Mercator as a web crawler

Priyanka-Saxena[1]
[1] **Department of Computer Science Engineering, Shobhit University,
Meerut, Uttar Pradesh-250001, India**

Abstract

The Mercator describes, as a scalable, extensible web crawler written entirely in Java. In term of Scalable, web crawlers must be scalable and it is important component of many web services, but their design is not well-documented in the literature. In this paper, 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 more efficient and comparable to that of other crawlers.

Keywords: *Introduction, Related Work, Architecture, Components, Extensibility, Conclusion.*

1. Introduction

Designing a scalable web crawler comparable to general crawler used by the major search engines is a complex endeavor. Due to the competitive nature of the search engine business, there are few papers in the literature form for describing the challenges and tradeoffs of inherent web crawler design. This paper describes, Mercator as a scalable, extensible web crawler written entirely in Java for filling the gap between the challenges comparable to that of other crawlers.

By *scalable is define as*, Mercator is designed to scale up to the entire web, and used for fetching millions (tens of) of web documents. We achieve scalability by implementing our data structures with bounded amount of memory and regardless for the size of the crawl. Vast majority of our data structures are stored on disk, and small parts of them are stored in memory for efficiency.

By *extensible is define as*, 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 collect a variety of statistics about the web, and to perform a series of random walks of the web.

The initial motivations is to collect statistics about the web 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* exact priority for which statistics are interesting, and the topics of interest may change overtime. Mercator makes it easy to collect new statistics—to configure for different crawling tasks—by allowing users to provide their own modules for processing downloaded documents.

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.

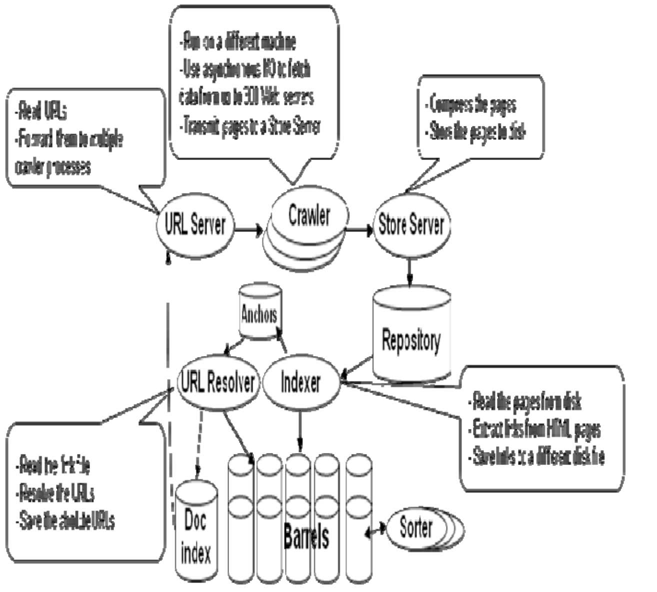
2. Related work

Web crawlers— are almost as old as the web itself. Several papers about web crawling were
presented at the first two World Wide Web conferences. However, time to time, the web was considered challenges like two to three orders of magnitude smaller. Today, those systems did not address the scaling problems inherent in a web crawler. Obviously, all of the popular search engines use crawlers that must scale up to substantial portions of the web such as two notable exceptions: the Google crawler and the Internet Archive crawler.

The Google search engine is a distributed system that uses multiple machines for crawling. 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.


-Run on a different machine
-Use asynchronous I/O to fetch data from up to 300 Web servers
-Transmit pages to a Store Server
-Read URLs
-Forward them to multiple crawler processes
-Compress the pages
-Store the pages to disk
URL Server
Crawler
Store Server
Anchors
Repository
URL Resolver
Indexer
-Read the pages from disk
-Extract links from HTML pages
-Save links to a different disk file
-Read the link file
-Resolve the URLs
-Save the absolute URLs
Doc index
Barrels
Sorter

The Internet Archive also uses multiple machines to crawl the web and each crawler process is assigned up to 64 sites to crawl, but 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, SPHINX system provides some of the same customizability features as Mercator. 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 a input and repeatedly execute the following steps. Remove a URL from the URL list; and determine the IP address of its Host Name, download the corresponding document, and extract any links contained in document. For each of the extracted links, ensure that it is an absolute URL, and add URL to the list of URLs to download, provided it has not been encountered before.

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;

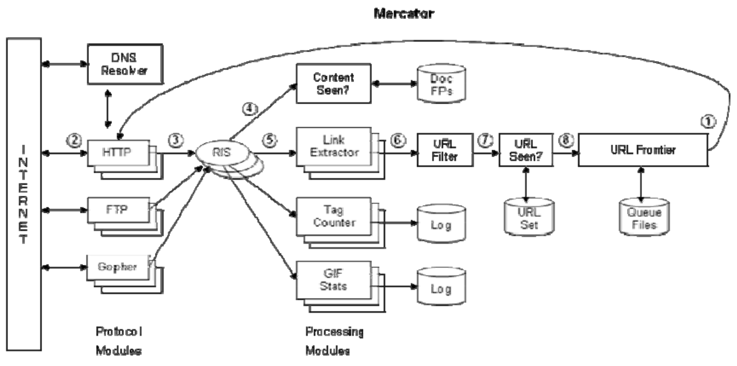


Figure 1. Mercator's main components

Fig. 1. Mercator's Main components

4. Mercator's components

Mercator's main components are described in figure 1. The Web Crawling is performed by multiple worker threads and each worker repeatedly performs the steps needed to download and process a document. In the above figure 1, Mercator's components working shown in the form of steps are as follows:-

(1) The first step is to remove an absolute URL from the shared URL frontier for downloading.

An absolute URL begins with a *scheme* (e.g., "http"), which identifies the network protocol that are implemented by *protocol modules*. The protocol modules is to be used in a crawl are specified in a user-supplied configuration file which is dynamically laded at the start of the crawl, and there is a separate instance of each protocol module per thread, which allows each thread to access local data without any synchronization.

(2) The second step is based on the URL's scheme, the worker selects the appropriate protocol module for downloading the document and the protocol module's *fetch* method, which downloads the document from the Internet.
(3) The third step is that the downloaded document from the internet after fetching by protocol module is write into a per-thread *RewindInputStream* (RIS) . A RIS is an I/O abstraction that is initialized from an arbitrary input stream, and that subsequently al-

lows that stream's contents to be re-read multiple times.(ADVANTAGE OF MERCATOR)

(4) After successful completion of step third then the forth step is the worker thread invokes the *content-seen test* to determine whether this document (associated with a different URL) has been seen before . If 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 and a Mercator configuration file also associates MIME types with one or more *processing modules* 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.

(5) Based on the downloaded document's MIME type, the fifth step of the worker invokes the *process* method of each processing module associated with that MIME type.

Note:-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.

(6) The *process* method of the processing module extracts all links from an HTML page and each link is converted into an absolute URL, and the sixth step is tested against a user-supplied *URL filter* to determine if it should be downloaded.(ADVANTAGE OF MERCATOR)

(7) If the URL passes from the URL filter, the seventh step is the worker performs the *URL-seen test* , which checks if the URL has been seen before.

(8) if it is in the URL frontier or has already been downloaded. If the URL is new, it is added to the frontier.

4.1 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 and this type of 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 matters are complicated by the multiple HTTP requests pending to the same server. If multiple requests are in parallel, the queue's *remove* operation should not simply return the head of the queue, but rather than the 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 implemented by a collection of distinct FIFO sub queues. First, there is one FIFO sub queue per worker thread. Second, when a new URL is added, the FIFO sub queue in which it is placed is determined by the URL's canonical host name. This design prevents Mercator from overloading a web server, also handle a bottleneck of the crawl.(ADVANTAGE OF MERCATOR).

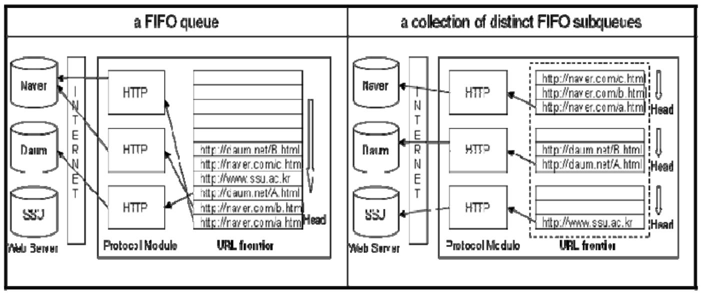


Fig. 2. Data Structure of URL Frontier

4.2 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 which is supported by Mercator include HTTP, FTP, and Gopher.

The Mercator implement the Robots Exclusion Protocol, which allows web masters to declare the wed crawler to fetch a special document containing these declarations from a web site before downloading any real content from it. To avoid downloading the Robot Exclusion File(Robot.txt) file on every request, Mercator's HTTP protocol module maintains a fixed-sized cache mapping host names to their robots exclusion rules and due to this by default, the cache is limited to 218 entries, and uses an LRU replacement strategy.

Mercator uses its own "lean and mean" HTTP protocol module; its requests time out after 1 minute, and it has minimal synchronization and allocation overhead.

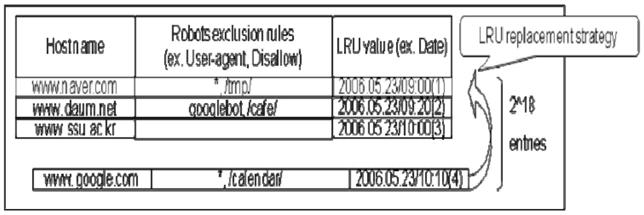


Fig. 3. The HTTP Protocol Module

4.3 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, Mercator 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(limit 1 MB). 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.

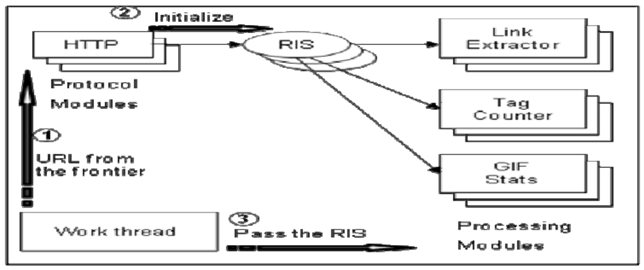


Fig. 4. The Rewind Input Stream

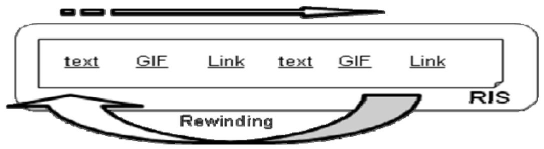
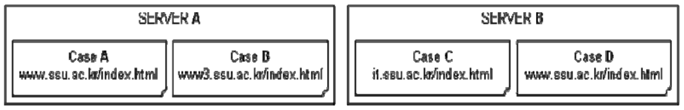


Fig. 5. The Rewind Process

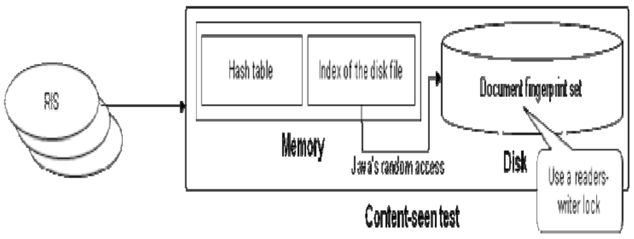
4.4 Content seen test

In web, many documents are available under multiple times with different URLs. There are also many cases in which documents are mirrored on multiple servers show in Fig. 6.

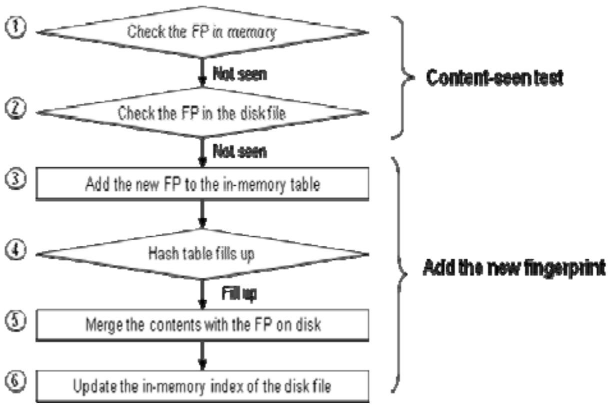

SERVER A
Case A www.ssu.ac.kr/index.html
Case B www3.ssu.ac.kr/index.html
SERVER B
Case C it.ssu.ac.kr/index.html
Case D www.ssu.ac.kr/index.html

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. Mercator using a content-seen test makes it possible to suppress link extraction from mirrored pages, 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.
To save space and time, Mercator uses data structure called the *document fingerprint set* that stores a 64-bit checksum of the contents of each downloaded document and also compute the checksum using fingerprinting algorithm. Fingerprints offer provably strong probabilistic guarantees that two different strings will not have the same fingerprint.

Mercator maintains two independent sets of fingerprints: a small hash table kept in memory, and a large sorted list kept in a single disk file. Fig. 7. Content-Seen Test

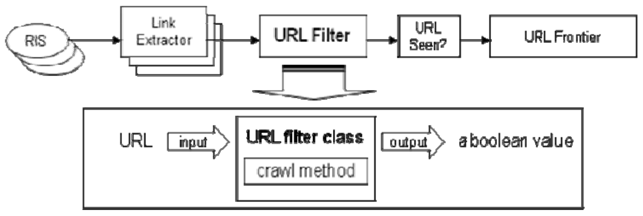


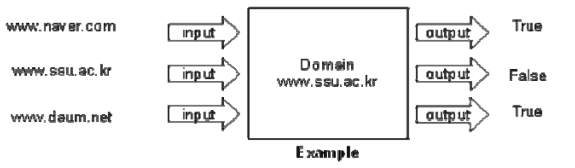
The content-seen test first checks if the fingerprint is contained in the in-memory table. If not, it has to check if the fingerprint resides in the disk file. To avoid multiple disk seeks and reads per disk search.
Mercator performs an interpolated binary search of an in-memory index of the disk file 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. Fig. 8. Steps of Content-Seen Test


① Check the FP in memory
Not seen
② Check the FP in the disk file
Not seen
Content-seen test
③ Add the new FP to the in-memory table
④ Hash table fills up
Fill up
⑤ Merge the contents with the FP on disk
⑥ Update the in-memory index of the disk file
Add the new fingerprint

4.5 The url filtering

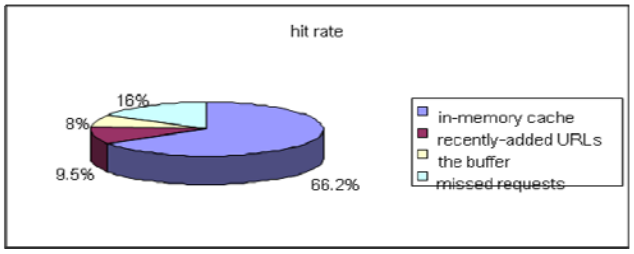
The URL filtering mechanism provides a customizable way to control the set of URLs that are downloaded.
The URL filter class has a single crawl method that takes a URL and returns a boolean value indicating whether or not to crawl that URL. Mercator includes a collection of different URL filter subclasses that provide facilities for restricting URLs by domain, prefix, or protocol type, and for computing the conjunction, disjunction, or negation of other filters. Fig. 9. The URL Filter with example



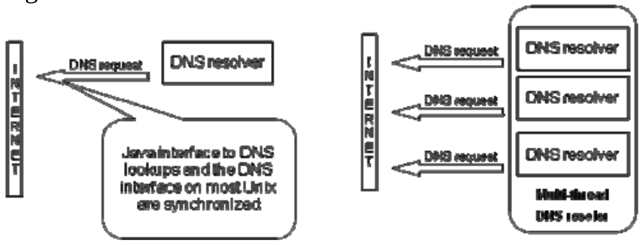


4.6 Domain name service

Before using 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 to avoid bottleneck of most web crawlers.



Mercator tried to alleviate the DNS bottleneck by caching DNS results, but that was only partially effective and also used its own multi-threaded DNS Resolver can resolve host names much more rapidly than either the Java or Unix resolvers. This meant that only one DNS request on an un cached name could be outstanding at once. The cache miss rate is high enough that this limitation causes a bottleneck. Fig. 10. Domain Name Resolution



Perform DNS looking accounted for 87% of each thread elapsed time and reduce that elapsed time to 25%

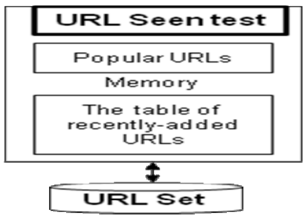
4.7 The url seen test

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 to perform the URL-seen test. All of the URLs seen by Mercator in canonical form in a large table called the *URL set*. To save space, Mercator does not store the textual representation of each URL in the URL set, but rather a fixed sized checksum. To reduce the number of operations on the backing disk file, Mercator therefore keep an in memory cache of popular URLs.

Unlike the fingerprints, the stream of URLs has a non-trivial amount of locality (URL locality). So, each URL set membership test induces one-sixth as many kernel calls as a membership test on the document fingerprint set. Host name locality arises because many links found in web pages are to different documents on the same server. To preserve the locality, we compute the checksum of a URL by merging two independent fingerprints:

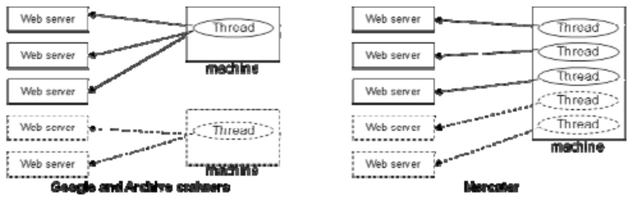
1. The fingerprint of the URL's host name
2. The fingerprint of the complete URL

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, this technique results in a significant reduction in disk load, and hence, in a significant performance improvement. Figure 11**.**Using an in-memory cache of 2^18 entries and the LRU-like clock replacement policy and Fig**.** 11. URL SEEN TEST



4.8 Synchronous and asynchronous i/o

Both Google and Internet Archive crawlers use single-threaded crawling processes and asynchronous I/O to perform multiple download in parallel and they are designed from the ground up to scale to multiple machines. Whereas, Mercator uses a multi-threaded process in which each thread performs synchronous I/O (It leads to a much simpler program structure **is** the main advantage of Mercator) and it would not be too difficult to adapt Mercator to run on multiple machines. One strength of the Google and the Internet Archive crawlers is that they are designed from the ground up to scale to multiple machines. Fig.12. Synchronous vs. asynchronous I/O



4.9 Checkpointing

To complete a crawl of the entire web, Mercator writes regular snapshots of its state to disk. An interrupted or aborted crawl can easily be restarted from the latest checkpoint. Mercator's core classes and all user-supplied modules are required to implement the check pointing interface. User-supplied protocol or processing modules are also required to implement the check pointing 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. Once a day, Mercator's main thread acquires the write lock, so it is guaranteed to be running in isolation and also it acquired the lock, the main thread arranges for the checkpoint methods to be called on Mercator's core.

5. Extensibility

Mercator is an extensible crawler it means two things.

1. First, Mercator can be extended with new functionality.
2. Second, Mercator can easily be reconfigured to use different versions of most of its major components.

Different versions of the URL frontier, document fingerprint set, URL filter, and URL set may be all are dynamically "plugged into" the crawler means multiple versions of each of these components, which we employ for different crawling tasks.
Making an extensible system, Mercator requires three ingredients:

1. The interface to each of the system's components must be well-specified is defined by an *abstract* class.
2. A mechanism must exist for specifying how the crawler is to be configured from its various components by supplying a *configuration file* which 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.
3. Sufficient infrastructure must exist to make it easy to write new components such as rich set of utility libraries with a set of existing pluggable components.

To demonstrate Mercator's extensibility, here we describe some of the extensions.

5.1 Protocol and processing module

By default, Mercator will crawl the web by fetching documents using the HTTP protocol, extracting links from documents of type text/html. 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 two methods.

1. The fetch method downloads the document corresponding to a given URL, and
2. the new URL method parses a given string, returning a structured URL object.

The abstract Analyzer class is the superclass for all processing modules and defines a single process method which is responsible for reading the document and processing it appropriately. Analyzers often keep private state or write data to the disk.
Other processing modules simply write the contents of each downloaded document to disk. As another experiment, 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.

5.2 Alternative url frontier and implementation

We described one implementation of the URL frontier data structure and that implementation on a crawl of our corporate intranet, with the drawback is that multiple hosts might be assigned to the same worker thread, while other threads were left idle. This situation is occur on an intranet because intranets typically contain a substantially smaller number of hosts than the internet at large.
To restore the parallelism, an alternative URL frontier component that dynamically assigns hosts to worker threads and that at most one worker thread will download documents from any given web server at once. It also maximizes the number of busy worker threads within the limits and 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.

5.3 Configuring mercator as random walker

Mercator used to perform random walks of the web in order to gather a sample of web to measure the quality of search engines. 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.
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.

6. 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.
The paper described Mercator; an extensible, scalable crawler is design with the 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. Mercator's scalability design has worked well. It is easy to configure the crawler for varying memory footprints. 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 and able to adapt Mercator to a variety of crawling tasks. Mercator is scheduled to be included in the next version of the *AltaVista Search Intranet* product is mostly to corporate clients who use it to crawl and index their intranets.

7. References

[1] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the Seventh International World Wide Web Conference*, April 1998

[2] Google! Search Engine. http://google.stanford.edu/.

[3] Monika Henzinger, Allan Heydon, Michael Mitzenmacher, and Marc A. Najork. Measuring Index Quality using Random Walks on the Web. In *Proceedings of the Eighth International World Wide Web Conference*, May 1999.

[4] The Internet Archive. http://www.archive.org/,

[5] Z. Smith. The Truth About the Web: Crawling towards Eternity. *Web Techniques Magazine*, 2(5), May 1997.

[6] The Web Robots Pages. http://info.webcrawler.com/mak/projects/robots/robots.html.

[7] The Robots Exclusion Protocol. http://info.webcrawler.com/mak/projects/robots/exclusion.html,

[8] M. O. Rabin. Fingerprinting by Random Polynomials. Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[9] Oliver A. McBryan. GENVL and WWWW: Tools for Taming the Web. In *Proceedings of the First International World Wide Web Conference*, pages 79–90, 1994.

[10] Robert C. Miller and Krishna Bharat. SPHINX: A framework for creating personal, site-specific Web crawlers. In *Proceedings of the Seventh International World Wide Web Conference*, April 1998.

First Author- I am Priyanka Saxena, presuing M.Tech in Computer Science Engineering from Shobhit University, Meerut Uttar Pradesh, INDIA and got 8 cgpa in every semester. My Thesis is left, which is hopefully completed up to August 2012. I have completed my B.Tech in Computer Science from Uttar Pradesh Technical University in 2009 with 72%. I have one year gap between my studies in which I was complete my Hardware Support Training form HP Delhi with 84% and A+ grade. It is my first paper on Web Crawler and I will find the details of IJCSI through Internet. I am heartily thank full to IJCSI for giving me this big opportunity to show my skills and knowledge which definitely generate positive energy for my next paper.