World Wide Web 2 (1999) 219–229

Mercator: A scalable, extensible Web crawler

Allan Heydon and Marc Najork

Compaq Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, USA
E-mail: {heydon,najork}@pa.dec.com

This paper describes Mercator, a scalable, extensible Web crawler written entirely in Java. Scalable Web crawlers are an important component of many Web services, but their design is not well-documented in the literature. We enumerate the major components of any scalable Web crawler, comment on alternatives and tradeoffs in their design, and describe the particular components used in Mercator. We also describe Mercator's support for extensibility and customizability. Finally, we comment on Mercator's performance, which we have found to be comparable to that of other crawlers for which performance numbers have been published.

1. Introduction

Designing a scalable Web crawler comparable to the ones used by the major search engines is a complex endeavor. However, due to the competitive nature of the search engine business, there are few papers in the literature describing the challenges and tradeoffs inherent in Web crawler design. This paper's main contribution is to fill that gap. It describes Mercator, a scalable, extensible Web crawler written entirely in Java.

By *scalable*, we mean that Mercator is designed to scale up to the entire Web, and has been used to fetch tens of millions of Web documents. We achieve scalability by implementing our data structures so that they use a bounded amount of memory, regardless of the size of the crawl. Hence, the vast majority of our data structures are stored on disk, and small parts of them are stored in memory for efficiency.

By *extensible*, we mean that Mercator is designed in a modular way, with the expectation that new functionality will be added by third parties. In practice, it has been used to create a snapshot of the Web pages on our corporate intranet, to collect a variety of statistics about the Web, and to perform a series of random walks of the Web [Henzinger *et al.* 1999].

One of the initial motivations of this work was to collect statistics about the Web. There are many statistics that might be of interest, such as the size and the evolution of the URL space, the distribution of Web servers over top-level domains, the lifetime and change rate of documents, and so on. However, it is hard to know *a priori* exactly which statistics are interesting, and topics of interest may change over time. Mercator makes it easy to collect new statistics – or more generally, to be configured for different crawling tasks – by allowing users to provide their own modules for processing downloaded documents. For example, when we designed Mercator, we did not anticipate the possibility of using it for the random walk application cited above. Despite the differences between random walking and traditional crawling, we were able to reconfigure Mercator as a random walker without modifying the crawler's core, merely by plugging in modules totaling 360 lines of Java source code.

The remainder of the paper is structured as follows. The next section surveys related work. Section 3 describes the main components of a scalable Web crawler, the alternatives and tradeoffs in their design, and the particular choices we made in Mercator. Section 4 describes Mercator's support for extensibility. Section 5 describes some of the Web crawling problems that arise due to the inherent anarchy of the Web. Section 6 reports on performance measurements and Web statistics collected during a recent extended crawl. Finally, section 7 offers our conclusions.

2. Related work

Web crawlers – also known as robots, spiders, worms, walkers, and wanderers – are almost as old as the Web itself [Koster]. The first crawler, Matthew Gray's Wanderer, was written in the spring of 1993, roughly coinciding with the first release of NCSA Mosaic [Gray]. Several papers about Web crawling were presented at the first two World Wide Web conferences [Eichmann 1994; McBryan 1994; Pinkerton 1994]. However, at the time, the Web was two to three orders of magnitude smaller than it is today, so those systems did not address the scaling problems inherent in a crawl of today's Web.

Obviously, all of the popular search engines use crawlers that must scale up to substantial portions of the Web. However, due to the competitive nature of the search engine business, the designs of these crawlers have not been publicly described. There are two notable exceptions: the Google crawler and the Internet Archive crawler. Unfortunately, the descriptions of these crawlers in the literature are too terse to enable reproducibility.

The Google search engine is a distributed system that uses multiple machines for crawling [Brin and Page 1998; Google]. The crawler consists of five functional components running in different processes. A *URL server process*

reads URLs out of a file and forwards them to multiple crawler processes. Each *crawler process* runs on a different machine, is single-threaded, and uses asynchronous I/O to fetch data from up to 300 Web servers in parallel. The crawlers transmit downloaded pages to a single *StoreServer process*, which compresses the pages and stores them to disk. The pages are then read back from disk by an *indexer process*, which extracts links from HTML pages and saves them to a different disk file. A *URL resolver process* reads the link file, derelativizes the URLs contained therein, and saves the absolute URLs to the disk file that is read by the URL server. Typically, three to four crawler machines are used, so the entire system requires between four and eight machines.

The Internet Archive also uses multiple machines to crawl the Web [Burner 1977; InternetArchive]. Each crawler process is assigned up to 64 sites to crawl, and no site is assigned to more than one crawler. Each single-threaded crawler process reads a list of seed URLs for its assigned sites from disk into per-site queues, and then uses asynchronous I/O to fetch pages from these queues in parallel. Once a page is downloaded, the crawler extracts the links contained in it. If a link refers to the site of the page it was contained in, it is added to the appropriate site queue; otherwise it is logged to disk. Periodically, a batch process merges these logged "cross-site" URLs into the site-specific seed sets, filtering out duplicates in the process.

In the area of extensible Web crawlers, Miller and Bharat's SPHINX system [Miller and Bharat 1998] provides some of the same customizability features as Mercator. In particular, it provides a mechanism for limiting which pages are crawled, and it allows customized document processing code to be written. However, SPHINX is targeted towards site-specific crawling, and therefore is not designed to be scalable.

3. Architecture of a scalable Web crawler

The basic algorithm executed by any Web crawler takes a list of *seed URLs* as its input and repeatedly executes the following steps. Remove a URL from the URL list, determine the IP address of its host name, download the corresponding document, and extract any links contained in it. For each of the extracted links, ensure that it is an absolute URL (derelativizing it if necessary), and add it to the list of URLs to download, provided it has not been encountered before. If desired, process the downloaded document in other ways (e.g., index its content). This basic algorithm requires a number of functional components:

- a component (called the *URL frontier*) for storing the list of URLs to download,
- a component for resolving host names into IP addresses,
- a component for downloading documents using the HTTP protocol,
- a component for extracting links from HTML documents, and
- a component for determining whether a URL has been encountered before.

This section describes how Mercator refines this basic algorithm, and the particular implementations we chose for the various components. Where appropriate, we comment on design alternatives and the tradeoffs between them.

3.1. Mercator's components

Figure 1 shows Mercator's main components. Crawling is performed by multiple worker threads, typically numbering in the hundreds. Each worker repeatedly performs the steps needed to download and process a document. The first step of this loop ❶ is to remove an absolute URL from the shared URL frontier for downloading.

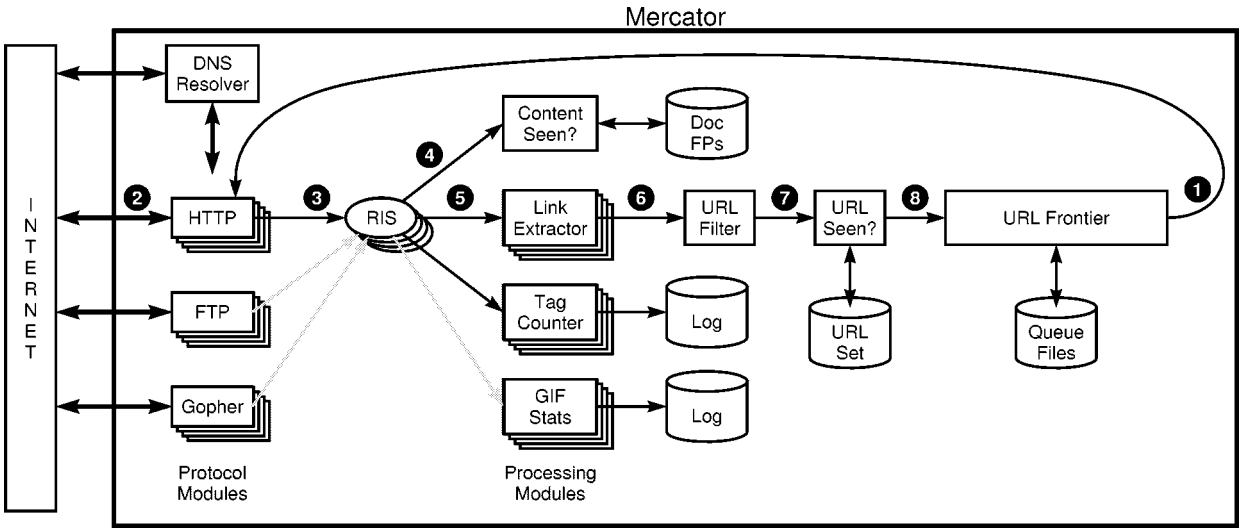

Figure 1. Mercator's main components.

An absolute URL begins with a *scheme* (e.g., "http"), which identifies the network protocol that should be used to download it. In Mercator, these network protocols are implemented by *protocol modules*. The protocol modules to be used in a crawl are specified in a user-supplied configuration file, and are dynamically loaded at the start of the crawl. The default configuration includes protocol modules for HTTP, FTP, and Gopher. As suggested by the tiling of the protocol modules in figure 1, there is a separate instance of each protocol module per thread, which allows each thread to access local data without any synchronization.

Based on the URL's scheme, the worker selects the appropriate protocol module for downloading the document. It then invokes the protocol module's *fetch* method, which downloads the document from the Internet ❷ into a per-thread *RewindInputStream* ❸ (or RIS for short). A RIS is an I/O abstraction that is initialized from an arbitrary input stream, and that subsequently allows that stream's contents to be re-read multiple times.

Once the document has been written to the RIS, the worker thread invokes the *content-seen test* to determine whether this document (associated with a different URL) has been seen before ❹. If so, the document is not processed any further, and the worker thread removes the next URL from the frontier.

Every downloaded document has an associated MIME type. In addition to associating schemes with protocol modules, a Mercator configuration file also associates MIME types with one or more *processing modules*. A processing module is an abstraction for processing downloaded documents, for instance extracting links from HTML pages, counting the tags found in HTML pages, or collecting statistics about GIF images. Like protocol modules, there is a separate instance of each processing module per thread. In general, processing modules may have side-effects on the state of the crawler, as well as on their own internal state.

Based on the downloaded document's MIME type, the worker invokes the *process* method of each processing module associated with that MIME type ❺. For example, the Link Extractor and Tag Counter processing modules in figure 1 are used for text/html documents, and the GIF Stats module is used for image/gif documents.

By default, a processing module for extracting links is associated with the MIME type text/html. The *process* method of this module extracts all links from an HTML page. Each link is converted into an absolute URL, and tested against a user-supplied *URL filter* to determine if it should be downloaded ❻. If the URL passes the filter, the worker performs the *URL-seen test* ❼, which checks if the URL has been seen before, namely, if it is in the URL frontier or has already been downloaded. If the URL is new, it is added to the frontier ❽.

The above description omits several important implementation details. Designing data structures that can efficiently handle hundreds of millions of entries poses many engineering challenges. Central to these concerns is the need to balance memory use and performance. In the remainder of this section, we describe how we address this time-space tradeoff in several of Mercator's main data structures.

3.2. The URL frontier

The URL frontier is the data structure that contains all the URLs that remain to be downloaded. Most crawlers work by performing a breath-first traversal of the Web, starting from the pages in the seed set. Such traversals are easily implemented by using a FIFO queue.

In a standard FIFO queue, elements are dequeued in the order they were enqueued. In the context of Web crawling, however, matters are complicated by the fact that it is considered socially unacceptable to have multiple HTTP requests pending to the same server. If multiple requests are to be made in parallel, the queue's *remove* operation should not simply return the head of the queue, but rather a URL close to the head whose host has no outstanding request.

To implement this politeness constraint, the default version of Mercator's URL frontier is actually implemented by a collection of distinct FIFO subqueues. There are two important aspects to how URLs are added to and removed from these queues. First, there is one FIFO subqueue per worker thread. That is, each worker thread removes URLs from exactly one of the FIFO subqueues. Second, when a new URL is added, the FIFO subqueue in which it is placed is determined by the URL's canonical host name. Together, these two points imply that at most one worker thread will download documents from a given Web server. This design prevents Mercator from overloading a Web server, which could otherwise become a bottleneck of the crawl.

In actual World Wide Web crawls, the size of the crawl's frontier numbers in the hundreds of millions of URLs. Hence, the majority of the URLs must be stored on disk. To amortize the cost of reading from and writing to disk, our FIFO subqueue implementation maintains fixed-size enqueue and dequeue buffers in memory; our current implementation uses buffers that can hold 600 URLs each.

As described in section 4 below, the URL frontier is one of Mercator's "pluggable" components, meaning that it can easily be replaced by other implementations. For example, one could implement a URL frontier that uses a ranking of URL importance (such as the PageRank metric [Brin and Page 1998]) to order URLs in the frontier set. Cho, Garcia-Molina, and Page have performed simulated crawls showing that such ordering can improve crawling effectiveness [Cho *et al.* 1998].

3.3. The HTTP protocol module

The purpose of a protocol module is to fetch the document corresponding to a given URL using the appropriate network protocol. Network protocols supported by Mercator include HTTP, FTP, and Gopher.

Courteous Web crawlers implement the Robots Exclusion Protocol, which allows Web masters to declare parts of their sites off limits to crawlers [RobotsExclusion]. The Robots Exclusion Protocol requires a Web crawler to fetch a special document containing these declarations from a Web site before downloading any real content from it. To avoid downloading this file on every request, Mercator's HTTP protocol module maintains a fixed-sized cache mapping host names to their robots exclusion rules. By default, the cache is limited to 2^{18} entries, and uses an LRU replacement strategy.

Our initial HTTP protocol module used the HTTP support provided by the JDK 1.1 Java class libraries. However, we soon discovered that this implementation did not allow us to specify any timeout values on HTTP connections, so a malicious Web server could cause a worker thread to hang indefinitely. Also, the JDK implementation was not particularly efficient. Therefore, we wrote our own "lean and mean" HTTP protocol module; its requests time out after 1 minute, and it has minimal synchronization and allocation overhead.

3.4. Rewind input stream

Mercator's design allows the same document to be processed by multiple processing modules. To avoid reading a document over the network multiple times, we cache the document locally using an abstraction called a *RewindInputStream* (RIS).

A RIS is an input stream with an *open* method that reads and caches the entire contents of a supplied input stream (such as the input stream associated with a socket). A RIS caches small documents (64 KB or less) entirely in memory, while larger documents are temporarily written to a backing file. The RIS constructor allows a client to specify an upper limit on the size of the backing file as a safeguard against malicious Web servers that might return documents of unbounded size. By default, Mercator sets this limit to 1 MB. In addition to the functionality provided by normal input streams, a RIS also provides a method for rewinding its position to the beginning of the stream, and various lexing methods that make it easy to build MIME-type-specific parsers.

Each worker thread has an associated RIS, which it reuses from document to document. After removing a URL from the frontier, a worker passes that URL to the appropriate protocol module, which initializes the RIS from a network connection to contain the document's contents. The worker then passes the RIS to all relevant processing modules, rewinding the stream before each module is invoked.

3.5. Content-seen test

Many documents on the Web are available under multiple, different URLs. There are also many cases in which documents are mirrored on multiple servers. Both of these effects will cause any Web crawler to download the same document contents multiple times. To prevent processing a document more than once, a Web crawler may wish to perform a *content-seen test* to decide if the document has already been processed. Using a content-seen test makes it possible to suppress link extraction from mirrored pages, which may result in a significant reduction in the number of pages that need to be downloaded (section 5 below elaborates on these points). Mercator includes just such a content-seen test, which also offers the side benefit of allowing us to keep statistics about the fraction of downloaded documents that are duplicates of pages that have already been downloaded.

The content-seen test would be prohibitively expensive in both space and time if we saved the complete contents of every downloaded document. Instead, we maintain a data structure called the *document fingerprint set* that stores a 64-bit checksum of the contents of each downloaded document. We compute the checksum using Broder's implementation [Broder 1993] of Rabin's fingerprinting algorithm [Rabin 1981]. Fingerprints offer provably strong probabilistic guarantees that two different strings will not have the same fingerprint. Other checksum algorithms, such as MD5 and SHA, do not offer such provable guarantees, and are also more expensive to compute than fingerprints.

When crawling the entire Web, the document fingerprint set will obviously be too large to be stored entirely in memory. Unfortunately, there is very little locality in the requests made on the document fingerprint set, so caching such requests has little benefit. We therefore maintain two independent sets of fingerprints: a small hash table kept in memory, and a large sorted list kept in a single disk file.

The content-seen test first checks if the fingerprint is contained in the in-memory table. If not, it has to check if the fingerprint resides in the disk file. To avoid multiple disk seeks and reads per disk search, Mercator performs an interpolated binary search of an in-memory index of the disk file to identify the disk block on which the fingerprint would reside if it were present. It then searches the appropriate disk block, again using interpolated binary search. We use a buffered variant of Java's random access files, which guarantees that searching through one disk block causes at most two kernel calls (one *seek* and one *read*). We use a customized data structure instead of a more generic data structure such as a B-tree because of this guarantee. It is worth noting that Mercator's ability to be dynamically configured (see section 4 below) would easily allow someone to replace our implementation with a different one based on B-trees.

When a new fingerprint is added to the document fingerprint set, it is added to the in-memory table. When this table fills up, its contents are merged with the fingerprints on disk, at which time the in-memory index of the disk file is updated as well. To guard against races, we use a readers-writer lock that controls access to the disk file. Threads must hold a read share of the lock while reading from the file, and must hold the write lock while writing to it.

3.6. URL filters

The URL filtering mechanism provides a customizable way to control the set of URLs that are downloaded. Before adding a URL to the frontier, the worker thread consults the user-supplied URL filter. The URL filter class has a single *crawl* method that takes a URL and returns a boolean value indicating whether or not to crawl that URL. Mercator includes a collection of different URL filter subclasses that provide facilities for restricting URLs by domain, prefix, or protocol type, and for computing the conjunction, disjunction, or negation of other filters. Users may also supply their own custom URL filters, which are dynamically loaded at start-up.

3.7. Domain name resolution

Before contacting a Web server, a Web crawler must use the Domain Name Service (DNS) to map the Web server's host name into an IP address. DNS name resolution is a well-documented bottleneck of most Web crawlers. This bottleneck is exacerbated in any crawler, like Mercator or the Internet Archive crawler, that uses DNS to canonicalize the host names of newly discovered URLs before performing the URL-seen test on them.

We tried to alleviate the DNS bottleneck by caching DNS results, but that was only partially effective. After some probing, we discovered that the Java interface to DNS lookups is synchronized. Further investigation revealed that the DNS interface on most flavors of Unix (i.e., the *gethostbyname* function provided as part of the Berkeley Internet Name Domain (BIND) distribution [BIND]) is also synchronized. This meant that only one DNS request on an uncached name could be outstanding at once. The cache miss rate is high enough that this limitation causes a bottleneck.

To work around these problems, we wrote our own multi-threaded DNS resolver class and integrated it into Mercator. The resolver forwards DNS requests to a local name server, which does the actual work of contacting the authoritative server for each query. Because multiple requests can be made in parallel, our resolver can resolve host names much more rapidly than either the Java or Unix resolvers.

This change led to a significant crawling speedup. Before making the change, performing DNS lookups accounted for 87% of each thread's elapsed time. Using our custom resolver reduced that elapsed time to 25%. (Note that the actual number of CPU cycles spent on DNS resolution is extremely low. Most of the elapsed time is spent waiting on remote DNS servers.) Moreover, because our resolver can perform resolutions in parallel, DNS is no longer a bottleneck; if it were, we would simply increase the number of worker threads.

3.8. URL-seen test

In the course of extracting links, any Web crawler will encounter multiple links to the same document. To avoid downloading and processing a document multiple times, a URL-seen test must be performed on each extracted link before adding it to the URL frontier. (An alternative design would be to instead perform the URL-seen test when the URL is removed from the frontier, but this approach would result in a much larger frontier.)

To perform the URL-seen test, we store all of the URLs seen by Mercator in canonical form in a large table called the *URL set*. Again, there are too many entries for them all to fit in memory, so like the document fingerprint set, the URL set is stored mostly on disk.

To save space, we do not store the textual representation of each URL in the URL set, but rather a fixed-sized checksum. Unlike the fingerprints presented to the content-seen test's document fingerprint set, the stream of URLs tested against the URL set has a non-trivial amount of locality. To reduce the number of operations on the backing disk file, we therefore keep an in-memory cache of popular URLs. The intuition for this cache is that links to some URLs are quite common, so caching the popular ones in memory will lead to a high in-memory hit rate.

In fact, using an in-memory cache of 2^{18} entries and the LRU-like clock replacement policy, we achieve an overall hit rate on the in-memory cache of 66.2%, and a hit rate of 9.5% on the table of recently-added URLs, for a net hit rate of 75.7%. Moreover, of the 24.3% of requests that miss in both the cache of popular URLs and the table of recently-added URLs, about $1/3$ produce hits on the buffer in our random access file implementation, which also resides in user-space. The net result of all this buffering is that each membership test we perform on the URL set results in an average of 0.16 *seek* and 0.17 *read* kernel calls (some fraction of which are served out of the kernel's file system buffers). So, each URL set membership test induces one-sixth as many kernel calls as a membership test on the document fingerprint set. These savings are purely due to the amount of URL locality (i.e., repetition of popular URLs) inherent in the stream of URLs encountered during a crawl.

However, due to the prevalence of relative URLs in Web pages, there is a second form of locality in the stream of discovered URLs, namely, host name locality. Host name locality arises because many links found in Web pages are to different documents on the same server. Unfortunately, computing a URL's checksum by simply fingerprinting its textual representation would cause this locality to be lost. To preserve the locality, we compute the checksum of a URL by merging two independent fingerprints: one of the URL's host name, and the other of the complete URL. These two fingerprints are merged so that the high-order bits of the checksum derive from the host name fingerprint. As a result, checksums for URLs with the same host component are numerically close together. So, the host name locality

in the stream of URLs translates into access locality on the URL set's backing disk file, thereby allowing the kernel's file system buffers to service read requests from memory more often. On extended crawls in which the size of the URL set's backing disk file significantly exceeds the size of the kernel's file system buffers, this technique results in a significant reduction in disk load, and hence, in a significant performance improvement.

The Internet Archive crawler implements the URL-seen test using a different data structure called a *bloom filter* [Bloom 1970]. A bloom filter is a probabilistic data structure for set membership testing that may yield false positives. The set is represented by a large bit vector. An element is added to the set by computing n hash functions of the element and setting the corresponding bits. An element is deemed to be in the set if the bits at all n of the element's hash locations are set. Hence, a document may incorrectly be deemed to be in the set, but false negatives are not possible.

The disadvantage to using a bloom filter for the URL-seen test is that each false positive will cause the URL not to be added to the frontier, and therefore the document will never be downloaded. The chance of a false positive can be reduced by making the bit vector larger. The Internet Archive crawler uses 10 hash functions and a separate 32 KB bit vector for each of the sites currently being crawled. For the batch phase in which non-local URLs are merged, a much larger 2 GB bit vector is used. As the Web grows, the batch process will have to be run on a machine with larger amounts of memory, or disk-based data structures will have to be used. By contrast, our URL-seen test does not admit false positives, and it uses a bounded amount of memory, regardless of the size of the Web.

3.9. Synchronous vs. asynchronous I/O

Both the Google and Internet Archive crawlers use single-threaded crawling processes and asynchronous I/O to perform multiple downloads in parallel. In contrast, Mercator uses a multi-threaded process in which each thread performs synchronous I/O. These two techniques are different means for achieving the same effect.

The main advantage to using multiple threads and synchronous I/O is that it leads to a much simpler program structure. Switching between threads of control is left to the operating system's thread scheduler, rather than having to be coded in the user program. The sequence of tasks executed by each worker thread is much more straightforward and self-evident.

One strength of the Google and the Internet Archive crawlers is that they are designed from the ground up to scale to multiple machines. However, whether a crawler is distributed or not is orthogonal to the choice between synchronous and asynchronous I/O. It would not be too difficult to adapt Mercator to run on multiple machines, while still using synchronous I/O and multiple threads within each process.

3.10. Checkpointing

A crawl of the entire Web takes weeks to complete. To guard against failures, Mercator writes regular snapshots of its state to disk. These snapshots are orthogonal to Mercator's other disk-based data structures. An interrupted or aborted crawl can easily be restarted from the latest checkpoint.

We define a general checkpointing interface that is implemented by the Mercator classes constituting the crawler core. User-supplied protocol or processing modules are also required to implement the checkpointing interface.

Checkpoints are coordinated using a global readers-writer lock. Each worker thread acquires a read share of the lock while processing a downloaded document. At regular intervals, typically once a day, Mercator's main thread acquires the write lock, so it is guaranteed to be running in isolation. Once it has acquired the lock, the main thread arranges for the checkpoint methods to be called on Mercator's core classes and all user-supplied modules.

4. Extensibility

As mentioned previously, Mercator is an extensible crawler. In practice, this means two things. First, Mercator can be extended with new functionality. For example, new protocol modules may be provided for fetching documents according to different network protocols, or new processing modules may be provided for processing downloaded documents in customized ways. Second, Mercator can easily be reconfigured to use different versions of most of its major components. In particular, different versions of the URL frontier (section 3.2), document fingerprint set (section 3.5), URL filter (section 3.6), and URL set (section 3.8) may all be dynamically "plugged into" the crawler. In fact, we have written multiple versions of each of these components, which we employ for different crawling tasks.

Making an extensible system such as Mercator requires three ingredients:

- The interface to each of the system's components must be well-specified. In Mercator, the interface to each component is defined by an *abstract* class, some of whose methods are also *abstract*. Any component implementation is required to be a subclass of this abstract class that provides implementations for the abstract methods.
- A mechanism must exist for specifying how the crawler is to be configured from its various components. In Mercator, this is done by supplying a *configuration file* to the crawler when it starts up. Among other things, the configuration file specifies which additional protocol and processing modules should be used, as well as the concrete implementation to use for each of the crawler's "pluggable" components. When Mercator is started, it reads the configuration file, uses Java's dynamic class loading feature to dynamically instantiate the necessary

components, plugs these instances into the appropriate places in the crawler core's data structures, and then begins the crawl. Suitable defaults are defined for all components.

- Sufficient infrastructure must exist to make it easy to write new components. In Mercator, this infrastructure consists of a rich set of utility libraries together with a set of existing pluggable components. An object-oriented language such as Java makes it easy to construct a new component by subclassing an existing component and overriding some of its methods. For example, our colleague Mark Manasse needed information about the distribution of HTTP 401 return codes across Web servers. He was able to obtain this data by subclassing Mercator's default HTTP protocol component, and by using one of our standard histogram classes to maintain the statistics. As a result, his custom HTTP protocol component required only 58 lines of Java source code.

To demonstrate Mercator's extensibility, we now describe some of the extensions we have written.

4.1. Protocol and processing modules

By default, Mercator will crawl the Web by fetching documents using the HTTP protocol, extracting links from documents of type text/html. However, aside from extracting links, it does not process the documents in any way. To fetch documents using additional protocols or to process the documents once they are fetched, new protocol and processing modules must be supplied.

The abstract *Protocol* class includes only two methods. The *fetch* method downloads the document corresponding to a given URL, and the *newURL* method parses a given string, returning a structured *URL* object. The latter method is necessary because URL syntax varies from protocol to protocol. In addition to the HTTP protocol module, we have also written protocol modules for fetching documents using the FTP and Gopher protocols.

The abstract *Analyzer* class is the superclass for all processing modules. It defines a single *process* method. This method is responsible for reading the document and processing it appropriately. The method may make method calls on other parts of the crawler (e.g., to add newly discovered URLs to the frontier). Analyzers often keep private state or write data to the disk.

We have written a variety of different *Analyzer* subclasses. Some of these analyzers keep statistics, such as our *GifStats* and *TagCounter* subtypes. Other processing modules simply write the contents of each downloaded document to disk. Our colleague Raymie Stata has written such a module; the files it writes are in a form suitable to be read by the AltaVista indexing software. As another experiment, we wrote a *WebLinter* processing module that runs the *Weblint* program on each downloaded HTML page to check it for errors, logging all discovered errors to a file. These processing modules range in size from 70 to 270 lines of Java code, including comments; the majority are less than 100 lines long.

4.2. Alternative URL frontier implementation

In section 3.2, we described one implementation of the URL frontier data structure. However, when we ran that implementation on a crawl of our corporate intranet, we discovered that it had the drawback that multiple hosts might be assigned to the same worker thread, while other threads were left idle. This situation is more likely to occur on an intranet because intranets typically contain a substantially smaller number of hosts than the internet at large.

To restore the parallelism lost by our initial URL frontier implementation, we wrote an alternative URL frontier component that dynamically assigns hosts to worker threads. Like our initial implementation, the second version guarantees that at most one worker thread will download documents from any given Web server at once; moreover, it maximizes the number of busy worker threads within the limits of this guarantee. In particular, all worker threads will be busy so long as the number of different hosts in the frontier is at least the number of worker threads. The second version is well-suited to host-limited crawls (such as intranet crawls), while the initial version is preferable for internet crawls, since it does not have the overheads required by the second version to maintain a dynamic mapping from host names to worker threads.

4.3. Configuring Mercator as a random walker

We have used Mercator to perform random walks of the Web in order to gather a sample of Web pages; the sampled pages were used to measure the quality of search engines [Henzinger *et al.* 1999]. A random walk starts at a random page taken from a set of seeds. The next page to fetch is selected by choosing a random link from the current page. The process continues until it arrives at a page with no links, at which time the walk is restarted from a new random seed page. The seed set is dynamically extended by the newly discovered pages, and cycles are broken by performing random restarts every so often.

Performing a random walk of the Web is quite different from an ordinary crawl for two reasons. First, a page may be revisited multiple times during the course of a random walk. Second, only one link is followed each time a page is visited. In order to support random walking, we wrote a new URL frontier class that does not maintain a set of all added URLs, but instead records only the URLs discovered on each thread's most recently fetched page. Its *remove* method selects one of these URLs at random and returns it. To allow pages to be processed multiple times, we also replaced the document fingerprint set by a new version that never rejects documents as already having been seen. Finally, we subclassed the default *LinkExtractor* class to perform extra logging. The new classes are "plugged into" the crawler core at runtime using the extension mechanism

described above. In total, the new classes required for random walking amount to 360 lines of Java source code.

5. Crawler traps and other hazards

In the course of our experiments, we had to overcome several pitfalls that would otherwise cause Mercator to download more documents than necessary. Although the Web contains a finite number of static documents (i.e., documents that are not generated on-the-fly), there are an infinite number of retrievable URLs. Three frequent causes of this inflation are URL aliases, session IDs embedded in URLs, and crawler traps. This section describes those problems and the techniques we use to avoid them.

5.1. URL aliases

Two URLs are aliases for each other if they refer to the same content. There are four causes of URL aliases:

- *Host name aliases*
  Host name aliases occur when multiple host names correspond to the same IP address. For example, the host names `coke.com` and `cocacola.com` both correspond to the host whose IP address is 208.134.241.178. Hence, every document served by that machine's Web server has at least three different URLs (one using each of the two host names, and one using the IP address).
  Before performing the URL-seen test, we canonicalize a host name by issuing a DNS request containing a CNAME ("canonical name") and an A ("addresses") query. We use the host's canonical name if one is returned; otherwise, we use the smallest returned IP address.
  It should be mentioned that this technique may be too aggressive for some *virtual domains*. Many Internet service providers (ISPs) serve up multiple domains from the same Web server, using the "Host" header field of the HTTP request to recover the host portion of the URL. If an ISP does not provide distinct CNAME records for these virtual domains, URLs from these domains will be collapsed into the same host space by our canonicalization process.
- *Omitted port numbers*
  If a port number is not specified in a URL, a protocol-specific default value is used. We therefore insert the default value, such as 80 in the case of HTTP, before performing the URL-seen test on it.
- *Alternative paths on the same host*
  On a given host, there may be multiple paths to the same file. For example, the two URLs `http://www.digital.com/index.html` and `http://www.digital.com/home.html` both refer to the same file. One common cause of this phenomenon is the use of symbolic links within the server machine's file system.
- *Replication across different hosts*
  Finally, multiple copies of a document may reside on different Web servers. Mirror sites are a common instance of this phenomenon, as are multiple Web servers that access the same shared file system.

In the latter two cases, we cannot avoid downloading duplicate documents. However, we do avoid processing all but the first copy by using the content-seen test described in section 3.5. During an eight-day crawl described in section 6 below, 8.5% of the documents we fetched were duplicates. Had we not performed the content-seen test, the number of unnecessary downloads would have been much higher, since we would have followed links from duplicate documents to other documents that are likely to be duplicates as well.

5.2. Session IDs embedded in URLs

To track the browsing behavior of their visitors, a number of Web sites embed dynamically-generated session identifiers within the links of returned pages. Session IDs create a potentially infinite set of URLs that refer to the same document. With the advent of cookies, the use of embedded session IDs has diminished, but they are still used, partly because most browsers have a facility for disabling cookies.

Embedded session IDs represent a special case of alternative paths on the same host as described in the previous section. Thus, Mercator's document fingerprinting technique prevents excessive downloading due to embedded session IDs, although it is not powerful enough to automatically detect and remove them.

5.3. Crawler traps

A *crawler trap* is a URL or set of URLs that cause a crawler to crawl indefinitely. Some crawler traps are unintentional. For example, a symbolic link within a file system can create a cycle. Other crawler traps are introduced intentionally. For example, people have written traps using CGI programs that dynamically generate an infinite Web of documents. The motivations behind such traps vary. Anti-spam traps are designed to catch crawlers used by "Internet marketeers" (better known as spammers) looking for e-mail addresses, while other sites use traps to catch search engine crawlers so as to boost their search ratings.

We know of no automatic technique for avoiding crawler traps. However, sites containing crawler traps are easily noticed due to the large number of documents discovered there. A human operator can verify the existence of a trap and manually exclude the site from the crawler's purview using the customizable URL filter described in section 3.6.

6. Results of an extended crawl

This section reports on Mercator's performance during an eight-day crawl, and presents some statistics about the

Web collected during that crawl. In our analysis of Mercator's performance, we contrast it with the performance of the Google and Internet Archive crawlers. We make no attempt to adjust for different hardware configurations since the papers describing the two other crawlers do not contain enough information to do so. Our main intention in presenting this performance data is to convey the relative speeds of the various crawlers.

6.1. Performance

Our production crawling machine is a Digital Ultimate Workstation with two 533 MHz Alpha processors, 2 GB of RAM, 118 GB of local disk, and a 100 Mbit/sec FDDI connection to the Internet. We run Mercator under *srcjava*, a Java runtime developed at our lab [Ghemawat]. Running on this platform, a Mercator crawl run in May 1999 made 77.4 million HTTP requests in 8 days, achieving an average download rate of 112 documents/sec and 1,682 KB/sec.

These numbers indicate that Mercator's performance compares favorably with that of the Google and the Internet Archive crawlers. The Google crawler is reported to have issued 26 million HTTP requests over 9 days, averaging 33.5 docs/sec and 200 KB/sec [Brin and Page 1998]. This crawl was performed using four machines running crawler processes, and at least one more machine running the other processes. The Internet Archive crawler, which also uses multiple crawler machines, is reported to fetch 4 million HTML docs/day, the average HTML page being 5 KB [Smith 1997]. This download rate is equivalent to 46.3 HTML docs/sec and 231 KB/sec. It is worth noting that Mercator fetches not only HTML pages, but documents of all other MIME types as well. This effect more than doubles the size of the average document downloaded by Mercator as compared to the other crawlers.

Achieving the performance numbers described above required considerable optimization work. In particular, we spent a fair amount of time overcoming performance limitations of the Java core libraries [Heydon and Najork 1999].

We used DCPI, the Digital Continuous Profiling Infrastructure [DCPI], to measure where Mercator spends CPU cycles. DCPI is a freely available tool that runs on Alpha platforms. It can be used to profile both the kernel and user-level processes, and it provides CPU cycle accounting at the granularity of processes, procedures, and individual instructions. We found that Mercator spends 37% of its cycles in JIT-compiled Java bytecode, 19% in the Java runtime, and 44% in the Unix kernel. The Mercator method accounting for the most cycles is the one that fingerprints the contents of downloaded documents (2.3% of all cycles).

6.2. Selected Web statistics

We now present statistics collected about the Web during the eight-day crawl described above.

One interesting statistic is the distribution of outcomes from HTTP requests. Roughly speaking, each URL removed from the frontier causes a single HTTP request.

Table 1
Relationship between the total number of URLs removed from the frontier and the total number of HTTP requests.

	No. of URLs removed	76,732,515
+	No. of robots.txt requests	3,675,634
−	No. of excluded URLs	3,050,768
=	No. of HTTP requests	77,357,381

Table 2
Breakdown of HTTP status codes.

Code	Meaning	Number	Percent
200	OK	65,790,953	87.03%
404	Not found	5,617,491	7.43%
302	Moved temporarily	2,517,705	3.33%
301	Moved permanently	842,875	1.12%
403	Forbidden	322,042	0.43%
401	Unauthorized	223,843	0.30%
500	Internal server error	83,744	0.11%
406	Not acceptable	81,091	0.11%
400	Bad request	65,159	0.09%
	Other	48,628	0.06%
	Total	75,593,531	100.0%

However, there are two wrinkles to that equation, both related to the Robots Exclusion Protocol [RobotsExclusion]. First, before fetching a document, Mercator has to verify whether it has been excluded from downloading the document by the target site's robots.txt file. If the appropriate robots.txt data is not in Mercator's cache (described in section 3.3), it must be downloaded, causing an extra HTTP request. Second, if the robots.txt file indicates that Mercator should not download the document, no HTTP request is made for the document. Table 1 relates the number of URLs removed from the frontier to the total number of HTTP requests.

1.8 million of the 77.4 million HTTP requests (2.3%) did not result in a response, either because the host could not be contacted or some other network failure occurred. Table 2 gives a breakdown of the HTTP status codes for the remaining 75.6 million requests. We were somewhat surprised at the relatively low number of 404 status codes; we had expected to discover a higher percentage of broken links.

Of the 65.8 million documents that were successfully downloaded, 80% were between 1 K and 32 K bytes in size. Figure 2 is a histogram showing the document size distribution. In this figure, the documents are distributed over 21 bins labeled with exponentially increasing document sizes; a document of size n is placed in the rightmost bin with a label not greater than n.

According to our content-seen test, 8.5% of the successful HTTP requests (i.e., those with status code 200) were duplicates[1]. Of the 60 million unique documents that were successfully downloaded, the vast majority were HTML pages, followed in popularity by GIF and JPEG images.

[1] This figure ignores the successful HTTP requests that were required to fetch robots.txt files.

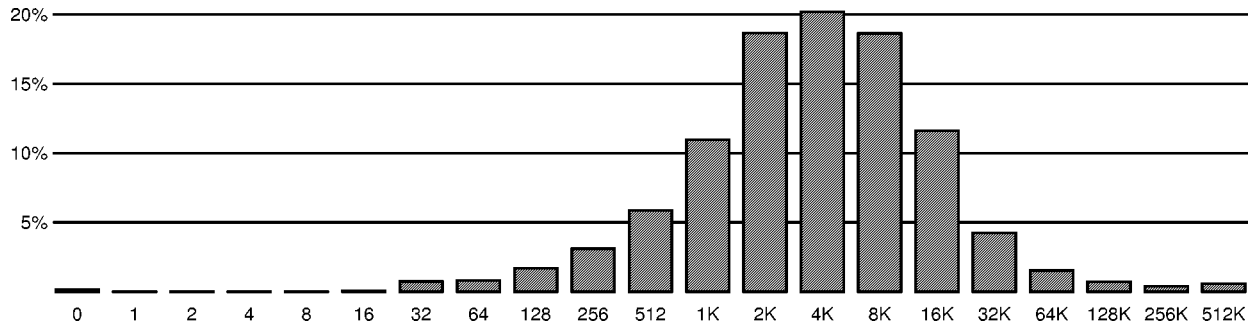

Figure 2. Histogram of the sizes of successfully downloaded documents.

Table 3
Distribution of MIME types.

MIME type	Number	Percent
text/html	41,490,044	69.2%
image/gif	10,729,326	17.9%
image/jpeg	4,846,257	8.1%
text/plain	869,911	1.5%
application/pdf	540,656	0.9%
audio/x-pn-realaudio	269,384	0.4%
application/zip	213,089	0.4%
application/postscript	159,869	0.3%
other	829,410	1.4%
Total	59,947.946	100.0%

Table 3 shows the distribution of the most popular MIME types.

7. Conclusions

Scalable Web crawlers are an important component of many Web services, but they have not been very well documented in the literature. Building a scalable crawler is a non-trivial endeavor because the data manipulated by the crawler is too big to fit entirely in memory, so there are performance issues relating to how to balance the use of disk and memory. This paper has enumerated the main components required in any scalable crawler, and it has discussed design alternatives for those components.

In particular, the paper described Mercator, an extensible, scalable crawler written entirely in Java. Mercator's design features a crawler core for handling the main crawling tasks, and extensibility through protocol and processing modules. Users may supply new modules for performing customized crawling tasks. We have used Mercator for a variety of purposes, including performing random walks on the Web, crawling our corporate intranet, and collecting statistics about the Web at large.

Although our use of Java as an implementation language was somewhat controversial when we began the project, we have not regretted the choice. Java's combination of features – including threads, garbage collection, objects, and exceptions – made our implementation easier and more elegant. Moreover, when run under a high-quality Java runtime, Mercator's performance compares well to other Web crawlers for which performance numbers have been published.

Mercator's scalability design has worked well. It is easy to configure the crawler for varying memory footprints. For example, we have run it on machines with memory sizes ranging from 128 MB to 2 GB. The ability to configure Mercator for a wide variety of hardware platforms makes it possible to select the most cost-effective platform for any given crawling task.

Mercator's extensibility features have also been successful. As mentioned above, we have been able to adapt Mercator to a variety of crawling tasks, and as stated earlier, the new code was typically quite small (tens to hundreds of lines). Java's dynamic class loading support meshes well with Mercator's extensibility requirements, and its object-oriented nature makes it easy to write variations of existing modules using subclassing. Writing new modules is also simplified by providing a range of general-purpose reusable classes, such as classes for recording histograms and other statistics.

Mercator is scheduled to be included in the next version of the *AltaVista Search Intranet* product [AltaVista], a version of the AltaVista software sold mostly to corporate clients who use it to crawl and index their intranets.

Acknowledgements

Thanks to Sanjay Ghemawat for providing the high-performance Java runtime we use for our crawls.

References

AltaVista, "AltaVista Software Search Intranet Home Page," `altavista.software.digital.com/search/intranet`.

BIND, "Berkeley Internet Name Domain (BIND)," `www.isc.org/bind.html`.

Bloom, B. (1970), "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Communications of the ACM 13*, 7, 422–426.

Brin, S. and L. Page (1998), "The Anatomy of a Large-Scale Hypertextual Web Search Engine," In *Proceedings of the Seventh International World Wide Web Conference*, pp. 107–117.

Broder, A. (1993), "Some Applications of Rabin's Fingerprinting Method," In *Sequences II: Methods in Communications, Security, and Computer Science*, R. Capocelli, A. De Santis, and U. Vaccaro, Eds., Springer-Verlag, pp. 143–152.

Burner, M. (1977), "Crawling Towards Eternity: Building an Archive of the World Wide Web," *Web Techniques Magazine 2*, 5.

Cho, J., H. Garcia-Molina, and L. Page (1998), "Efficient Crawling Through URL Ordering," In *Proceedings of the Seventh International World Wide Web Conference*, pp. 161–172.

DCPI, "Digital Continuous Profiling Infrastructure," `www.research.digital.com/SRC/dcpi/`.

Eichmann, D. (1994), "The RBSE Spider – Balancing Effective Search Against Web Load," In *Proceedings of the First International World Wide Web Conference*, pp. 113–120.

Ghemawat, S., "srcjava home page," `www.research.digital.com/SRC/java/`.

Google, "Google! Search Engine," `google.stanford.edu/`.

Gray, M., "Internet Growth and Statistics: Credits and Background," `www.mit.edu/people/mkgray/net/background.html`.

Henzinger, M., A. Heydon, M. Mitzenmacher, and M.A. Najork (1999), "Measuring Index Quality Using Random Walks on the Web," In *Proceedings of the Eighth International World Wide Web Conference*, pp. 213–225.

Heydon, A. and M. Najork (1999), "Performance Limitations of the Java Core Libraries," In *Proceedings of the 1999 ACM Java Grande Conference*, pp. 35–41.

InternetArchive, "The Internet Archive," `www.archive.org/`.

Koster, M., "The Web Robots Pages," `info.webcrawler.com/mak/projects/robots/robots.html`.

McBryan, O.A. (1994), "GENVL and WWWW: Tools for Taming the Web," In *Proceedings of the First International World Wide Web Conference*, pp. 79–90.

Miller, R.C. and K. Bharat (1998), "SPHINX: A Framework for Creating Personal, Site-Specific Web Crawlers," In *Proceedings of the Seventh International World Wide Web Conference*, pp. 119–130.

Pinkerton, B. (1994), "Finding What People Want: Experiences with the WebCrawler," In *Proceedings of the Second International World Wide Web Conference*.

Rabin, M.O. (1981), "Fingerprinting by Random Polynomials," Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University.

RobotsExclusion, "The Robots Exclusion Protocol," `info.webcrawler.com/mak/projects/robots/exclusion.html`.

Smith, Z. (1997), "The Truth About the Web: Crawling Towards Eternity," *Web Techniques Magazine 2*, 5.