Stanford WebBase Components and Applications

JUNGHOO CHO, HECTOR GARCIA-MOLINA, TAHER HAVELIWALA, WANG LAM,
ANDREAS PAEPCKE, SRIRAM RAGHAVAN, and GARY WESLEY
Stanford University

We describe the design and performance of WebBase, a tool for Web research. The system includes a highly customizable crawler, a repository for collected Web pages, an indexer for both text and link-related page features, and a high-speed content distribution facility. The distribution module enables researchers world-wide to retrieve pages from WebBase, and stream them across the Internet at high speed. The advantage for the researchers is that they need not all crawl the Web before beginning their research. WebBase has been used by scores of research and teaching organizations world-wide, mostly for investigations into Web topology and linguistic content analysis. After describing the system's architecture, we explain our engineering decisions for each of the WebBase components, and present respective performance measurements.

Categories and Subject Descriptors: H.3.5 [**INFORMATION STORAGE AND RETRIEVAL**]: Online Information Services—*Web-based services*; *Data sharing*; H.5.3 [**INFORMATION INTERFACES AND PRESENTATION**]: Group and Organization Interfaces—*WEB*; H.3.4 [**INFORMATION STORAGE AND RETRIEVAL**]: Systems and Software—*Distributed systems*; *Performance evaluation (efficiency and effectiveness)*; H.3.1 [**INFORMATION STORAGE AND RETRIEVAL**]: Content Analysis and Indexing—*Indexing methods*; H.2 [**DATABASE MANAGEMENT**]: Systems—*Textual databases*; H.3.2 [**INFORMATION STORAGE AND RETRIEVAL**]: Information Storage—*File organization*; H.3.7 [**INFORMATION STORAGE AND RETRIEVAL**]: Digital Libraries; H.4.m [**INFORMATION SYSTEMS APPLICATIONS**]: Miscellaneous

General Terms: Design; Documentation; Experimentation; Measurement; Performance
Additional Key Words and Phrases: WebBase Web crawler, site crawling; hyperlink indexing; distribution

1. INTRODUCTION

WebBase is an experimental Web repository built at Stanford University. The system collects a sizeable portion of the Web at regular intervals, stores a local copy, constructs a number of indexes over the collected content, and then makes the content available for rapid streaming over the Internet. WebBase is a continuation of the original Google repository and search engine. The commercial version of Google forked off in the late 1990's, and the academic version was renamed WebBase. Unlike the commercial Google (and all commercial search engines), WebBase is a system open to experimentation, and its content is available to researchers outside Stanford.[1]

In this paper we describe the WebBase system, and present its main engineering and design tradeoffs. We present

[1] This work was supported under NSF Grant CS98-92A DLI2.

Current addresses: Junghoo Cho, University of California, Los Angeles Taher Haveliwala, Google Inc.; and Sriram Raghavan, IBM Almaden Research Center

experimental results that support the design decisions we made for some components, and in addition, present overall performance results for the system as a whole.

While members of our group have written papers on various aspects of Web repositories (gathering the data, indexing it, etc.; see related work in Section 5), this paper is the first paper that describes the running WebBase system itself, and the lessons learned. Our earlier papers used data gathered by WebBase for experiments, or studied algorithms in a general context. This paper, on the other hand, describes the actual implementation and its specific performance.

We start by giving a general overview of WebBase. WebBase includes a (i) Crawler Module, (ii) a distributed Indexing Engine that creates both text and link-related indexes, (iii) a Storage Module that stores the crawled pages, (iv) a Query Engine, and (v) a high-volume Web-page Distribution Module.

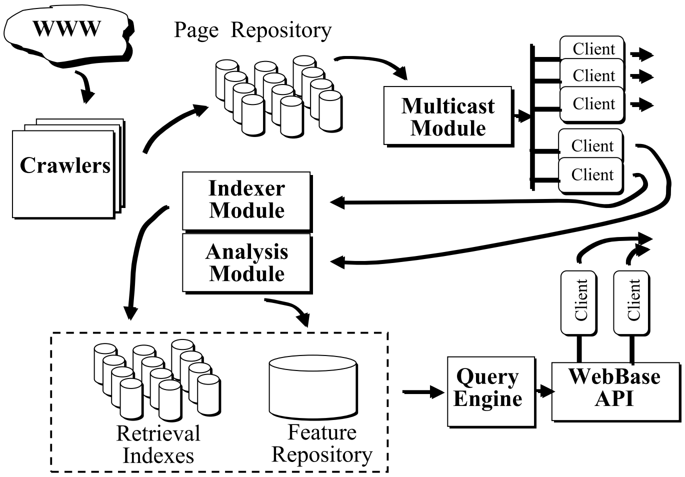

Fig. 1. WebBase Architecture

Figure 1 shows the WebBase architecture. Multiple **crawlers** retrieve HTML text pages or other files from the Web and deposit them in the **Page Repository**. (The crawlers are described in Section 2.) From there the pages are routed on demand to the **Multicast Module**, which manages the stream distribution. While this module currently establishes point-to-point links to deliver its feeds, the intent is eventually to dynamically combine destinations with similar stream content into multicast networks that can be efficiently serviced through a single stream.

As we will discuss in more detail in Section 4, the clients of the Multicast Module can be remote. This feature makes it possible for researchers worldwide to obtain WebBase data, and use it in the development of new search and analysis algorithms. Using WebBase saves researchers the tedious, time consuming, and expensive process of collecting Web data on their own, allowing them to focus on their research instead. WebBase has already been used in this mode for many purposes, including large-scale link-distribution analysis; the preservation of government Web pages (which are highly ephemeral); project-based teaching of Web-search technology; and the construction of vocabulary usage distribution statistics for Web materials.

Note that the lower two clients of the Multicast Module feed the repository contents right back into the core of WebBase, namely to the **Indexer Module** and the **Analysis Module**. This mechanism simplifies the design, since there is only one way that content is extracted from the repository, even for indexing.

The Indexer Module processes every collected page and constructs an inverted index. This index allows the system to quickly find all the pages in its repository that contain some given word. Such an operation is, of course, needed to query the repository for pages that contain keywords of interest.

The Analysis Module provides for indexing needs that are not necessarily related to words. For example, this module builds an index that enables fast determination of which Web pages contain links to some given page. This module is extensible. The intent is to allow the insertion of new index types. For example, after the first prototype was completed, we added a facility to the Analysis Module that computes the degree of similarity among Web pages, based on their common ancestry. The scheme identifies the graph 'siblings' of a given page and thereby helps identify pages that might be topically related.

The WebBase API is an interface that allows clients to submit queries to the indexes, much as one would query a database. The facility differs from such more standard query servers in that WebBase queries are intended to allow unified search over both the text and feature indexes. The intent is, for example, to solve queries of the form 'retrieve from the Web Repository all pages that contain the word "chair" and are linked to by at least 10 other pages' ([Raghavan 2003][2]). A separate, much simpler API provides access to the Multicast Module (top right of Figure 1). Clients of this 'wholesale' API request large streams of Web resources to be transmitted to them from the Page Repository. We describe this interface in Section 4.

This work presents WebBase's engineering and design tradeoffs. Each of the following sections explains the associated design problems and our choice of solutions for WebBase. We also present for each module the results of performance measurements that support our ultimate design decisions.

2. CRAWLER

The Web crawler module is responsible for downloading pages from the Web and storing them compressed in the repository. The design goal is to have this module collect as many Web pages as possible as quickly as computational and network resources allow. The challenge is to achieve this goal without violating any formal and informal rules of crawler conduct that have developed for the Web over time. We first describe the typical operation of a general Web crawler below; then, in the following sections, we explain our particular crawler's architecture, describe tradeoffs that impact our design goal, and evaluate the performance of our crawler.

Figure 2 shows a crawler's conceptual operation. The thin arrows in the diagram show data flow, and the thick arrows represent the operational procedures that the crawler follows. A crawler maintains two data structures to manage page URLs: URLsToVisit and VisitedURLs. VisitedURLs maintains the list of downloaded pages. The list is necessary for the crawler to avoid downloading a page multiple times in a single crawl. The URLsToVisit queue contains the list of pages to download in the future.

The initial contents of the URLsToVisit queue is called a *seed list*. This list is expanded over time. Before the crawler can run for the first time the seed URLs are specified manually, or obtained from some source. The initial list might cover a diverse set of Web sites, or a crawler can be biased towards particular interests using a more focused

[2] WebBase does not export this functionality.

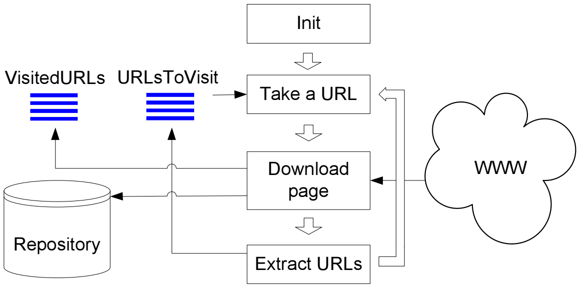

Fig. 2. Conceptual operation of a Web crawler

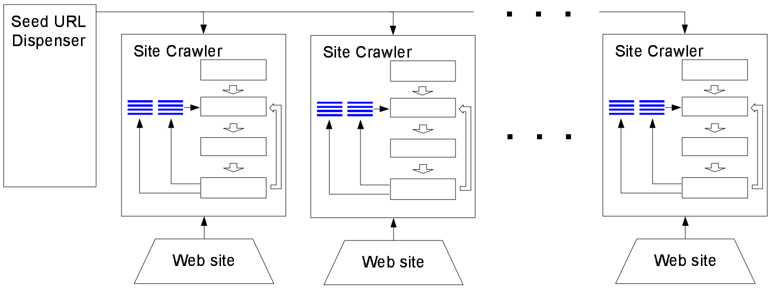

Fig. 3. Multiple site crawlers (conceptual)

selection of URLs. We began with a collection of well-known URLs. As we conducted additional crawls, we added newly found URLs to the seed list.

The crawler loops through taking a URL from the URLsToVisit queue, downloading the corresponding page, storing the page in the repository, and putting the URL into the VisitedURLs queue. Conceptually, the crawler extracts all hyperlinks from each downloaded page, converts relative links (URLs) into absolute URLs, and checks whether any of the new URLs has already been downloaded. If a URL has already been fetched, it is discarded. If it has not, then the crawler adds the URL to the URLsToVisit queue. This process is repeated until the crawler has exhausted the URLsToVisit queue, or is stopped by an operator.

If the fetch of a URL fails, the crawler can return the URL to the URLsToVisit queue for another attempt. The crawler must give up if it is unable to fetch a URL repeatedly, because seemingly transient errors, such as a Web server not responding, can in truth be symptoms of fatal errors (e.g., a Web server is permanently disabled).

In this way, a crawler is able to fetch an arbitrarily large piece of the Web from a small list of seed URLsToVisit, using the pages it fetches to enumerate additional yet-uncrawled pages to fetch.

In implementing and operating our WebBase crawler, we have come across a number of design issues and potential pitfalls that highlight the difficulty in designing and running a large-scale crawler. We will discuss a few of these issues below.

2.1 Parallelization

Rather than the monolithic design implied by Figure 2, our WebBase crawler in fact operates as a loose federation of multiple, more focused *site crawlers*. This approach has a variety of advantages, which we detail in this section.

In WebBase's federated approach each site crawler is responsible for downloading pages from a Web site, maintaining separate URLsToVisit and VisitedURLs queues for it. Each process of our WebBase crawler typically has site crawlers for anywhere from one to two hundred Web sites simultaneously. For simplicity, we initially describe in Section 2.1.1 a simple site crawler that covers only one Web site at a time. In Section 2.1.2 we expand to multi-site-crawler processes. Figure 3, and later on Figure 5, illustrate this more detailed design.

2.1.1 *Web Site as Unit for Parallelism.* Notice in Figure 3 that parallelization introduces a *seed-URL dispenser*. This dispenser contains a seed list of URLs for all of the crawl processes. Each entry in this list contains the seed URLs for one Web site (as defined below). Usually, the entry consists of one URL for the root of that site.

In this design, each site crawler starts by retrieving the seed URLs for one Web site from the seed-URL dispenser, and uses them as the site crawler's initial URLsToVisit queue. For the typical case of one seed URL for the root of one Web site, the site crawler therefore begins its crawl at the referenced root. The crawler fills URLsToVisit with URLs it collects as it crawls within 'its' site. Each site crawler follows any given link only if that link refers to a page within the site crawler's Web site. The site crawler does not follow links to pages that are external to its site. We discuss later how such links are processed.

We consider a Web "site" to be the set of Web pages that share the same fully-qualified domain name (FQDN) at the beginning of their URLs. [3] For example, `http://www.whitehouse.gov/privacy.html` and `http://www.whitehouse.gov/kids/` share the same FQDN and are considered part of the same site. `http://www.whitehouse.com/` would be a different site. Therefore, all relative links (e.g. `congress/issues107.html`) are by definition references to pages within the same site.

Site granularity for parallelism is a natural choice for several reasons. A site usually operates in a homogeneous physical environment. Appropriate access policies for the crawler to follow, such as frequency of request submissions are therefore usually consistent for all the pages within one site. In addition, individual sites are usually governed by a single set of administrative policies, especially pertaining to the location of pages that are not to be crawled. Finally, a single human site operator is usually the contact point for an entire site should technical problems lead to the need for human intervention.

Depending on available network and CPU resources, WebBase typically runs 500 site crawlers simultaneously to fully utilize available download resources. The optimal number would vary among installations.

We favored running multiple site crawlers over running a monolithic single crawler for the following three reasons:

(1) *Size of the URL data structures.* Given the scale of the WebBase crawler, the data in URLsToVisit and VisitedURLs are large. Assuming that a monolithic crawler downloads one billion Web pages and the average length of a URL is 40 bytes, the crawler needs to manage roughly 40GB of data in URLsToVisit and VisitedURLs. The typical main memory size of existing machines is much smaller than 40GB. Therefore, a monolithic single crawler would have to store the URLs on disk and perform disk-based operations to manage them. By running multiple site crawlers independently we can avoid disk-based URL management because each crawler maintains

[3] This definition has a few drawbacks. We discuss this issue shortly.

only the list of URLs within its site. Since the number of pages in a site is typically fewer than a million, each crawler can safely manage the URL data structures in main memory.

(2) *Crawler independence.* Because site crawlers need little coordination, they can run independently of each other. This independence makes it easy to run multiple site crawlers in parallel on multiple machines. In order to increase the overall download throughput, we simply need to start more site crawlers on the same or different machines. In contrast, a single, massive crawler requires careful design and implementation of parallelization.

(3) *Site-specific rule enforcement.* The modularity of independent site crawlers simplifies the enforcement of site-specific rules.

This latter point merits elaboration, because it illustrates that seemingly simple requirements for crawling have consequences that impact the core crawler design. In this case, informal issues of courtesy make the Web site a natural and intuitive choice as the unit of crawl parallelism.

Crawler page requests comprise a significant portion of traffic on the Web. When crawlers operate on behalf of search engines, crawler visits are generally welcome at Web sites, because search engines help potential visitors find those sites. Some sites, however, wish to limit the amount or scope of non-human access because their owners must pay for the required bandwidth, or because some resources are particularly expensive for the Web server to generate. Another motive for restricting crawler traffic is that sites sometimes wish to keep some or all of their content from being mirrored elsewhere. Conventions have developed on the Web that govern the behavior of crawlers. These rules are not enforced by any authority, but proper crawlers are expected to honor them.

2.1.1.1 *Courtesy Pause.* Site administrators expect that crawlers will not request pages in rapid succession. Such crawler behavior would impact human visitors to their sites. Crawlers are therefore expected to observe a ‘courtesy pause’ between successive requests to any one site. Unfortunately, there is no fixed rule for the necessary time interval. The courtesy pause varies with the size and bandwidth capacity of each site. WebBase solves this problem by associating a request interval with each site. This information is stored in the seed URL list. WebBase defaults to a 5 second interval for large commercial sites, and 20 seconds for small, private servers. Nevertheless, some site administrators have contacted us and requested different intervals, so the capability of customizing request intervals per site is required. Crawls that are parallelized by page groupings other than site would suffer large coordination overhead to observe the per-site crawl interval pause.

2.1.1.2 *Multiple Policies per Server.* When Web site administrators wish to withhold some pages from crawlers, they may use a special tool: The Robots Exclusion Protocol [Koster 1994] specifies that every crawler should search for a file called `robots.txt` at the root of each site that the crawler intends to harvest. This file lists prefixes of URLs where crawlers are not welcome, and well-constructed crawlers must honor the applicable restrictions.

A problem arises when large Web sites are served using ‘virtual hosting.’ This mechanism allows a single physical computer to serve several distinct Web sites at once. The sets of Web pages for `example.org` and for `example.com`, for example, might both reside on a single machine with one IP address. To a human user these URLs are different; the Web server will note the requested site and handle the request accordingly.

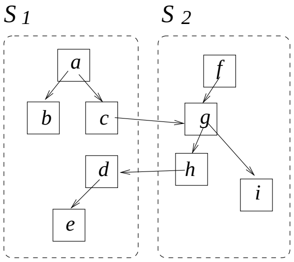

Fig. 4. Coverage issue of site crawlers

For crawlers, however, the situation is complicated by the fact that multiple aliases are also often registered for a single site. The URLs `http://example.com/` and `http://www.example.com/`, for example, might refer to a single site, not to two sites as in the example above. Crawlers must recognize when two names map to a single site in order to avoid crawling that site twice, while at the same time treating the names of multiple virtual sites as distinct, even though they too refer to one IP address.

An unsophisticated crawler would not distinguish between the above two multi-name cases and blindly conflate multiple URLs that resolve to one IP address into a single site. This crawler would subsequently retrieve “the” `robots.txt` file and honor it. This behavior overlooks that `example.org` and `example.com` might each maintain their own `robots.txt` file. WebBase addresses this problem by probing each hostname for the existence of a `robots.txt` file, and comparing them, before concluding that two hostnames refer to a single site. It would be prohibitively expensive to perform this analysis every time before retrieving a page. The site granularity for parallelism makes the analysis of aliasing and of `robots.txt` files much simpler.

2.1.1.3 *Objections to Deep Linking.* Finally, many alternatives to site granularity parallelism would register to site administrators as “deep linking.” A deep link into a site refers to a page other than the site’s root page. Some sites generate revenue by presenting advertisements on their root page, and therefore discourage deep linking. They prefer visitors to navigate down into the site, starting at the root page. Using sites as the granularity for parallelism makes it easy for a crawler to avoid entering a site via a deep link.

One drawback of the site-based crawling approach is the *locally-isolated page problem*. Since site crawlers do not follow the links to pages on other Web sites, the overall crawl may completely miss some pages. Figure 4 illustrates this problem. Two Web sites S_1 and S_2 are downloaded by site crawlers C_1 and C_2, respectively. Because the page d is reachable only through the link from site S_2, the pages d and e will not be downloaded by either crawler.

In order to download these missed pages over time, WebBase can use the snapshot of the Web that was obtained in a previous crawl. In this approach, if a page in the previous snapshot of the Web contains a link to page d, but the page d itself does not exist in the snapshot, d is added to the seed-URL dispenser for the next crawl.

2.1.2 *Alternatives for Implementing Crawl Parallelism.* One approach to implementing the conceptual design above is a *process-per-site* design. This design runs each site crawler as a process, which sleeps to ensure proper

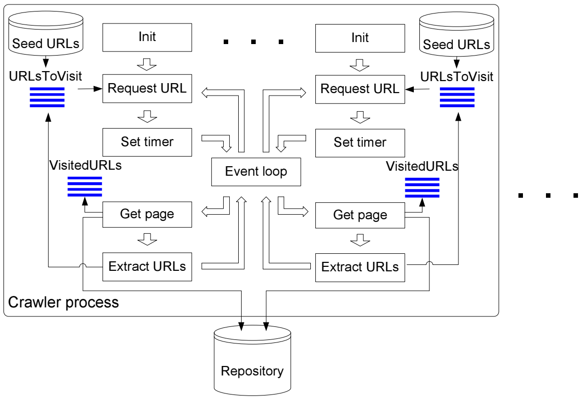

Fig. 5. Crawling multiple sites in a single process

request spacing. This design offloads parallelization management to the operating system's process scheduler. On the other extreme, the *single-crawl-process* design packs all of the site crawlers into a single process and manages request spacing as part of an event loop. In this design the loop generates successive page request events to sites only after the proper courtesy pause has been observed. This implementation approach is shown in Figure 5.

The figure shows two crawlers running within a single process. In contrast to the conceptual design of Figure 2, the single-crawl-process implementation introduces a timer before the page URL retrieval from URLsToVisit. The timer enforces the delay before each page request. The central event loop manages the loops of the asynchronous crawlers.

WebBase implements a hybrid approach of the single process and the process-per-site designs. Each crawler runs as its own process as in the per-crawler design, but as in the event loop model, multiple sites are crawled simultaneously within each process.

We will show below that this mixture of both approaches offers a good tradeoff between the approaches' design characteristics. Below, we describe the advantages of each approach separately, then we explain how WebBase blends the two designs. An exposition of the performance issues follows.

Single crawler process	Multiple crawler processes
Avoids process limits	Can crawl on multiple machines
Lower URL-seed-dispenser load	Degrades gracefully
Efficient resource utilization	Fast event loop

Table I. Some Parallelization Tradeoffs

In Table I, we enumerate the following design characteristics in favor of a **single-process solution**:

2.1.2.1 *Avoids process limits.* At the time of our initial development, the Linux environment, for example, supported only 1024 simultaneous processes. Such operating system specific limitations do not affect the single-process solution.

A single process is also easier to manage than multiple parallel ones when we need to monitor or control the crawl's rate of progress. With the crawl spread across multiple processes an individual crawler is more difficult to locate when it needs to be paused or stopped.

2.1.2.2 *Lower seed-URL-dispenser load.* With multiple processes, the load on the seed-URL-dispenser database is less controlled than in a single-process solution. Each parallel crawler process will issue an independent request for a new site's FQDN at unpredictable times. In a single process, by contrast, only one Web site is "complete," and requires replacement, at any one time. Therefore, using a single process serializes requests to the seed-URL-dispenser database. Such an access pattern induces a more efficient use of the seed-URL-dispenser database, as it helps avoid transaction conflicts (and subsequent rollbacks). Conflicts arise during transactions whenever two crawlers try to claim the next Web site to crawl at the same time. Serial access also reduces database server overhead in maintaining its (smaller) connection pool.

2.1.2.3 *Efficient resource utilization.* If we use many processes, we incur higher process-switching overhead in the operating system. Also, we incur more memory overhead: Each crawler must maintain its own file descriptors and buffers for its page repository and other I/O, as well as its own sockets to receive commands during its execution. Any singleton data structures in a crawler process must also be cloned in each crawler. The resulting larger memory footprint of multiple processes may hurt the CPU's cache hit rate as it switches from process to process.

In favor of a **multi-process solution** we find the following:

2.1.2.4 *Can crawl on multiple machines.* Only separate crawler processes can run on separate (shared-nothing) machines, making it possible to spread the crawling load across a network.

2.1.2.5 *Degrades gracefully.* The more distinct crawler processes we use for a crawl, the less data we lose should a crawler process fail. When a crawling process crashes, we lose its in-memory information for every Web site that it is crawling, including URLsToVisit and VisitedURLs.

Without this partial state, we cannot easily tell a different crawler process to "pick up" where a prior crawler process left off. So we either have to recrawl the sites that the crashed crawler was crawling (loading those Web servers excessively); read and analyze a large tail of the crawler repository to reconstruct the relevant partial state (which requires special-case roll-forward code, and a lot of disk activity every time a crawler process crashes); or give up on whatever portions of the affected Web sites have not been crawled. By using separate site crawlers, we can afford the latter crash recovery strategy: any one site crawler only loses operational data for a single site.

2.1.2.6 *Fast event loop.* An implementation fact also weighs in on the side of a multi-process solution: There is a drawback to placing many Web sites in an event loop using a (necessarily large) `select(2)` or `poll(2)` call. Each HTTP connection to one of the sites being crawled is represented by a file descriptor. When the event loop is notified that some number of the file descriptors is ready, it must examine each of the file descriptors to determine which of the (possibly many) descriptors are ready for access. This overhead increases with the number of Web sites being crawled in one process.

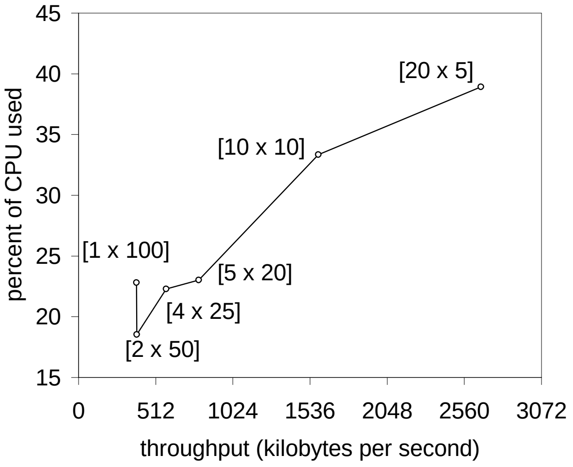

Fig. 6. Throughput of processes versus event loop (zoomed view of Figure 7)

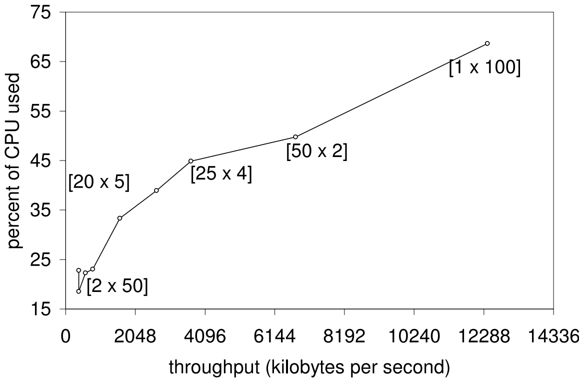

Fig. 7. Throughput of processes versus event loop

In Figures 6 and 7, we quantify the performance tradeoff between the many-process and single-process approaches to crawling numerous sites concurrently. For each point in the figures, we crawled 100 sites simultaneously, varying the number of processes for each of those crawls. Therefore, since we kept the total number of crawled sites constant, the number of sites that each process was responsible for varied.

We executed this experiment on a dual-processor 2.8GHz Xeon with 4GB RAM and gigabit Ethernet running Red Hat Linux 8.0. [4] "Total CPU utilization" in the discussion below is therefore out of 200% maximum for both processors.

In Figures 6 and 7, we plot CPU utilization against the amount of Web data that is fetched per second; Figure 6 shows a subset of the data points in Figure 7 in detail. Each point in the figure documents the result of one experimental condition. For example, the point marked "[2x50]" is the result of simultaneously crawling 100 sites by running two processes, each responsible for collecting fifty sites.

We (crudely) measured CPU utilization by observing all CPU time that the Unix command `top(1)` did not consider idle during the crawlers' run. This measure includes a slight inaccuracy in that it includes CPU cycles that serve `top`

[4] Linux 2.4.18-14bigmem and the local version of gcc "2.96"

itself, but no other significant work was being performed on the machines at the time of measurement. We chose this measure instead of simply adding up the CPU time explicitly charged to the crawler processes, because we wanted to include the CPU utilization for all supporting activity for the crawlers, such as that of the seed-URL-dispenser database.

Because of the varied disk configurations available for use on systems where the crawler might run, we avoid measuring the CPU usage (loss) of disk activity by having the crawler not write its pages to disk. Instead, the crawler performs all the motions of writing to its repository, from creating directories for each Web site to compressing the individual Web pages. Then, the crawler omits the final step of writing the compressed pages to disk. As a result, we are able to measure the CPU consumption and performance of the crawler's own operations, rather than that of its local file system.

A point in the result space therefore shows how much the two CPUs need to participate in the crawl, and how much throughput is achieved for the point's respective process-to-site ratio.

For the experimental condition "[2 x 50]" we see along the vertical axis that 18.5% CPU usage, out of 200% available on the dual-processor machine, is required. Along the horizontal axis, we see that this configuration fetches about 386 kilobytes per second. Using these result values we can determine that each percent of consumed CPU contributes, on average, to 20.8 kilobytes/second of throughput.

Qualitatively, points towards the right of the graph and close to its bottom indicate good system utilization. The configuration to the left of [2 x 50], one process crawling all 100 sites, causes a relatively high CPU load while achieving comparable throughput; therefore, it is not an efficient system configuration.

At [2 x 50] a minimal involvement by the CPUs is required. For this configuration the overhead that is "wasted" for OS context switches and event loop management is minimized. Beyond this point an increase in the number of processes does yield increased throughput, but at the cost of increased overhead.

In practice we operate WebBase at 10 processes, each crawling 50 to 100 sites in round-robin discipline. These numbers arise because the above experiments still do not model the whole operation. As described, we carefully constrained the number of variables for the experiments. The operational system functions under two additional constraints: the gathered data needs to be written to the storage subsystem, and we must share access to the Internet backbone with the rest of the University. Empirical evidence leads to the optimal [10 x 50/100] operating point.

The variance of 500 (10 processes * 50 sites/process) to 1000 (10 processes * 100 sites/process) simultaneous sites comes about as follows. When all processes operate with a minimal acceptable courtesy pause, no additional throughput is gained by adding simultaneous sites beyond 500. However, we lengthen the pause for small sites, because their servers tend to be weaker. Our seed list is consequently arranged such that small sites are collected late in the crawl. At that point we dynamically add sites to crawler work loads.

2.2 Resolving Domain Names In Seed-URL Dispenser

The URL seed list that the dispenser feeds to the crawlers is large and growing after every crawl. The WebBase seed list contains hundreds of thousands of seed URLs. As described so far, each entry in the seed list contains only the entry's respective URL, and therefore the hostname of the corresponding Web site. However, when a page is actually fetched at runtime, the respective Web server must be accessed by its IP address. Every URL in the seed list must therefore at some point be resolved to some server's address. This resolution is accomplished with the help of Domain Name System (DNS) servers, which reside on the Internet. These servers receive domain name requests and return the corresponding IP addresses.

One solution to resolving the URLs in the seed list would be to have the crawlers behave like a human Web surfer would. During human surfing activity, URLs are resolved on demand when needed. Unfortunately, for the large number of IP lookups that a WebBase-scale crawling operation requires, name resolution is a significant, performance-impacting activity that precludes such a simple approach.

A number of issues arise when trying to resolve seed domain names at runtime. Because of the way name service works, a high name server load can cause DNS requests or their responses to be dropped on the network. This happens because DNS servers usually operates over a network link that does not guarantee delivery (UDP), rather than the higher overhead transport control protocol (TCP), which guarantees that it either successfully delivers a packet, or notifies the sender.

If a DNS request or response is dropped, the operating system's resolver code will need to time out before reporting the loss. This means that a name resolution call at crawl time will block a crawler process from making progress on *any* Web site for a significant time whenever any DNS lookup fails.

In addition, most existing implementations of the name resolution function (`gethostbyname(3)`) have small memory leaks, which become a significant problem when the crawlers perform name resolution repeatedly over a long time.

Furthermore, resolving site names during the crawl makes it more difficult to start a crawl initially. All crawler processes begin by trying to establish connections to all the Web sites they plan to crawl. This synchronized, massive name lookup demand can grossly delay the system reaching a steady state. Even if DNS resolution were performed over a perfectly reliable network, the lookup introduces a significant delay that would interfere with crawling.

WebBase therefore resolves the entire seed list before the start of each crawl. This approach adds a new and long step to be carried out before a crawl can begin. For example, we took over three days to resolve 310,000 FQDNs in advance, at a DNS load that still drew attention from our observant network administrators. By resolving domain names in advance, though, we are able to control the tradeoff between lookup time and DNS server load. Also, we are able to perform basic detection of multiple Web site aliases pointing to the same Web site, and thus avoid crawling the same Web site repeatedly under different hostnames (e.g. `stanford.edu` and `www.stanford.edu`).

Of course, converting names into addresses in advance also carries the risk that if the IP addresses are not used promptly, they could be changed between the time they are found and the time they are used. We therefore repeat the resolution process from time to time.

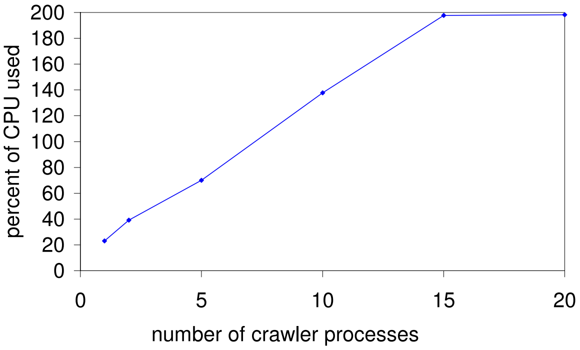

Fig. 8. Crawler CPU usage. Measurements taken when each process crawls 100 sites simultaneously. Point [5, 70] therefore indicates that crawling 500 sites concurrently via 5 processes requires 70% of one CPU.

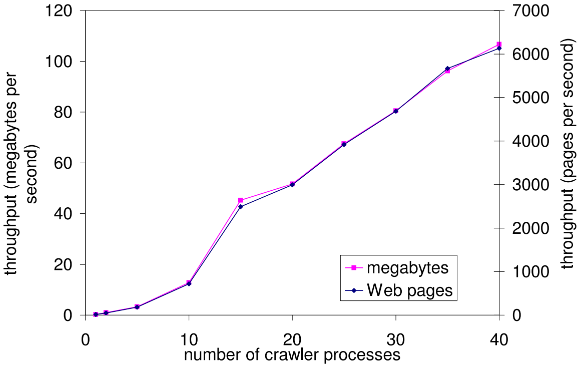

Fig. 9. Crawler throughput for many processes

2.3 Crawler Performance

For the following performance experiments we crawled HTML text pages only. The page sizes averaged about 18,000–19,000 bytes each. The crawler can, of course, collect other resources such as images as well.

We found that our crawler scales close to linearly up to the limits of our CPU, as shown in Figure 8. In the figure, we plot the CPU usage of our crawler processes, each crawling up to 100 sites concurrently. In Figure 9, we plot the corresponding throughput of the crawler processes, in bytes (left vertical axis) and Web pages (right vertical axis). Figure 8 shows that we add processes until their CPU usage approaches the 200% limit of the dual-processor machine.

For example, when we were running five concurrent crawler processes, they consumed 70% of the first CPU while crawling 500 sites concurrently, and gathering over 3.25 megabytes per second of HTTP data.

Starting at the lower-left of Figure 8, with one crawler process, we are able to keep adding crawler processes until we reach 15 processes before we consume all of the CPU available in the system. We are then able to continue adding crawler processes up to 20 at the upper right, collectively crawling 2000 Web sites concurrently. Even though we introduce CPU contention for the individual crawler processes when we reach concurrency of 15 processes, we continue to gain crawl throughput (Figure 9); as we can see, the crawl speed scales up well, allowing us to reach 50 megabytes per second, or 3000 Web pages per second, in this configuration.

Once we reach 20 simultaneous crawler processes, we are still able to add more crawlers to the system, but in this

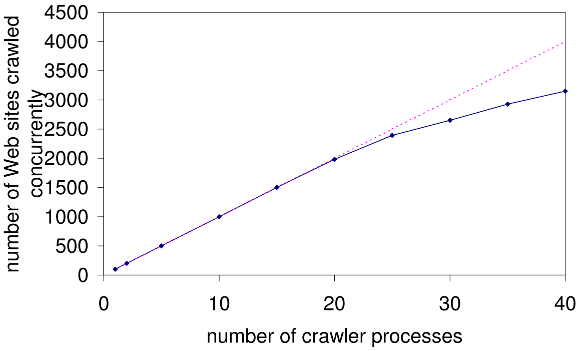

Fig. 10. Crawled sites for many processes

scenario, there is no longer enough CPU time for the database to ensure that the crawlers maintain a collective 2000+ independent Web sites to crawl. The database simultaneously loses more CPU time to other crawler processes and is expected to service requests for more of them. We see the effect in Figure 10, which shows the number of Web sites the crawler is able to crawl concurrently given the number of crawler processes running. For example, running 30 crawler processes with our configuration (100 Web sites per process) should yield a crawl of 3000 Web sites concurrently (as shown in the dotted line), but the crawler is actually only crawling 2651 Web sites at the same time (as indicated by the solid line).

We see the crawler's throughput behavior in this scenario in Figure 9. Though the number of Web sites crawled per process begins to fall after we start consuming all the CPU, the crawler's throughput continues to grow about linearly. For example, with 40 crawler processes, we are able to fetch over 111 million bytes per second of HTTP data (not counting TCP/IP overhead) from 3151 Web sites simultaneously. For comparison, gigabit Ethernet has a theoretical maximum throughput of 125 million bytes per second.

2.4 Crawler Design Summary

While the conceptual operation of Web crawling is simple, the following experiences from designing and implementing an operational crawling facility are noteworthy:

2.4.0.7 *Parallelization—Content Granularity:*. "Sites," as defined by the set of pages that share one fully qualified domain name, are a convenient unit for crawling parallelism:

—A site usually operates in a homogeneous physical environment. Appropriate courtesy pauses, or the total number of pages to collect from a site, are therefore usually consistent for all the pages within one site.

—Individual sites are usually governed by a single set of administrative policies, especially pertaining to the location of pages that are not to be crawled.

—A single human site operator is usually the contact point for an entire site should technical problems lead to the need for human intervention.

However, the practice of virtual hosting, which has a single physical environment supporting multiple sites, adds complexity to site-level granularity. Multiple virtual sites are often also governed by different administrative policies, all of which must be accommodated.

2.4.0.8 *Parallelization—Implementation:*. Parallel crawlers can be implemented by starting multiple processes for different sites or by crawling multiple sites within a single process. A hybrid approach of these two is optimal. Throughput gains level out at about 500 simultaneously crawled sites because of disk contention, unless additional machines are deployed.

2.4.0.9 *Issues of Courtesy:*. Crawlers must be able to accommodate a wide range of site policies. Some of these policies are captured in a file called `robots.txt` by convention. The need to observe courtesy requirements has a strong impact on the design of a crawling facility.

3. INDEXING

The *Indexer* and *Analysis* Modules (see Section 1) are reponsible for building and maintaining the indexes that support queries over WebBase. The *Indexer* module builds three basic indexes: an *offset index*, a *text index*, and a *link/graph index*. The offset index records the physical location (or "offset") of each Web page in the containers (very large files) in which WebBase stores compressed pages. The pages are concatenated within the containers, with special separators in between. This index enables random access to an arbitrary page in the repository (e.g., retrieve the page corresponding to the URL `http://www-db.stanford.edu/db_pages/projects.html`). The text index enables content-based retrieval, using an inverted index to support keyword searches over the repository (e.g., retrieve pages containing the keyword "WebBase"). Finally, the link index supports link-based retrieval (e.g., retrieve the set of pages that page X points to), by compactly representing the subgraph of the Web induced by the pages in the repository.

Using these basic indexes and the pages, the *Analysis* module builds a variety of other derived indexes. For instance, using the link index and the iterative algorithm described in [Brin and Page 1998], the analysis module computes and stores the PageRank of each page in the repository (the PageRank index). Similarly, by combining link information with the textual content of the pages as described in [Haveliwala et al. 2002; Haveliwala et al. 2000], the analysis module builds a similarity index that maps each page to a set of "similar" pages.

3.1 Design Approach

To deal effectively with the massive scale of the Web, we have designed new build schemes and representation structures for many of the indexes used in WebBase. Our designs are characterized by the following:

—*Index Build Parallelization and Distribution.* The scale of the Web often precludes the direct adaptation of simple sequential index-building schemes that perform well for smaller sized data collections (e.g., a collection of a few million documents or a graph of a few thousand nodes). Schemes that actively parallelize and distribute the computation can result in large savings in cost and index-build time (e.g., the parallelization scheme described in Section 3.2).

—*Compression and Caching of Index Structures.* Many indexes over Web collections are too massive to fit in main memory. Therefore, index compression (to squeeze larger portions of the index into main memory) and caching are crucial for reducing index access times. Compression techniques that exploit unique characteristics of Web data are particularly effective (see, for example, [Raghavan and Garcia-Molina 2003], and other methods referenced in Section 5).

—*Index-specific Page Identifiers.* While a global page identifier is necessary to allow interoperability of indexes [5], it is often necessary for each index structure to employ its own index-specific identifiers to minimize access times and index size. While index-specific identifiers require an additional translation map to convert to and from global page identifiers, it has been our experience that the savings in using custom identifiers are worth the extra translation step.

In the following sections, we take up the text index for further discussion. We highlight the techniques employed in WebBase to build efficient and Web-scalable index structures.

3.2 Text index

To support text-based retrieval, we build an inverted index over the collection of Web pages in the repository. The size of Web repositories and the need to periodically crawl and rebuild indexes to maintain “freshness” require that we employ a highly scalable and efficient index build scheme. In this paper, we do not discuss the design of our text-indexing system in detail (we refer the reader to [Melnik et al. 2001]) but merely highlight the key features and provide some illustrative performance results.

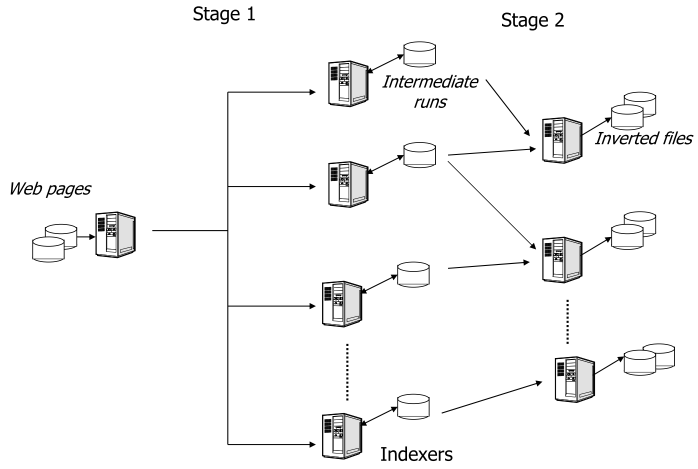

Fig. 11. Text indexing stages

The WebBase inverted index build system employs a distributed shared-nothing architecture as shown in Figure 11. The inverted index is built in two stages. In the first stage, each *indexer* receives a mutually disjoint subset of pages. 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.

[5] In WebBase, we apply a uniform hash function to the URL associated with each page, to generate a 64-bit global page identifier. Before applying the hash function, the URL is first “normalized” to ensure that equivalent representations of the same URL generate the same identifier.

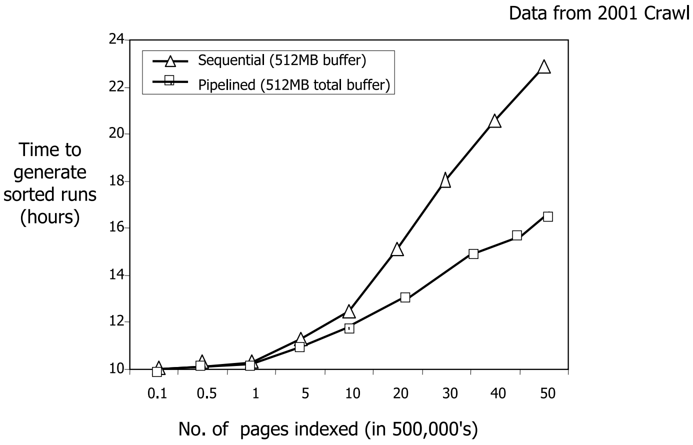

Fig. 12. Performance gain through pipelining

The two key features of our build system are a highly parallel indexer and the use of an embedded database system to store and manage inverted files.

3.2.0.10 *Highly parallel indexer.*. We split the process of generating sorted runs into three phases - *loading* (some number of pages are read from the input stream and stored in memory), *processing* (pages are parsed and tokenized to generate postings, which are sorted in-place), and *flushing* (sorted postings are flushed to disk). Our indexer is designed as a multi-threaded software pipeline that uses several buffers in parallel, with each buffer in one of these three phases. We analytically determine an optimal pipeline schedule [Melnik et al. 2001] and configure the indexer to operate according to this schedule. Figure 12 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 in our execution environment, a sequential indexer is about 30–40% slower than a pipelined indexer.

3.2.0.11 *Embedded-database-backed inverted index storage.*. We use an embedded database system[6] called Berkeley DB [Olson et al. 1999] to store and manage inverted files. The embedded database provides device-sensitive file allocation, database access methods that include B-trees, and optimized caching, with the advantage of a much smaller footprint compared to full-fledged client-server database systems. However, the challenge is to design an inverted index storage format that fully utilizes the features of the database system. The standard organization of a B-tree based inverted file involves storing the index terms in the B-tree along with pointers to inverted lists that are stored separately. However, such an organization does not fully utilize the caching capabilities of the database system.

We have therefore designed a *mixed-list storage scheme* to efficiently map inverted lists to the *(key,value)* organization of a Berkeley DB B-tree. In this scheme, each key of the database contains a posting, i.e., an (index term, location) pair. Each "value" consists of a number of successive postings in sorted order, potentially including postings

[6] An embedded database is a software toolkit that provides database support for applications through a well-defined programming API.

that refer to different index terms. The postings in the value field are compressed, and in every value field the number of postings is chosen so that the length of the field is approximately a constant. In [Melnik et al. 2001], we empirically demonstrate that such an organization provides the ideal balance between too many internal pages (very tall B-trees) and too many overflow pages in the database, resulting in very compact inverted indexes.

Number of pages (million)	Input HTML size (GB)	Index size (GB)	Index size (%age)
0.1	0.81	0.04	6.17
0.5	4.03	0.24	6.19
2.0	16.11	0.99	6.54
5.0	40.28	2.43	6.33
39.0	314.18	20.10	6.40
119.0	964.00	56.39	5.85

Table II. Mixed-list scheme index sizes

Table II shows the size of the inverted index for different collection sizes, assuming only one posting is generated for all occurrences of a term in a page. The table shows that the mixed-list storage scheme scales very well to large collections. The size of the index is consistently below 7% the size of the input HTML text.

Query set	No. of postings	Execution time (in secs)	
	(fraction of repository size)	No URL lookup	Including URL lookup
1	1%	94.8	108.5
2	0.1%	26.3	55.5
3	0.01%	15.8	53.9
4	0.001%	12.9	53.5

Table III. End-to-end query execution times (for 100 single-word queries)

Finally, Table III summarizes the results from experiments to measure end-to-end query execution times when using the index. The experiments were conducted using an inverted index over 119 million pages – the data set corresponding to the last row of Table II. Each row of Table III corresponds to a set of 100 single-word queries. The words for each set were chosen based on the length of their postings list. For instance, each word in query set 1 had a postings list whose length was 1% (as indicated in column 2) of the number of pages in the repository. The inverted index was stored as a set of 5 inverted files on separate disks. All the queries for a set were executed successively, but the index cache was flushed between each set. For each query, the top 20 results, using a ranking function that makes use of cosine and PageRank, were requested. Note that since the ranking function requires access to PageRank and document length, the execution times include the time to access these two indexes. Column 3 lists the time to execute each query set (i.e., the total time for all 100 queries belonging to a set) when only the page identifiers were requested. Column 4 includes the extra step of looking up the URL for each of the top 20 page identifiers. The results indicate that even for queries whose postings contain over a million entries (query set 1), the end-to-end query execution time is just over a second per query.

4. WHOLESALE REPOSITORY INTERFACE

Once the effort of crawling has been invested, WebBase enables researchers within and outside of Stanford University to access the collected data 'wholesale.' The requesters thus avoid the complex, slow, and expensive crawling process and can focus on the research for which they require the pages.

To use the Multicast Module researchers obtain an open source client, which they operate on their machine. Through the client a researcher can contact WebBase's Multicast Module across the Internet and request any number of Web pages that the crawler has previously collected (top right of Figure 1). Those pages are returned to the researcher at Internet speeds as a stream that can be stored or analyzed as it arrives. Requesters may ask for all the pages from a particular set of sites, or for any number of pages to be delivered in the order in which they were crawled. A researcher might, for example, index each arriving page and store the resulting index entries, or she might perform automated linguistic analysis on arriving text.

The client to this interface is designed to allow its users to specify Web page processing functions that are called whenever a new page is received from WebBase. The researcher may in fact specify multiple such page analysis functions, all of which are invoked for each arriving page. Beyond this degree of parallelism at the researcher's site, the researcher may request multiple simultaneous Web page streams, each directed to a different machine.

Two traditional designs for an interface to the wholesale service come to mind: a database style interface and a file system interface. Neither is appropriate for the purpose of the Multicast Module. The former would enable, for example, SQL queries, but the data model for the WebBase archive is not relational; nor is it otherwise clearly structured. Creating the illusion of such structure would be expensive and unnecessary for the interface's purpose.

A file system interface would be a closer fit for the wholesale interface. One might think of Web sites in the repository as directories and individual pages within those sites as files. However, this analogy would be misleading. The Multicast Module does not provide random access to individual pages, as a file system would, but to streams. Also, the notion of subdirectories that would be implied by a file system analogy does not apply to the interface and would thus be misleading.

The notion that is closest to our final design for the wholesale interface is the Unix `cat` command, modified to produce streams of Web pages from one or more Web sites that are delivered across a network. This interface is simple and has been successful.

4.1 Distribution Machinery

Figure 13 shows an overview of the wholesale interface to WebBase. The Figure is divided into three portions: on the left is the repository of previously crawled pages. The block on the repository's right (labeled *WebBase Server*) is part of the local WebBase installation. The components within the block implement the combination of the Multicast Module in Figure 1, and its associated API. Further yet towards the right a vertical line represents the network. The portions to the right of this line are located at the remote requesters' site. Each of the two vertically stacked enclosing blocks on the right, finally, represent one remote requester, *User X* and *User Y*.

We discuss details of these building blocks in the following sections. Note that the only code a requesting client needs to write are the modules called *Handler* on the far right of the Figure. All other code either runs as part of the main WebBase installation or is obtained by the requesting user from the Web and run as-is on his machine.

4.1.1 *Feeders and Distributors.* *Feeder*s in Figure 13 are processes that convert repository contents into streams of compressed pages. Modules to the right of a Feeder are thereby insulated from the details of repository storage organization. At the heart of the distribution service implementation are a single *Distributor Daemon* process and one *Distributor* process for each feeding requester. Each user contacts the Distributor daemon at a well-known address and port, requesting a stream of pages. Each request can include the name(s) of the Web site(s) to stream out, or have the Distributor stream out each of its Web sites in turn. A maximum number of pages to transmit can also be specified.

The Distributor Daemon launches a Distributor process for the requesting client and returns that Distributor's internet address and port to the requesting client. The new Distributor exclusively serves that client. The Distributor in turn instantiates a Feeder and is then ready to stream pages across the network to its user's machine. The stream follows a pull discipline; clients thereby regulate the stream volume to their needs.

4.1.2 *Parallelization During Delivery.* Inside User *Y*'s block in Figure 13 is a *Client RunHandlers* module. This module is a piece of C code that *Y* has acquired from the WebBase Web site. The module runs as a process within *Y*'s machine and receives the compressed stream from the associated Distributor. Attached to RunHandlers 1 are three *Handler* modules. These modules are written by *Y* and linked to the RunHanders module. Each Handler receives a decompressed copy of every page that travels across to *Y*'s machine. Each Handler may perform a different kind of analysis on the incoming stream. Handlers are deliberately structured very simply. They need only provide a small number of methods that RunHandlers can call. Figure 14 shows the Handlers class that *Y* needs to specialize.

An alternative to this design of multiple Handlers analyzing the stream in real time would be to undertake the (in this case) three analyses after the stream has been received, rather than while the stream is arriving. After all, the processing speed of the slowest Handler determines the speed of the stream, since the Client RunHandlers module pulls pages across the network, thereby exerting back pressure on the channel.

The great advantage of our parallel, runtime stream analysis is that streams do not need to be stored at the receiving requester's site. Consider, for example, one Handler extracting each page's title, the second noting which pages contain the name of a particular company, and the third analyzing each page to determine its topic areas. This extracted metadata occupies little space compared to the entire stream. Client *Y* can therefore make due with much lower storage capacity than if the entire received stream needed to be stored. Of course, storing the stream is an option. A single Handler might simply direct each incoming page to disk.

Notice that User *X* hosts two Client RunHandlers processes, each feeding from the same Distributor. This arrangement indicates that User *X* has two CPUs, each of which is capable of receiving streams and passing pages to Handlers. User *X* may have these CPUs (and the RunHandlers processes) on the same machine or on different machines with distinct IP addresses. In either case, the Distributor process splits the outgoing stream into two disjoint smaller streams

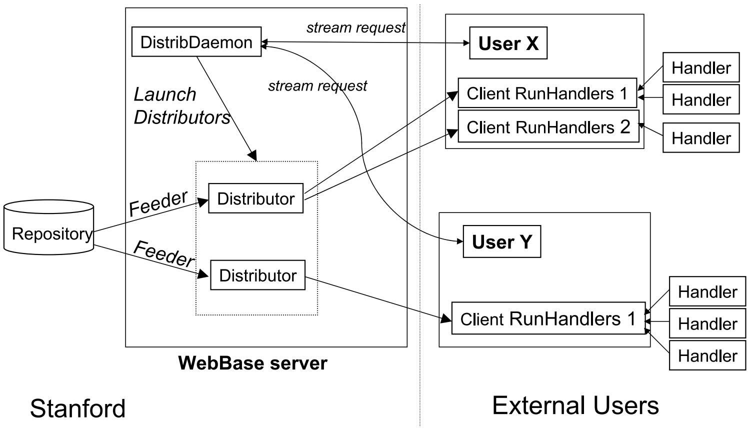

Fig. 13. Illustration of remote streaming of the Web repository to two researchers. User *X* is processing his Web stream using two CPUs. User *Y* is using a single machine to process her Web stream.

```
class Handler {
 public:
  virtual ~Handler() {};
  virtual void Process(const string& page, string url, string time,
                       int docid, unsigned long reppos) = 0;
  virtual void Init() = 0;
  virtual void Finish() = 0;

  bool get_recoverable_error()   { return  recoverable_error; }
  bool get_unrecoverable_error() { return  unrecoverable_error; }

 protected:
  bool recoverable_error   ; // you can restart me if you want
  bool unrecoverable_error ; // go away, I'm really dead
};
```

Fig. 14. Users implement a Handler to receive Web resources and process them as desired. Handler::Process() is called with one Web resource at a time.

by a simple round-robin scheme. More than the two RunHandlers of Figure 13 are supported, providing for easily implemented large-scale client-side parallelism.

The following section presents an example for how this architecture can be used for an application.

4.2 Implementation Example: Anchor Text Index

To illustrate the distribution facility consider the problem of *User X* building an index over the anchor texts of all documents in the repository. Anchor text are the words in Web documents that can be clicked on to travel along a link to a new document. The new index will reside at *X*'s site and allow him to quickly find all the words that occur in anchor texts of links that point to a given document (URL). Such a facility can be used, for example, to find words that describe the Web page at a particular URL.

The anchor text indexer executes through two phases of operation. For the first phase *X* installs the latest WebBase client, which provides the code within the box marked *User X* in Figure 13. Assume that, like *User X* in the Figure, *X*'s

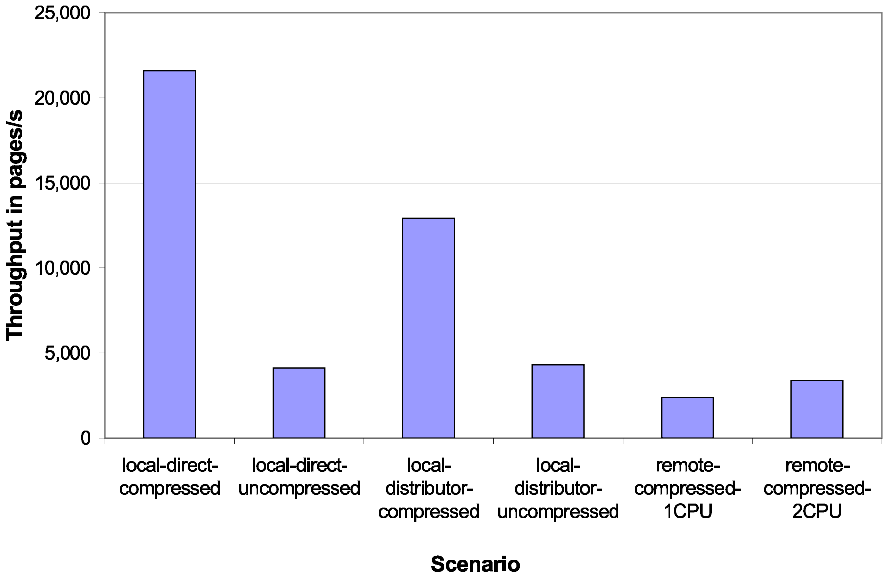

Fig. 15. Graph showing the throughput in $pages/s$ for streaming Web pages, under various scenarios.

site has two CPUs available for the anchor text extraction task. Next, *X* writes a Handler `AnchorTextExtractor`, which receives one repository page at a time and extracts a pair $< targetURL, anchortext >$ for each hypertext link it finds on the page. A Handler simply implements three key methods shown in Figure 14 to perform the desired task. As part of the build process for the WebBase client, `AnchorTextExtractor` is compiled and linked into the client.

To begin the process of streaming all of WebBase's contents to *X*'s site the WebBase client software contacts the Distributor Daemon to request a Distributor. The Daemon creates a Distributor process and passes its Internet address and port to the requesting WebBase client. User *X* then starts two Client RunHandlers processes, each of which has a separate copy of the AnchorTextExtractor, and they connect to the Distributor process. They cause the Distributor to partition the page stream into two disjoint streams, one for each RunHandlers process.

Each Client RunHandlers copy passes the Web pages it receives to its AnchorTextExtractor Handler, which extracts the anchor text words and accumulates them in main memory, purging accumulated sets to disk from time to time.

At the end of Phase 1, when both `RunHandlers` are finished processing their portion of the repository, Phase 2 merges all the resulting sets of pairs into one index, which implements the desired mapping $\{$document $d \rightarrow$ terms in anchor texts for links to $d\}$.

4.3 Performance

We now describe the performance of our repository wholesale interface and identify bottlenecks. The core performance question is how much overhead the architecture of Figure 13 introduces beyond disk access costs.

Our experimental setup included a repository of 350GB compressed HTML text, which corresponds to (the hypertext portions of) 125 million Web pages. [7] The repository was stored on a dual AMD Athlon 1533GHz machine with

[7] Our measured compression ratio for this material is 0.27. The corresponding uncompressed data would thus require 1.3TB of space.

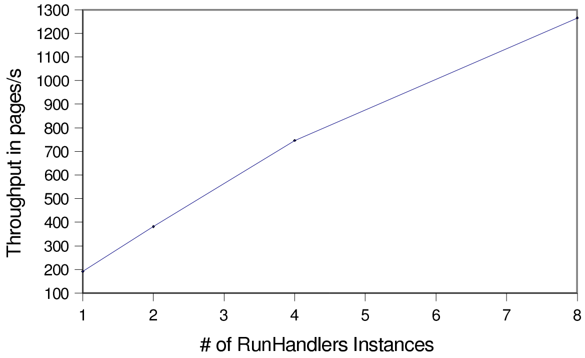

Fig. 16. Graph showing the throughput in $pages/s$ for an analyzer that tabulates document frequencies of terms, as the number of analyzer instances, each utilizing a CPU, varies. Note the near linear scale-up in throughput in the number of RunHandlers processes.

2.5GB of main memory, and a five-disk software RAID-5 volume. The RAID volume was measured to have a sequential read throughput of 77MB/s. Unless otherwise stated, all performance results were measured by processing the first 10 million pages of the repository, which consists of 26GB of compressed (97GB uncompressed) HTML content.

Our first set of experiments measured how fast content could be extracted and delivered to a Handler when neither the network nor the Distributor machinery was involved. We measured this setup under two conditions: for the first we simply transferred the compressed data, for the second we added decompression.

For these experiments we processed the test content locally on the machine that houses the page repository. In this scenario all of Figure 13 is confined to a single machine, and one RunHandlers module is attached directly to a Feeder, bypassing the Distributor elements. A single `NoOpHandler` is attached to the RunHandlers module. This Handler immediately returns when it is passed a page for processing.

The leftmost two bars of Figure 15 show results from this set of experiments. The rate at which pages were processed from the repository to the `NoOpHandler` was 22Kpages/s. Elapsed time for the 10 million pages (not shown) was 7m 43s, which translates to about 57MB/s, or 75% of the raw I/O throughput. Throughout the experiment CPU usage hovered at about 90%.

The second bar in Figure 15 shows performance when decompression is added to the experimental condition. Throughput then drops to 3.8Kpages/s, or 38MB/s of uncompressed data. In this setup the RunHandlers process became CPU bound; disk access presented no bottleneck. Given that in practice any useful analysis will require decompression we conclude that these numbers are currently the upper bound for local WebBase repository processing.

In the second set of experiments, we added the Distributor interface, used locally, to measure the overhead from the Distributor mechanism itself. Network delays are thus still excluded in this setup. For our initial measurements we also left the pages compressed. Figure 15's third and fourth bars show the results. Without decompression, throughput is 13 Kpages/s, a 41% drop from the 22 Kpages/s measured without the Distributor.

A surprise results when decompression is added to this setup. Throughput *improves* under this condition to 4.3Kpages/s, as compared to 3.8Kpages/s when the RunHandlers/Handler components retrieve directly from the repository, rather than through a Distributor.

This apparent contradiction is explained as follows. The server is a dual-CPU machine. However, when the Run-

Handlers module retrieves through a local Feeder, all operations take place in a single process on a single CPU. However, the pairing of Distributor and RunHandlers introduces an opportunity for parallelism. Repository access and decompression now each occur on a different processor. We validated this conclusion by repeating the experiment while having one of the two CPUs utilized at 100% by a dummy process. Throughput then dropped to a 3.5Kpages/s level.

Note that the advantage of parallelism persists even when a network is introduced between the Distributor and the RunHandlers module. Decompression will in that case take place at the remote client.

In the third set of experiments, we investigated remote distribution. A single Distributor operated on the repository server. The server was connected to a client machine[8] through a 100Mb/s network. Initially, one RunHandlers process with the `NoOpHandler` on the client machine connected to the Distributor to retrieve and immediately discard the compressed test content.

Under this condition we measured a throughput of 2.4 Kpages/s; this translates to 6.6 MB/s, which is close to 53% of the maximum network bandwidth. In this scenario, the *s*erver CPU was the bottleneck.

We then had a second RunHandlers process on the client machine connect to the existing single Distributor on the server machine. As discussed earlier this deployment of two recipient processes will cause the Distributor to deliver pages alternately to the two RunHandlers in round robin fashion. The underlying implementation multithreads the distributor. The addition of the second RunHandlers module thus introduced an opportunity for parallelism on the server. The resulting addition of the second server CPU is reflected in an increase of throughput to 3.4 Kpages/s, which translates to 8.9 MB/s (72 Mb/s). The limiting factor in this case was still the CPUs on the server, although 72% of the network bandwidth was now utilized. Thus, the current repository server is capable of delivering compressed streams from disk to remote client at a maximum of 3.4 Kpages/s (across a 100Mb/s network). These results are also summarized in Figure 15.

In a less constrained setup, and to demonstrate the ability of our architecture to support distributed computation for indexing and analysis, we now consider a scenario in which decompression is enabled, and a non-trivial page handler is activated. We measured the page throughput when we operated a simple word statistics handler, similar to the anchor text indexer of Section 4.2.

We varied the number of simultaneous RunHandlers instances, which retrieved a 100K-page subset from the repository. For this set of experiments, we made use of two quad Pentium III 700 MHz machines on the client side. Figure 16 shows the throughput in pages/s as an increasing number of RunHandlers processes are invoked at the client. A single RunHandlers process on one of the eight remote client CPUs is only able to process 200 pages/s. Extrapolating, it would take roughly 90 hours to process the entire repository of 125 million pages. The client CPU decompressing is the bottleneck.

The maximum number of RunHandlers processes we instantiated on the client machine was eight to ensure that each process had full use of one CPU. Note the near-linear scaling of throughput in the number of instantiated RunHandlers processes.

[8] This client machine has the same hardware specifications as the server.

Disk access is not a bottleneck in any of the realistic scenarios we considered. The five-way RAID 5 disk configuration is ideally suited to the application of streaming Web pages —sequential I/O over a large data set.

4.4 Wholesale Interface Design Summary

Facilities: The delivery interface provides clients with facilities for retrieving pages from WebBase across the network. The interface is simple and modeled on the Unix `cat` command.

Performance: The design allows maximum use of parallel processing on both the server and client sides. The addition of CPUs on the client side yields linear gains in throughput.

Parallelism on the server side manifests in the stream-partitioning facility, which enables the threaded Distributors to utilize multiple CPUs, one for each stream partition. On the client side multiple RunHandlers processes and their associated Handlers may each operate on a different CPU. This capability allows clients to process partitioned streams on multiple CPUs. The Handlers architecture enables clients to process each stream or stream partition towards multiple purposes at once.

Performance in our experiments was never limited by either disk I/O or the 100Mb/s network. For all experimental conditions that included decompression the CPU was the limiting factor.

The current wholesale interface can deliver 22Kpages/s compressed locally from disk to distributor. When deliving to a remote machine via a 100Mb/s network throughput is 3.4Kpages/s compressed when two processors are utilized at the client. The architecture scales linearly with the number of RunHandler modules at the client when each module can use one CPU.

5. RELATED WORK

In earlier publications we discussed alternatives for the Page Repository design ([Hirai et al. 2000]), and we examined optimal strategies for keeping the repository fresh, given the constantly changing Web and limited crawling resources ([Cho and Garcia-Molina 2000c; 2003]). In [Cho et al. 2000] we explored how we can automatically and efficiently identify site mirrors so as to conserve crawling resources. We developed strategies for Web servers to increase their exposure to crawlers, while minimizing the load that these crawlers impose on them ([Brandman et al. 2000]). The possibilities for incremental Web crawls in place of the currently preferred collection of periodic Web snapshots was discussed in [Cho and Garcia-Molina 2000a]. In [Melnik et al. 2001] we showed how we use distributed computing resources to build the large text indexes for the pages that our crawlers return. In [Raghavan and Garcia-Molina 2003] we explained how large link graph indexes can be stored efficiently. In [Arasu et al. 2001], finally, we presented a broad sweep of algorithms that pertain to Web repositories.

In the following we list a number of research results by others. The material is organized to mirror this article.

5.1 Crawling

Web crawlers have been studied since the advent of the Web [McBryan 1994; Pinkerton 1994; Burner 1998; Brin and Page 1998; Heydon and Najork 1999; Cho and Garcia-Molina 2000b; Miller and Bharat 1998; Eichmann 1994; Cho et al. 1998; Chakrabarti et al. 1999; Diligenti et al. 2000; Coffman, Jr. et al. 1997; Cho and Garcia-Molina 2000d].

These studies can be roughly categorized into one of the following topics:

—*General architecture* [Brin and Page 1998; Heydon and Najork 1999; Cho and Garcia-Molina 2000b; Miller and Bharat 1998; Eichmann 1994]: The work in this category describes the general architecture of a Web crawler and studies how a crawler works. For example, reference [Heydon and Najork 1999] describes the architecture of the then-DEC SRC crawler and its major design goals. Some of these studies briefly describe how the crawling task is parallelized. For instance, reference [Brin and Page 1998] describes a crawler that distributes individual URLs to multiple machines, which download Web pages in parallel. The downloaded pages are then sent to a central machine, on which links are extracted and sent back to the crawling machines.

—*Page selection* [Cho et al. 1998; Chakrabarti et al. 1999; Diligenti et al. 2000]: Since many crawlers can download only a small subset of the Web, crawlers need to carefully decide which pages to download. By retrieving "important" or "relevant" pages early, a crawler may improve the "quality" of the downloaded pages. The studies in this category explore how a crawler can discover and identify "important" pages early, and propose various algorithms to achieve this goal.

—*Page update* [Coffman, Jr. et al. 1997; Cho and Garcia-Molina 2000d]: Web crawlers need to update the downloaded pages periodically, in order to maintain the pages up to date. The studies in this category discuss various *page revisit policies* to maximize the "freshness" of the downloaded pages. For example, reference [Cho and Garcia-Molina 2000d] studies how a crawler should adjust *revisit frequencies* for pages when the pages change at different rates. We believe these studies are orthogonal to what we discuss in this paper.

Another body of literature designs and implements *distributed operating systems*, where a process can use distributed resources transparently (e.g., distributed memory, distributed file systems) [Tanenbaum and Renesse 1985; Anderson et al. 1995]. Clearly, such OS-level support makes it easy to build a general distributed application, but we believe that we cannot simply run a centralized crawler on a distributed OS to achieve parallelism. A Web crawler contacts millions of Web sites in a short period of time and consumes extremely large network, storage and memory resources. Since these loads push the limit of existing hardware, the task should be carefully partitioned among processes and they should be carefully coordinated. Therefore, a general-purpose distributed operating system that is not tuned to the semantics of Web crawling will likely lead to unacceptably poor performance.

5.2 Web repository distribution

There is little prior work in providing external researchers with a streaming interface to large scale Web-page repositories. The Internet Archive [The Internet Archive] provides researchers with access to historical crawl data. In their model, researchers are provided with accounts on the Internet Archive machine cluster, where they are given the ability to execute their programs on the archive's dataset. Our distribution model differs in that we provide a mechanism for researchers to stream the Web repository to their own machines for analysis and indexing. The Google search engine [Google Inc] provides a programmatic API for querying their indexes; however, the Google API does not allow for the retrieval of large subsets of the Web. The goal of their API is to eliminate the need for "screen-scraping" by researchers wishing to access their index programmatically, rather than the large scale dissemenation of Web-page repositories.

5.3 Text index

Motivated by the Web, there has been recent interest in designing scalable techniques to speed up inverted index construction using distributed architectures. In [Ribeiro-Neto et al. 1999], Ribeiro-Neto, et al. describe three techniques to efficiently build an inverted index using a distributed architecture. However, they focus on building global (partitioning index by term), rather than local (partitioning by collection), inverted files. Furthermore, they do not address issues such as global statistics collection and optimization of the indexing process on each individual node.

Our technique for structuring the index engine as a pipeline has much in common with pipelined query execution strategies employed in relational database systems [Garcia-Molina et al. 2000]. Chakrabarti, et al. [Chakrabarti and Muthukrishnan 1996] present a variety of algorithms for resource scheduling with applications to scheduling pipeline stages.

There has been prior work on using relational or object-oriented data stores to manage and process inverted files [Brown et al. 1994; Gorssman and Driscoll 1992]. Brown, et al. [Brown et al. 1994] describe the architecture and performance of an IR system that uses a persistent object store to manage inverted files. Their results show that using an off-the-shelf data management facility improves the performance of an information retrieval system, primarily due to intelligent caching and device-sensitive file allocation. We experienced similar performance improvements for the same reasons even though our storage format differs greatly from theirs because (we utilize a B-tree storage system and not an object store).

Reference [Viles and French 1995] discusses the questions of when and how to maintain global statistics in a distributed text index, but their techniques only deal with challenges that arise from incremental updates. We wished to explore strategies for gathering statistics during index construction.

A great deal of work has been done on several other issues, relevant to inverted-index based information retrieval, that have not been discussed in this paper. Such issues include index compression [Moffat and Zobel 1996; NgocVo and Moffat 1998; Witten et al. 1999], incremental updates [Jeong and Omiecinski 1995; Tomasic et al. 1994; Zobel et al. 1992; Witten et al. 1999], and distributed query performance [Tomasic and Garcia-Molina 1993].

6. CONCLUSION

We explored three WebBase components in some detail. In particular, we introduced design tradeoffs for Web crawlers, text indexing, and large-scale stream distribution of Web content. One of the major choices to make in crawler design is whether, and how to parallelize the crawl activity. For example, one problem with parallelization is the need to avoid multiple crawlers visiting the same Web sites. Such mishaps would be a significant waste of bandwidth resources. Small Web sites would be particularly hard hit. Our solution is a hybrid of centralized and federated crawler control.

The necessity to consider numerous details in the Web's massively heterogeneous services makes crawling much more difficult than its basic concepts would suggest. For example, we pointed to the difficulties posed by virtual hosting, which requires special attention when examining a site's crawl restrictions (robots.txt).

For the task of indexing large amounts of text, the WebBase indexing engine uses a pipelined approach. Operations are grouped into two stages. During the first stage, indexing work is distributed to indexing subsystems, which produce sorted index entry subsets on disk. In stage two, these 'sorted runs' are merged. We showed that pipelining provides significant performance benefits, particularly once the number of pages to be indexed grows beyond five million pages.

Of WebBase's wholesale, high-speed, wide-area Web page distribution module we described the architecture that allows clients to start, control, and utilize page streams effectively. In particular, we showed that a single stream may be fed to multiple analysis modules that observe or derive properties of pages. We presented performance figures for several streaming scenarios. When stream clients are located on the same machine as the page repository (i.e., no network delays), our stream rate is about 75% of the raw sequential-disk-read rate. When decompression is taken into account, local WebBase streams provide about 4.3Kpages/sec, with the CPU being the bottleneck. For scenarios where the client is remote, WebBase delivers 3.4Kpages/sec, surprisingly again with the CPU being the bottleneck in our setup.

In addition to being a research vehicle for research teams at Stanford and elsewhere, WebBase is providing the research community with highly customizable, open-source search engine components and fast streams of Web pages to use in developing effective algorithms for the acquisition, storage, and indexing of large-scale Web collections.

REFERENCES

ANDERSON, T. E., DAHLIN, M. D., NEEFE, J. M., PATTERSON, D. A., ROSELLI, D. S., AND WANG, R. Y. 1995. Serverless network file systems. In *Proc. of SOSP*.

ARASU, A., CHO, J., GARCIA-MOLINA, H., PAEPCKE, A., AND RAGHAVAN, S. 2001. Searching the web. *ACM Transactions on Internet Technology*. Also available at http://dbpubs.stanford.edu/pub/2000-37.

BRANDMAN, O., CHO, J., GARCIA-MOLINA, H., AND SHIVAKUMAR, N. 2000. Crawler-friendly web servers. In *Proceedings of the Workshop on Performance and Architecture of Web Servers (PAWS)*. Santa Clara, California. Held in conjunction with ACM SIGMETRICS 2000. Available at http://dbpubs.stanford.edu/pub/2000-25.

BRIN, S. AND PAGE, L. 1998. The anatomy of a large-scale hypertextual Web search engine. In *Proc. of the 7th Intl. World Wide Web Conf.*

BROWN, E. W., CALLAN, J. P., CROFT, W. B., AND MOSS, J. E. B. 1994. Supporting full-text information retrieval with a persistent object store. In *4th Intl. Conf. on Extending Database Technology*. 365–378.

BURNER, M. 1998. Crawling towards eternity: Building an archive of the world wide web. *Web Techniques Magazine 2,* 5 (May).

CHAKRABARTI, S. AND MUTHUKRISHNAN, S. 1996. Resource scheduling for parallel database and scientific applications. In *8th ACM Symposium on Parallel Algorithms and Architectures*. 329–335.

CHAKRABARTI, S., VAN DEN BERG, M., AND DOM, B. 1999. Focused crawling: A new approach to topic-specific web resource discovery. In *Proc. of WWW Conf.*

CHO, J. AND GARCIA-MOLINA, H. 2000a. The evolution of the web and implications for an incremental crawler. In *Proceedings of the Twenty-sixth International Conference on Very Large Databases*. Available at http://dbpubs.stanford.edu/pub/1999-22.

CHO, J. AND GARCIA-MOLINA, H. 2000b. The evolution of the web and implications for an incremental crawler. In *Proc. of VLDB Conf.*

CHO, J. AND GARCIA-MOLINA, H. 2000c. Synchronizing a database to improve freshness. In *Proceedings of the International Conference on Management of Data*. Available at http://dbpubs.stanford.edu/pub/1999-40.

CHO, J. AND GARCIA-MOLINA, H. 2000d. Synchronizing a database to improve freshness. In *Proc. of SIGMOD Conf.*

CHO, J. AND GARCIA-MOLINA, H. 2003. Estimating frequency of change. *ACM Transactions on Internet Technology 3,* 3 (August). Available at http://dbpubs.stanford.edu/pub/2000-4.

CHO, J., GARCIA-MOLINA, H., AND PAGE, L. 1998. Efficient crawling through URL ordering. In *Proc. of WWW Conf.*

CHO, J., SHIVAKUMAR, N., AND GARCIA-MOLINA, H. 2000. Finding replicated web collections. In *Proceedings of the International Conference on Management of Data*. Available at http://dbpubs.stanford.edu/pub/1999-64.

COFFMAN, JR., E., LIU, Z., AND WEBER, R. R. 1997. Optimal robot scheduling for web search engines. Tech. rep., INRIA.

DILIGENTI, M., COETZEE, F. M., LAWRENCE, S., GILES, C. L., AND GORI, M. 2000. Focused crawling using context graphs. In *Proc. of VLDB Conf.*

EICHMANN, D. 1994. The RBSE spider: Balancing effective search against web load. In *Proc. of WWW Conf.*

GARCIA-MOLINA, H., ULLMAN, J., AND WIDOM, J. 2000. *Database System Implementation*. Prentice-Hall.

GOOGLE INC. `http://www.google.com`.

GORSSMAN, D. A. AND DRISCOLL, J. R. 1992. Structuring text within a relation system. In *Proc. of the 3rd Intl. Conf. on Database and Expert System Applications*. 72–77.

HAVELIWALA, T. H., GIONIS, A., AND INDYK, P. 2000. Scalable techniques for clustering the web. In *Proc. of the 3rd Intl. Workshop on the Web and Databases (WebDB)*.

HAVELIWALA, T. H., GIONIS, A., KLEIN, D., AND TR INDYK, P. 2002. Evaluating strategies for similarity search on the web. In *Proc. of the 11th Intl. World Wide Web Conf.*

HEYDON, A. AND NAJORK, M. 1999. Mercator: A scalable, extensible web crawler. *Word Wide Web 2,* 4 (December), 219–229.

HIRAI, J., RAGHAVAN, S., GARCIA-MOLINA, H., AND PAEPCKE, A. 2000. WebBase: A repository of web pages. In *Proc. of the 9th Intl. World Wide Web Conf.* 277–293.

JEONG, B.-S. AND OMIECINSKI, E. 1995. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems 6,* 2 (February), 142–153.

KOSTER, M. 1994. A standard for robot exclusion. http://www.robotstxt.org/wc/norobots.html.

MCBRYAN, O. A. 1994. GENVL and WWWW: Tools for taming the web. In *Proc. of WWW Conf.*

MELNIK, S., RAGHAVAN, S., YANG, B., AND GARCIA-MOLINA, H. 2001. Building a distributed full-text index for the web. In *Proceedings of the Tenth International World-Wide Web Conference*.

MILLER, R. C. AND BHARAT, K. 1998. SPHINX: a framework for creating personal, site-specific web crawlers. In *Proc. of WWW Conf.*

MOFFAT, A. AND ZOBEL, J. 1996. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems 14,* 4 (October), 349–379.

NGOCVO, A. AND MOFFAT, A. 1998. Compressed inverted files with reduced decoding overheads. In *Proc. of the 21st Intl. Conf. on Research and Development in Information Retrieval*. 290–297.

OLSON, M., BOSTIC, K., AND SELTZER, M. 1999. Berkeley DB. In *Proc. of the 1999 Summer Usenix Technical Conf.*

PINKERTON, B. 1994. Finding what people want: Experiences with the web crawler. In *Proc. of WWW Conf.*

RAGHAVAN, S. 2003. Complex queries over web repositories. In *Proceedings of the 29th Conference on Very Large Databases (VLDB)*.

RAGHAVAN, S. AND GARCIA-MOLINA, H. 2003. Representing Web graphs. In *Proc. of the 19th International Conference on Data Engineering*.

RIBEIRO-NETO, B., MOURA, E. S., NEUBERT, M. S., AND ZIVIANI, N. 1999. Efficient distributed algorithms to build inverted files. In *Proc. of the 22nd ACM Conf. on Research and Development in Information Retrieval*. 105–112.

TANENBAUM, A. S. AND RENESSE, R. V. 1985. Distributed operating systems. *ACM Computing Surveys 17,* 4 (December).

The Internet Archive. The Internet Archive. http://www.archive.org/.

TOMASIC, A. AND GARCIA-MOLINA, H. 1993. Query processing and inverted indices in shared-nothing document information retrieval systems. *VLDB Journal 2,* 3, 243–275.

TOMASIC, A., GARCIA-MOLINA, H., AND SHOENS, K. 1994. Incremental update of inverted list for text document retrieval. In *Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data*. 289–300.

VILES, C. L. AND FRENCH, J. C. 1995. Dissemination of collection wide information in a distributed information retrieval system. In *Proc. of the 18th Intl. ACM Conf. on Research and Development in Information Retrieval*. 12–20.

WITTEN, I. H., MOFFAT, A., AND BELL, T. C. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images*, 2nd ed. Morgan Kauffman Publishing, San Francisco.

ZOBEL, J., MOFFAT, A., AND SACKS-DAVIS, R. 1992. An efficient indexing technique for full-text database systems. In *18th Intl. Conf. on Very Large Databases*. 352–362.