the buffer sizes used during these N executions. The sum $\sum_{i=1}^{N} B_i = B_{total}$ is fixed for a given amount of input and represents the total size of all the postings extracted from the pages.

In [45], we show that for an indexer with a single resource of each type (single CPU, single disk, and a single network connection over which to receive the pages), optimal speedup of the pipeline is achieved when the buffer sizes are identical in all executions of the pipeline, i.e., $B = B_1 \ldots = B_N = \frac{B_{total}}{N}$. In addition, we show how bottleneck analysis can be used to derive an expression for the optimal value of B in terms of various system parameters. In [45], we also extend the model to incorporate indexers with multiple CPU's and disks.

Experiments: We conducted a number of experiments using a single index-builder to verify our model and measure the effective performance gain achieved through a parallelized index-builder. Figure 9 highlights the importance of the theoretical analysis as an aid in choosing the right buffer size. Even though the predicted optimum size differs slightly from the observed optimum, the difference in running times between the two sizes is less than 15 minutes for a 5 million page collection. Figure 10 shows how pipelining impacts the time taken to process and generate sorted runs for a variety of input sizes. Note that for small collections of pages, the performance gain through pipelining, though noticeable, is not substantial. This is because small collections require very few pipeline executions and the overall time is dominated by the time required at startup (to load up the buffers) and shutdown (to flush the buffers). Our experiments showed that in general, for large collections, a sequential index-builder is

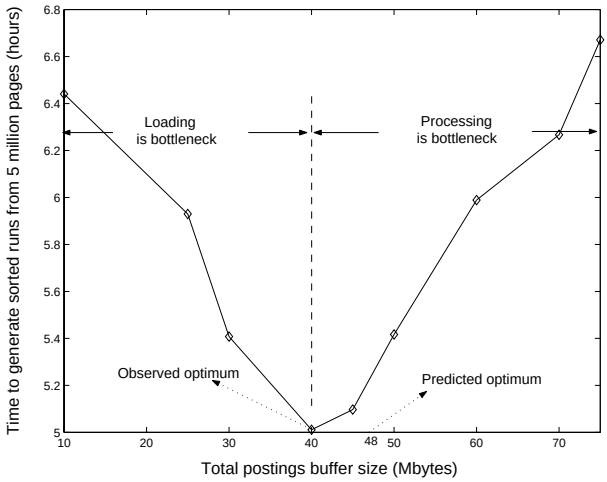

Figure 9: Optimal buffer size

about 30–40% slower than a pipelined index-builder.

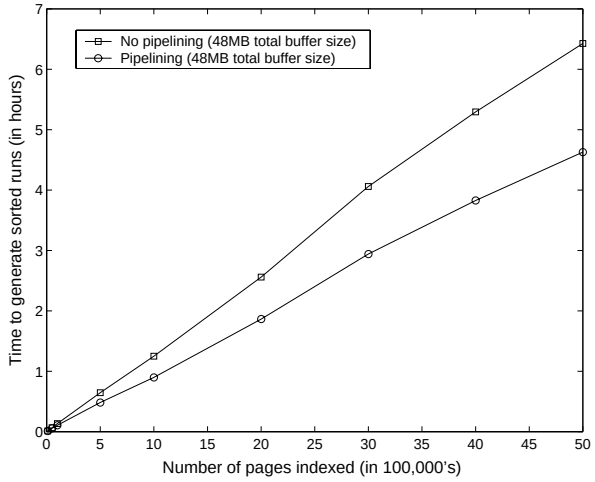

Figure 10: Performance gain through pipelining

4.4.3 Efficient global statistics collection

As mentioned in Section 4, term-level[4] statistics are often used to rank the search results of a query. For example, one of the most commonly used statistic is *inverse document frequency* or IDF. The IDF of a term w, is defined as $\log \frac{N}{df_w}$, where N is the total number of pages in the collection and df_w is the number of pages that contain at least one occurrence of w [55]. In a distributed indexing system, when the indexes are built and stored on a collection of machines, gathering global (i.e., collection-wide)

[4] *Term-level* refers to the fact that any gathered statistic describes only single terms, and not higher level entities such as pages or Web sites.

term-level statistics, with minimum overhead, becomes an important issue [57].

Some authors suggest computing global statistics at query time. This would require an extra round of communication among the query servers to exchange local statistics[5]. However, this communication adversely impacts query processing performance, especially for large collections spread over many servers.

Statistics-collection in WebBase: To avoid this query time overhead, the WebBase indexing system precomputes and stores statistics as part of index creation. A dedicated server known as the statistician (see Figure 6) is used for computing statistics. Local information from the indexers is sent to the statistician as the pages are processed. The statistician computes the global statistics and broadcasts them back to all the indexers. These global statistics are then integrated into the lexicons during the merging phase, when the sorted runs are merged to create inverted files (see Section 4.4.1). Two techniques are used to minimize the overhead of statistics collection.

- *Avoiding explicit I/O for statistics:* To avoid additional I/O, local data is sent to the statistician only when it is already available in memory. We have identified two phases in which this occurs: *flushing* — when sorted runs are written to disk, and *merging* — when sorted runs are merged to form inverted lists. This leads to the two strategies, *FL* and *ME*, described below.

- *Local aggregation:* In both FL and ME, postings occur in at least partially sorted order, meaning multiple postings for a term pass through memory in groups. Such groups are condensed into *(term, local aggregated information)* pairs which are sent to the statistician. For example, if the indexer's buffer has 1000 individual postings for the term "cat", then a single pair ("cat", 1000) can be sent to the statistician. This technique greatly reduces the communication overhead of collecting statistics.

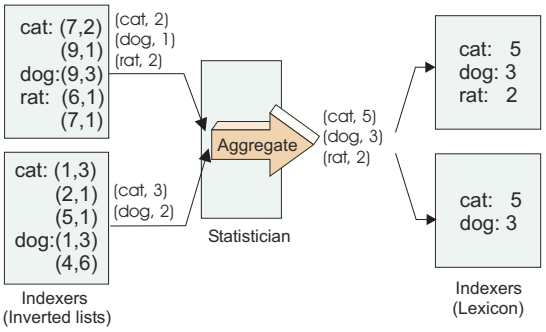

Figure 11: ME strategy

[5]By local statistics, we mean the statistics that a query server can deduce from the portion of the index that is stored on that node.

ME Strategy: sending local information during merging. Summaries for each term are aggregated as inverted lists are created in memory, and sent to the statistician. The statistician receives parallel sorted streams of *(term, local-aggregate-information)* values from each indexer and merges these streams by term, aggregating the sub-aggregates for each term. The resulting global statistics are then sent back to the indexers in sorted term order. This approach is entirely stream based, and does not require in-memory or on-disk data structures at the statistician or indexer to store intermediate results. However, the progress of each indexer is synchronized with that of the statistician, which in turn causes indexers to be synchronized with each other. As a result, the slowest indexer in the group becomes the bottleneck, holding back the progress of faster indexers.

Figure 11 illustrates the ME strategy for computing the value of df_w (i.e., the total number of documents containing w) for each term, w, in the collection. The figure shows two indexers with their associated set of postings. Each indexer aggregates the postings and sends local statistics (e.g., the pair (cat,2) indicates that the first indexer has seen 2 documents containing the word 'cat') to the statistician.

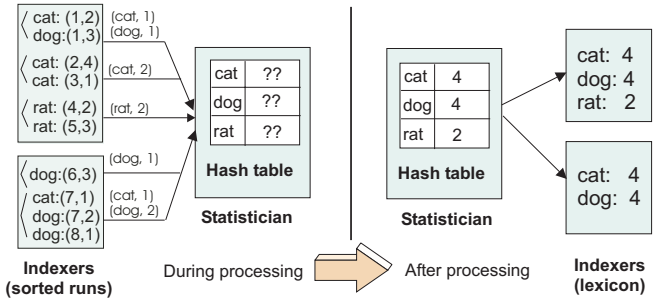

Figure 12: FL strategy

FL Strategy: sending local information during flushing. As sorted runs are flushed to disk, postings are summarized and the summaries sent to the statistician. Since sorted runs are accessed sequentially during processing, the statistician receives streams of summaries in globally *unsorted* order. To compute statistics from the unsorted streams, the statistician keeps an in-memory hash table of all terms and their related statistics, and updates the statistics as summaries for a term are received. At the end of the processing phase, the statistician sorts the statistics in memory and sends them back to the indexers. Figure 12 illustrates the FL strategy for collecting page frequency statistics.

Reference [46] presents and analyzes experiments that compare the relative overhead[6] of the two strategies for different collection sizes. Their studies show that the relative overheads of both strategies is acceptably small (less than 5% for a 2 million page collection) and exhibit sub-linear growth with

[6]Relative overhead of a strategy is given by $\frac{T_2 - T_1}{T_1}$, where T_2 is the time for full index creation with statistics collection using that strategy, and T_1 is the time for full index creation with *no* statistics collection

increase in collection size. This indicates that centralized statistics collection is feasible even for very large collections.

	Phase	Statistician load	Memory usage	Parallelism
ME	merging	+−	+	+−
FL	flushing	−	−	++

Table 2: Comparing strategies

Table 2 summarizes the characteristics of the FL and ME statistics gathering strategies.

4.5 Conclusion

The fundamental issue in indexing Web pages, when compared with indexing in traditional applications and systems, is scale. Instead of representing hundred or thousand node graphs, we need to represent graphs with millions of nodes and billions of edges. Instead of inverting 2 or 3 gigabyte collections with a few hundred thousand documents, we need to build inverted indexes over millions of pages and hundreds of gigabytes. This requires careful rethinking and redesign of traditional indexing architectures and data structures to achieve massive scalability.

In this section, we provided an overview of the different indexes that are normally used in a Web search service. We discussed how the fundamental structure and content indexes as well as other utility indexes (such as the PageRank index) fit into the overall search architecture. In this context, we illustrated some of the techniques that we have developed as part of the Stanford WebBase project to achieve high scalability in building inverted indexes.

There are a number of challenges that still need to be addressed. Techniques for incremental update of inverted indexes that can handle the massive rate of change of Web content are yet to be developed. As new indexes and ranking measures are invented, techniques to allow such measures to be computed over massive distributed collections need to be developed. At the other end of the spectrum, with the increasing importance of *personalization*, the ability to build some of these indexes and measures on a smaller scale (customized for individuals or small groups of users and using limited resources) also becomes important. For example, [33] discusses some techniques for efficiently evaluating PageRank on modestly equipped machines.

5 Ranking and Link Analysis

As shown in Figure 1, the Query Engine collects search terms from the user and retrieves pages that are likely to be relevant. As mentioned in Section 1, there are two main reasons why traditional Information Retrieval (IR) techniques may not be effective enough in ranking query results. First, the Web is very

large, with great variation in the amount, quality and the type of information present in Web pages. Thus, many pages that contain the search terms may be of poor quality or not relevant.

Second, many Web pages are not sufficiently self-descriptive, so the IR techniques that examine the contents of a page alone may not work well. An often cited example to illustrate this issue is the search for "search engines" [38]. The homepages of most of the principal search engines do not contain the text "search engine". Moreover, Web pages are frequently manipulated by adding misleading terms so they are ranked higher by a search engine (spamming). Thus, techniques that base their decisions on the content of pages alone are easy to manipulate.

The link structure of the Web contains important implied information, and can help in filtering or ranking Web pages. In particular, a link from page A to page B can be considered a recommendation of page B by the author of A. Some new algorithms have been proposed that exploit this link structure — not only for keyword searching, but also other tasks like automatically building a Yahoo-like hierarchy, or identifying communities on the Web. The qualitative performance of these algorithms is generally better than the IR algorithms since they make use of more information than just the contents of the pages. While it is indeed possible to influence the link structure of the Web locally, it is quite hard to do so at a global level. So link analysis algorithms that work at a *global level* are relatively robust against spamming.

The rest of this section describes two interesting link based techniques — PageRank and HITS. Some link based techniques for other tasks like page classification and identifying online communities are also discussed briefly.

5.1 PageRank

In [50], Page and Brin define a global ranking scheme, called PageRank, which tries to capture the notion of "importance" of a page. For instance, the Yahoo! homepage is intuitively more important than the homepage of the Stanford Database Group. This difference is reflected in the number of other pages that point to these two pages, that is, more pages point to the Yahoo homepage than to the Stanford Database group homepage. The rank of page A could thus be defined as the number of pages in the Web that point to A, and could be used to rank the results of a search query. (We used the same citation ranking scheme for $IB(P)$ in Section 2.) However, citation ranking does not work very well, especially against spamming, since it is quite easy to artificially create a lot of pages to point to a desired page.

PageRank extends the basic citation idea by taking into consideration the importance of the pages that point to a given page. Thus a page receives more importance if Yahoo points to it, than if some unknown page points to it. Citation ranking, in contrast, does not distinguish between these two cases. Note that the definition of PageRank is recursive — the importance of a page both *depends on* and *influences* the importance of other pages.

5.1.1 Simple PageRank

We first present a simple definition of PageRank that captures the above intuition, and discuss its computational aspects, before describing a practical variant. Let the pages on the Web be denoted by $1, 2, \ldots, m$. Let $N(i)$ denote the number of forward (outgoing) links from page i. Let $B(i)$ denote the set of pages that point to page i. For now, assume that the Web pages form a strongly connected graph (every page can be reached from any other page). (In Section 5.1.4 we discuss how we can relax this assumption.) The simple PageRank of page i, denoted by $r(i)$, is given by

$$r(i) = \sum_{j \in B(i)} r(j)/N(j).$$

The division by $N(j)$ captures the intuition that pages which point to page i evenly distribute their rank boost to all of the pages they point to. In the language of linear algebra [31], this can be written as $r = A^T r$, where r is the $m \times 1$ vector $[r(1), r(2), \ldots, r(m)]$, and the elements $a_{i,j}$ of the matrix A are given by, $a_{ij} = 1/N(i)$, if page i points to page j, and $a_{i,j} = 0$ otherwise. Thus the PageRank vector r is the eigenvector of matrix A^T corresponding to the eigenvalue 1. Since the graph is strongly connected, it can be shown that 1 is an eigenvalue of A^T, and the eigenvector r is uniquely defined (when a suitable normalization is performed and the vector is non-negative).

5.1.2 Random Surfer Model

The definition of simple PageRank lends itself to an interpretation based on random walks — called the *Random Surfer model* in [50]. Imagine a person who surfs the Web by randomly clicking links on the visited pages. This random surfing is equivalent to a random walk on the underlying link graph. The random walk problem is a well studied combinatorial problem [48]. It can be shown that the PageRank vector r is proportional to the stationary probability distribution of the random walk. Thus, the PageRank of a page is proportional to the frequency with which a random surfer would visit it.

5.1.3 Computation of PageRank

As indicated in Section 5.1.1, the PageRank computation is equivalent to computing the principal eigenvector of the matrix A^T defined above. One of the simplest methods of computing the principal eigenvector of a matrix is called *power iteration.*[7] In the power iteration, an arbitrary initial vector is repeatedly multiplied with the given matrix [31] until it converges to the principal eigenvector. The power iteration for PageRank computation is given below.

1. $s \leftarrow$ any random vector

[7]The power iteration is guaranteed to converge only if the graph is *aperiodic*. (A strongly connected directed graph is aperiodic if the greatest common divisor of the lengths of all closed walks is 1.) In practice, the Web is always aperiodic.

2. $r \leftarrow A^T \times s$

3. if $|| r - s ||< \epsilon$ end. r is the PageRank vector.

4. $s \leftarrow r$, goto 2

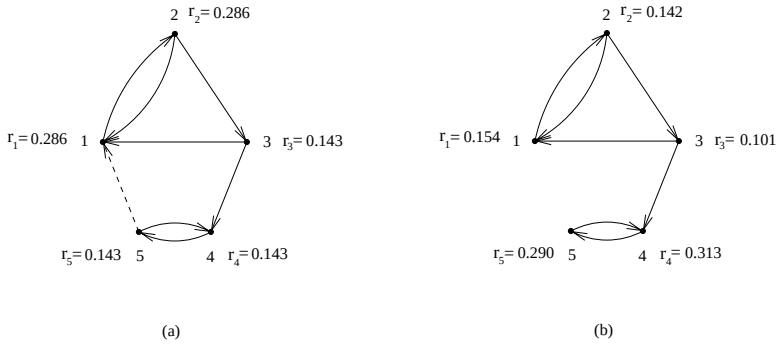

Figure 13: (a) Simple PageRank (b) Modified PageRank with $d = 0.8$

To illustrate, Figure 13(a) shows the PageRank for a simple graph. It is easy to verify that this assignment of ranks satisfies the definition of PageRank. For instance, node 2 has a rank of 0.286 and two outgoing links. Half of it's rank (0.143) flows to node 1 and half to node 3. Since node 3 has no other incoming links, its rank is what is received from node 2, i.e., 0.143. Node 1 receives 0.143 from 2, plus 0.143/2 from node 3, plus 0.143/2 from node 5, for a total of 0.286. Note that node 1 has a high rank because it has three incoming links. Node 2 has the same high rank because anyone who visits 1 will also visit 2. Also note that the ranks over all nodes add up to 1.0.

5.1.4 Practical PageRank

Simple PageRank is well defined only if the link graph is strongly connected. However, the Web is far from strongly connected (see Section 1). In particular, there are two related problems that arise on the real Web: rank sinks and rank leaks.

Any strongly (internally) connected cluster of pages within the Web graph from which no links point outwards forms a *rank sink*. An individual page that does not have any outlinks constitutes a *rank leak*. Although, technically, a *rank leak* is a special case of *rank sink*, a rank leak causes a different kind of problem. In the case of a rank sink, nodes not in the sink receive a zero rank, which means we cannot distinguish the importance of such nodes. For example, suppose that in Figure 13(a) we remove the $5 \rightarrow 1$ link, making nodes 4 and 5 a sink. A random surfer visiting this graph would eventually get stuck in nodes 4 and 5; i.e., nodes 1, 2 and 3 would have rank 0 (and nodes 4 and 5 would have rank 0.5). On the other hand, any rank reaching a *rank leak* is lost forever. For instance, in Figure 13(a), if we remove node 5 (and all links associated with it), node 4 becomes a leak. This leak causes all the

ranks to eventually converge to 0. That is, our random surfer would eventually reach node 4 and will never be seen again!

Page and Brin [50] suggest eliminating these problems in two ways. First, they remove all the leak nodes with out-degree 0.[8] Second, in order to solve the problem of sinks, they introduce a decay factor d $(0 < d < 1)$ in the PageRank definition. In this modified definition, only a fraction d of the rank of a page is distributed among the nodes that it points to. The remaining rank is distributed equally among all the pages on the Web. Thus, the modified PageRank is

$$r(i) = d * \sum_{j \in B(i)} r(j)/N(j) + (1-d)/m,$$

where m is the total number of nodes in the graph. Note that Simple PageRank (Section 5.1.1) is a special case that occurs when $d = 1$.

In the random surfer model, the modification models the surfer getting "bored" occasionally and making a jump to a random page on the Web (instead of following a random link from the current page). The decay factor d dictates how often the surfer gets bored.

Figure 13(b) shows the modified PageRank (for $d = 0.8$) for the graph of Figure 13(a) with the link $5 \rightarrow 1$ removed. Nodes 4 and 5 now have higher ranks than the other nodes, indicating that surfers will tend to gravitate to 4 and 5. However, the other nodes have non-zero ranks.

5.1.5 Computational Issues

In order for the power iteration to be practical, it is not only necessary that it converges to the PageRank, but also that it does so in a few iterations. Theoretically, the convergence of the power iteration for a matrix depends on the *eigenvalue gap*, which is defined as the difference between the modulus of the two largest eigenvalues of the given matrix. Page and Brin [50] claim that this is indeed the case, and that the power iteration converges reasonably fast (in around 100 iterations).

Also, note that we are more interested in the relative ordering of the pages induced by the PageRank (since this is used to rank the pages), than the actual PageRank values themselves. Thus we can terminate the power iteration once the ordering of the pages becomes reasonably stable. Experiments [33] indicate that the ordering induced by the PageRank converges much faster than the actual PageRank.

5.1.6 Using PageRank for keyword searching

In [10], Brin and Page describe a prototype search engine they developed at Stanford called Google (which has subsequently become the search engine Google.com [32]). Google uses both IR techniques and PageRank to answer keyword queries. Given a query, Google computes an IR score for all the pages

[8] Leak nodes will thus get no PageRank. An alternative would be to assume that leak nodes have links back to all the pages that point to them. This way, leak nodes that are reachable via high rank pages will have a higher rank than leak nodes reachable through unimportant pages.

that contain the query terms. The IR score is combined with the PageRank of these pages to determine the final rank for this query.[9]

We present a few top results from a search in Google for "java".

`java.sun.com`	*The source of Java Technology*
`javaboutique.internet.com`	*The Java Boutique*
`www.sun.com/java`	*Java Technology*
`www.java-pro.com`	*Java Pro*
`www.microsoft.com/java/default.htm`	*Home Page – Microsoft Technologies for Java*

5.2 HITS

In this section, we describe another important link based search algorithm, called HITS (Hypertext Induced Topic Search). This algorithm was first proposed by Kleinberg in [38]. In contrast to the PageRank technique, which assigns a global rank to every page, the HITS algorithm is a query dependent ranking technique. Moreover, instead of producing a single ranking score, the HITS algorithm produces two, the authority and the hub score.

Authority pages are those most likely to be relevant to a particular query. For instance, the Stanford University homepage is an authority page for the query "Stanford University", while some page that discusses the weather at Stanford would be less so. The *hub* pages are pages that are not necessarily authorities themselves but point to several authority pages. For instance, the page "searchenginewatch.com" is likely to be a good hub page for the query "search engine" since it points to several authorities, i.e., the homepages of search engines. There are two reasons why one might be interested in hub pages. First, these hub pages are used in the HITS algorithm to compute the authority pages. Second, the hub pages are themselves a useful set of pages to be returned to the user in response to the query [15].

How are the hub and authority pages related? The hub pages are ones that point to many authority pages. Conversely, one would also expect that if a page is a good authority, then it would be pointed to by many hubs. Thus there exists a *mutually reinforcing* relationship between the hubs and authorities: an authority page is a page that is pointed to by many hubs and hubs are pages that point to many authorities.[10] This intuition leads to the HITS algorithm.

5.2.1 The HITS Algorithm

The basic idea of the HITS algorithm is to identify a small subgraph of the Web and apply link analysis on this subgraph to locate the authorities and hubs for the given query. The subgraph that is chosen depends on the user query. The selection of a small subgraph (typically a few thousand pages), not

[9] Google also uses other text-based techniques to enhance the quality of results. For example, anchor text is considered part of the page pointed at. That is, if a link in page A with anchor "Stanford University" points to page B, the text "Stanford University" will be considered part of B and may even be weighted more heavily than the actual text in B.

[10] Interestingly, mutually reinforcing relationships have been identified and exploited for other Web tasks, see for instance [9].

only focuses the link analysis on the most relevant part of the Web, but also reduces the amount of work for the next phase. (Since both the subgraph selection and its analysis are done at query time, it is important to complete them quickly.)

Identifying the focused subgraph

The *focused subgraph* is generated by forming a root set R - a random set of pages containing the given query string, and expanding the root set to include the pages that are in the "neighborhood" of R. The text index (Section 4) can be used to construct the root set.[11] The algorithm for computing the focused subgraph is as follows:

1. $R \leftarrow$ set of t pages that contain the query terms (using the text index).
2. $S \leftarrow R$.
3. for each page $p \in R$,
   (a) Include all the pages that p points to in S.
   (b) Include (up to a maximum d) all pages that point to p in S.
4. The graph induced by S is the focused subgraph.

The algorithm takes as input the query string and two parameters t and d. Parameter t limits the size of the root set, while parameter d limits the number of pages added to the focused subgraph. (This latter control limits the influence of an extremely popular page like `www.yahoo.com` if it were to appear in the root set.[12] The expanded set S should be rich in authorities since it is likely that an authority is pointed to by at least some page in the root set. Likewise, a lot of good hubs are also likely to be included in S.

Link Analysis

The link analysis phase of the HITS algorithm uses the mutually reinforcing property to identify the hubs and authorities from the expanded set S. (Note that this phase is oblivious to the query that was used to derive S.) Let the pages in the focused subgraph S be denoted as $1, 2, \ldots, n$. Let $B(i)$ denote the set of pages that point to page i. Let $F(i)$ denote the set of pages that the page i points to. The link analysis algorithm produces an authority score a_i and a hub score h_i for each page in set S. To begin with, the authority scores and the hub scores are initialized to arbitrary values. The algorithm is an iterative one and it performs two kinds of operations in each step, called I and O. In the I operation,

[11] In [38], Altavista is used to construct the root set in the absence of a local text index.

[12] Several heuristics can also be used to eliminate non-useful links in the focused subgraph. The reader is referred to [38] for more details.

the authority score of each page is updated to the sum of the hub scores of all pages pointing to it. In the O step, the hub score of each page is updated to the sum of authority scores of all pages that it points to. That is,

$$\text{I step:} \quad a_i = \sum_{j \in B(i)} h_j$$

$$\text{O step:} \quad h_i = \sum_{j \in F(i)} a_j$$

The I and the O steps capture the intuition that a good authority is pointed to by many good hubs and a good hub points to many good authorities. Note incidentally that a page can be, and often is, both a hub and an authority. The HITS algorithm just computes two scores for each page, the hub score and the authority score. The algorithm iteratively repeats the I and O steps, with normalization, until the hub and authority scores converge:

1. Initialize a_i, h_i $(1 \leq i \leq n)$ to arbitrary values.
2. Repeat until convergence,
   (a) Apply the I operation.
   (b) Apply the O operation.
   (c) Normalize $\sum_i {a_i}^2 = 1$ and $\sum_i {h_i}^2 = 1$.
3. End.

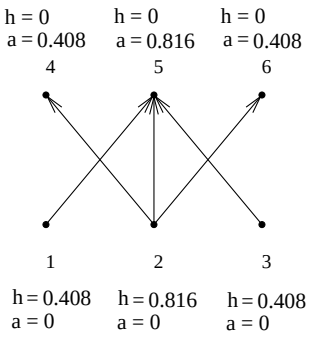

Figure 14: HITS Algorithm

An example of hub and authority calculation is shown in Figure 14. For example, the authority score of node 5 can be obtained by adding the hub scores of nodes that point to 5 (i.e., $0.408 + 0.816 + 0.408$) and normalizing this value by the combined hub scores (i.e., $(0.408)^2 + (0.816)^2 + (0.408)^2$.

The hub and authority values have interesting mathematical properties just as in the case of PageRank. Let $A_{m \times m}$ be the adjacency matrix of the focused subraph. The $(i, j)^{th}$ entry of A equals 1 if

page i points to page j, and 0 otherwise. Let a be the vector of authority scores $[a_1, a_2, \ldots, a_n]$, and h the vector of hub scores $[h_1, h_2, \ldots, h_n]$. Then the I and O operations can be expressed as, $a = Ah$ and $h = A^T a$ respectively. A simple substitution shows that the final converged hub and authority scores satisfy $a = c_1 AA^T a$ and $h = c_2 A^T Ah$ (the constants being added to account for the normalizing step). Thus the vector of authority scores and the vector of hub scores are the eigenvectors of the matrices AA^T and $A^T A$ respectively. The link analysis algorithm presented above is a simple power iteration that multiplies the vectors a and h by AA^T and $A^T A$ respectively without explicitly materializing them. Thus the vectors a and h are the principal eigenvectors of AA^T and $A^T A$ respectively. Just as in the case of PageRank, the rate of convergence of the hub and authority scores depends on the eigenvalue gap. The order of hubs and authority pages induced by the hub scores and the authority scores converges much faster (around 10 iterations) than the actual values themselves.

Interestingly, the idea of using the hubs to identify the authorities does not have a close analog in other hyperlinked research communities like bibliometrics [26, 29, 51]. Kleinberg et al [38] argue that in bibliometrics one authority typically acknowledges the existence of other authorities. But the situation is very different in the case of the Web. One would scarcely expect the homepage of Toyota to point to the homepage of Honda. However, there are likely to be many pages (hubs) that point to both Honda and Toyota homepages, and these hub pages can be used to identify that Honda and Toyota homepages are authorities in the same topic, say "automobile companies".

We conclude our discussion of HITS algorithm with a sample result for the query "java", borrowed from [38].

Authorities

`www.gamelan.com/`	*Gamelan*
`java.sun.com/`	*Javasoft homepage*
`www.digitalfocus.com/digitalfocus/faq/howdoi.html`	*Java Developer: How do I ...*
`lightyear.ncsu.uiuc.edu/ srp/java/javabooks.html`	*The Java Book Pages*
`sunsite.unc.edu/javafaq/javafaq.html`	*comp.lang.java FAQ*

5.3 Other Link Based Techniques

We have looked at two different link based algorithms that can be used for supporting keyword based querying. Similar link based techniques have been proposed for other tasks on the Web. We briefly mention some of these.

Identifying communities

The Web has a lot of online *communities* — a set of pages created and used by people sharing a common interest. For instance, there is a set of pages devoted to *database research*, forming the database research community. There are thousands of such communities on the Web ranging from the most exotic to the mundane. An interesting problem is to identify and study the nature and evolution of the online

communities — not only are these communities useful information resources for people with matching interests, but also because such a study sheds light on the sociological evolution of the Web. Gibson, Kleinberg and Raghavan [30] observe that the communities are characterized by a central "core" of hub and authority pages. Kumar et. al. [40] note that this dense, nearly bipartite graph induced by the edges between the hubs and the authorities is very likely to contain a bipartite core — a completely connected bipartite graph. They have discovered over 100,000 online communities by enumerating such bipartite cores, a process which they call *trawling*.

Finding Related Pages

A problem with keyword searching is that the user has to formulate his query using keywords, and may have difficulty in coming up with the right words to describe his interest. Search engines can also support a different kind of query called the *find related* query, or query by example. In this kind of query, the user gives an instance, typically a Web page, of the kind of information that he is seeking. The search engine retrieves other Web pages that are similar to the given page. This kind of query is supported, for instance, by Google [32] and Netscape's *What's Related* service.

A find-related query could be answered with traditional IR techniques based on text similarity. However, on the Web we can again exploit link structure. The basic idea is that of *cocitation*. If a page A points to two pages B and C, then it is more likely that B and C are related. Note that the HITS algorithm implicitly uses cocitation information (a hub "cocites" authorities). Kleinberg [38] suggests that a simple modification of the HITS algorithm can be used to support the *find related* query. Dean and Henzinger [22] propose two algorithms for *find related query*, namely the *Companion* algorithm which extends the Kleinberg's idea, and the *Cocitation* algorithm which uses a simple cocitation count to determine related pages. They find that both these link based algorithms perform better than Netscape's service, with the *Companion* algorithm being the better of the two.

Classification and Resource Compilation

Many search portals like Yahoo! and Altavista have a hierarchical classification of (a subset of) Web pages. The user can browse the hierarchy to locate the information that he is interested in. These hierarchies are typically manually compiled. The problem of automatically classifying documents based on some example classifications has been well studied in IR. Chakrabarti, Dom and Indyk [14] have extended the IR techniques to include hyperlink information. The basic idea is to use not only the words in the document but also the categories of the neighboring pages for classification. They have shown experimentally that their technique enhances the accuracy of classification.

A related problem that has been studied is that of *automatic resource compilation* [15, 13]. The emphasis in this case is to identify high quality pages for a particular category or topic. The algorithms that have been suggested are variations of the HITS algorithm that make use of the content information

in addition to the links.

5.4 Conclusion and Future Directions

The link structure of the Web contains a lot of useful information that can be harnessed to support keyword searching and other information retrieval tasks on the Web. We discussed two interesting link analysis techniques in this section. PageRank is a global ranking scheme that can be used to rank search results. The HITS algorithm identifies, for a given search query, a set of *authority* pages and a set of *hub* pages. A complete performance evaluation of these link based techniques is beyond the scope of this paper. The interested reader is referred to [3] for a comparison of the performance of different link based ranking techniques.

There are many interesting research directions to be explored. One promising direction is to study how other sources of information, like query logs, and click streams can be used for improving Web searching. Another important research direction is to study more sophisticated text analysis techniques (such as Latent Semantic Indexing [25]) and explore enhancements for these in a hyperlinked setting.

6 Conclusion

Searching the World-Wide Web successfully is the basis for many of our information tasks today. Search engines are thus increasingly being relied upon to extract just the right information from a vast number of Web pages. The engines are being asked to accomplish this task with minimal input from users, usually just one or two keywords.

In Figure 1 we have shown how such engines are put together. Several main functional blocks make up the typical architecture. Crawlers travel the Web, retrieving pages. These pages are stored locally, at least until they can be indexed and analyzed. A query engine then retrieves URLs that seem relevant to user queries. A ranking module attempts to sort these returned URLs such that the most promising results are presented to the user first.

This simple set of mechanisms requires a substantial underlying design effort. Much of the design and implementation complexity stems from the Web's vast scale. We explained how crawlers, with their limited resources in bandwidth and storage must use heuristics to ensure that the most desirable pages are visited, and that the search engine's knowledge of existing Web pages stays current.

We have shown how the large-scale storage of Web pages in search engines must be organized to match a search engine's crawling strategies. Such local Web page repositories must also enable users to access pages randomly, and to have the entire collection streamed to them.

The indexing process, while studied extensively for smaller, more homogeneous collections, requires new thinking when applied to the many millions of Web pages that search engines must examine. We discussed how indexing can be parallelized, and how needed statistics can be computed during the indexing process.

Fortunately, the interlinked nature of the Web offers special opportunities for enhancing search engine performance. We introduced the notion of PageRank, a variant of the traditional citation count. The Web's link graph is analyzed, and the number of links pointing to a page is taken as an indicator of that page's value. The HITS algorithm, or 'Hubs and Authorities' is another technique that takes advantage of Web linkage. This algorithm classifies the Web into pages that primarily function as major information sources for particular topics (authority pages), and other pages, which primarily point readers to authority pages (hubs). Both PageRank and HITS are used to boost search selectivity by identifying 'important' pages through link analysis.

A substantial amount of work remains to be accomplished, as search engines hustle to keep up with the ever expanding Web. New media, such as images and video, pose new challenges for search and storage. We have offered an introduction into current search engine technologies, and have pointed to several upcoming new directions.

Acknowledgement. We would like to thank Gene Golub and the referees for many useful suggestions.

References

[1] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[2] Reka Albert, Albert-Laszlo Barabasi, and Hawoong Jeong. Diameter of the World Wide Web. *Nature*, 401(6749), September 1999.

[3] B. Amento, L. Terveen, and W. Hill. Does authority mean quality? Predicting expert quality ratings of web documents. In *Proceedings of the Twenty-Third Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2000.

[4] 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*, 2000.

[5] Albert-Laszlo Barabasi and Reka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, October 1999.

[6] 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*, 1999.

[7] Krishna Bharat, Andrei Broder, Monika Henzinger, Puneet Kumar, and Suresh Venkatasubramanian. The connectivity server: Fast access to linkage information on the web. In *Proceedings of the Seventh International World-Wide Web Conference*, April 1998.

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

[9] Sergey Brin. Extracting patterns and relations from the world wide web. In *WebDB Workshop at 6th International Conference on Extending Database Technology, EDBT'98*, 1998. Available at http://www-db.stanford.edu/ sergey/extract.ps.

[10] 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*, 1998.

[11] 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*, 2000.

[12] S. Chakrabarti and S. Muthukrishnan. Resource scheduling for parallel database and scientific applications. In *8th ACM Symposium on Parallel Algorithms and Architectures*, pages 329–335, June 1996.

[13] Soumen Chakrabarti, Byron Dom, David Gibson, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Spectral filtering for resource discovery. In *ACM SIGIR workshop on Hypertext Information Retrieval on the Web*, 1998.

[14] Soumen Chakrabarti, Byron Dom, and Piotr Indyk. Enhanced hypertext categorization using hyperlinks. In *Proceedings of the International Conference on Management of Data*, 1998.

[15] Soumen Chakrabarti, Byron Dom, Prabhakar Raghavan, Sridhar Rajagopalan, David Gibson, and Jon Kleinberg. Automatic resource compilation by analyzing hyperlink structure and associated text. In *Proceedings of the Seventh International World-Wide Web Conference*, 1998.

[16] 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*, 1999.

[17] Junghoo Cho and Hector Garcia-Molina. Estimating frequency of change. In *Submitted for publication*, 2000.

[18] 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*, 2000. Available at http://www-diglib.stanford.edu/cgi-bin/get/SIDL-WP-1999-0129.

[19] Junghoo Cho and Hector Garcia-Molina. Synchronizing a database to improve freshness. In *Proceedings of the International Conference on Management of Data*, 2000. Available at http://www-diglib.stanford.edu/cgi-bin/get/SIDL-WP-1999-0116.

[20] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through url ordering. In *Proceedings of the Seventh International World-Wide Web Conference*, 1998. Available at http://www-diglib.stanford.edu/cgi-bin/WP/get/SIDL-WP-1999-0103.

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

[22] Jeffrey Dean and Monika R. Henzinger. Finding related pages in the world wide web. In *Proceedings of the Eighth International World-Wide Web Conference*, 1999.

[23] M. Diligenti, F. M. Coetzee, S. Lawrence, C. L. Giles, and M. Gori. Focused crawling using context graphs. In *Proceedings of the Twenty-sixth International Conference on Very Large Databases*, 2000.

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

[25] Susan T. Dumais, George W. Furnas, Thomas K. Landauer, Scott Deerwester, and Richard Harshman. Using latent semantic analysis to improve access to textual information. In *Proceedings of the Conference on Human Factors in Computing Systems CHI'88*, 1988.

[26] L. Egghe and R. Rousseau. *Introduction to Informetrics*. Elsevier, 1990.

[27] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Office Information Systems*, 2(4):267–288, October 1984.

[28] Christos Faloutsos. Access methods for text. *ACM Computing Surveys*, 17(1):49–74, March 1985.

[29] E. Garfield. Citation analysis as a tool in journal evaluation. *Science*, 178:471–479, 1972.

[30] David Gibson, Jon M. Kleinberg, and Prabhakar Raghavan. Inferring web communities from link topology. In *HyperText*, 1998.

[31] G. Golub and C. Van Loan. *Matrix Computations*. John Hopkins Press, 1989.

[32] Google inc. `http://www.google.com`.

[33] Taher Haveliwala. Efficient computation of pagerank. Technical Report 1999-31, Database Group, Computer Science Department, Stanford University, February 1999. Available at `http://dbpubs.stanford.edu/pub/1999-31`.

[34] D. Hawking and N. Craswell. Overview of TREC-7 very large collection track. In *Proc. of the Seventh Text Retrieval Conf.*, pages 91–104, November 1998.

[35] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. Webbase: A repository of web pages. In *Proceedings of the Ninth International World-Wide Web Conference*, pages 277–293, May 2000. Available at http://www-diglib.stanford.edu/cgi-bin/get/SIDL-WP-1999-0124.

[36] Bernardo A. Huberman and Lada A. Adamic. Growth dynamics of the World-Wide Web. *Nature*, 401(6749), September 1999.

[37] The internet archive. `http://www.archive.org/`.

[38] Jon Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, November 1999.

[39] Martijn Koster. Robots in the web: threat or treat? *ConneXions*, 4(4), April 1995.

[40] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Trawling the web for emerging cyber-communities. In *Proceedings of the Eighth International World-Wide Web Conference*, 1999.

[41] Steve Lawrence and C. Lee Giles. Searching the World Wide Web. *Science*, 280(5360):98, 1998.

[42] Steve Lawrence and C. Lee Giles. Accessibility of information on the web. *Nature*, 400:107–109, 1999.

[43] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *Proc. of the 1st ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1990.

[44] Patrick Martin, Ian A. Macleod, and Brent Nordin. A design of a distributed full text retrieval system. In *Proceedings of the Ninth International Conference on Research and Development in Information Retrieval*, pages 131–137, September 1986.

[45] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the web. Technical Report SIDL-WP-2000-0140, Stanford Digital Library Project, Computer Science Department, Stanford University, July 2000. Available at http://www-diglib.stanford.edu/cgi-bin/get/SIDL-WP-2000-0140.

[46] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the web. In *Proceedings of the Tenth International World-Wide Web Conference*, 2001.

[47] A. Moffat and T. Bell. In situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537–550, 1995.

[48] Rajeev Motwani and Prabakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[49] Anh NgocVo and Alistair Moffat. Compressed inverted files with reduced decoding overheads. In *Proceedings of the Twenty-First Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 290–297, August 1998.

[50] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Computer Science Department, Stanford University, 1998.

[51] G. Pinski and F. Narin. Citation influence for journal aggregates of scientific publications: Theory, with application to the literature of physics. *Inf. Proc. and Management*, 12, 1976.

[52] 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*, 1997.

[53] B. Ribeiro-Neto and R. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the Third ACM International Conference on Digital Libraries*, pages 182–190, June 1998.

[54] Robots exclusion protocol. `http://info.webcrawler.com/mak/projects/robots/exclusion.html`.

[55] Gerard Salton. *Automatic Text Processing*. Addison-Wesley, Reading, Mass., 1989.

[56] Anthony Tomasic and Hector Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proceedings of 2nd International Conference on Parallel and Distributed Information Systems*, pages 8–17, January 1993.

[57] Charles L. Viles and James C. French. Dissemination of collection wide information in a distributed information retrieval system. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 12–20, July 1995.

[58] 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*, 1999.

[59] Ian H. Witten. *Managing gigabytes : compressing and indexing documents and images*. Van Nostrand Reinhold, New York, 1994.