DESIGN AND IMPLEMENTATION OF A PARALLEL CRAWLER

By

Ankur M. Patwa

A project submitted to the graduate faculty of

The University of Colorado at Colorado Springs in partial

fulfillment of the Master of Science degree

Department of Computer Science

2006

This project for Master's of Science degree by

Ankur M. Patwa has been approved for

the Department of Computer Science

by

_____

Dr. J. Kalita (Advisor)

_____

Dr. C. E. Chow (Committee Member)

_____

Dr. S. Semwal (Committee Member)

_____

Date

Table of Contents

List of Tables:

List of Figures:

Abstract:

A Web crawler is a module of a search engine that fetches data from various servers. It can be implemented as an independent module or in coalition with other modules. This module demands much processing power and network consumption. It is a time-taking process to gather data from various sources around the world. Such a single process faces limitations on the processing power of a single machine and one network connection. If the workload of crawling Web pages is distributed amongst several machines, the job can be performed faster.

The goal for the project is to discover problems and issues pertaining to the design and development of a scalable, parallel architecture for crawling a set of domains; to enumerate important problems that arise and provide solutions for the same. The crawled pages can be used as input for an indexer module and/or an analysis module. The task of distribution of domains to crawling machines and a strategy to crawl the domains without overloading the servers are experimented with and concerns important to such tasks are recorded.

Chapter 1: Introduction

With the ever expanding Internet, it is difficult to keep track of information added by new sites and new pages being uploaded or changed everyday. While the Internet is nearing chaos, it is difficult for a user to find correct information in a timely manner. Today's search engines are greatly used to get whereabouts of relevant information ver y quickly. They are like maps and signs which point the user in right direction.

A search engine consists of following modules:

- A crawling module which fetches pages from Web servers
- Indexing and analysis modules which extract information from the fetched pages and organize the information
- A front-end user interface and a supporting querying engine which queries the database and presents the results of searches.

Web crawlers are a part of the search engines that fetch pages from the Web and extract information. The downloaded pages are indexed according to the amount and quality of informational content by an indexing module and the results from the storage is provided to a user via a user interface. The crawlers can also be used to check links of a site, harvest email addresses and other tasks. These days, one can witness mobile crawlers, crawling agents, collaborative agents, crawlers implemented on a point-to-point network and others.

A crawling module consists of the following functional parts:

- URL frontier: It is a list of URLs to be crawled by the crawler.
- A page-fetching function: This is implemented as a browser object which queries a Web server and downloads the document at a given URL.
- Link extracting module: Downloaded documents are parsed for links and links which have not been encountered before are extracted and pushed into the URL frontier. Relative links are converted to absolute or canonical URLs. The functionality of extracting URLs can be enhanced by adding parsers to parse different types of documents. Some of the types of documents other than HTML documents that can contain links are CSS and Javascript files, PDF and Word documents etc.
- Storing module: This module stores the fetched document. A physical disk is used as the medium to store the pages.
- A database to store information about the crawling task and meta-information about the pages that are crawled.

The modules listed above can be implemented simultaneously or executed one after another. The information gathering and extracting processes can be implemented in a distributed fashion and the results can be put into one place to be processed by another module. A parallel crawler is implemented in a distributed fashion to crawl educational domains ending in *.edu.*

Information gathered by the experiments will be used for the Needle project which is an outcome of Zhang's work [YDI 2006] and Patankar's project [SNC 2006].

1.1 Problem Statement:

Design and implement a centrally managed architecture of parallel crawlers to crawl US educational sites (.edu). Perform site-based distribution of work among the machines and simultaneously crawl as many domains as possible.

Some of the problems faced were evident during research and the goal of the project was to device solutions for the problems. Following is a list of problems known beforehand:

- Duplication of data amongst the machines.
- Presence of skew in terms of work load amongst the crawlers.
- Lack of an efficient communication system amongst the machines with a focus on decreasing number of URLs exchanged amongst the machines.
- Fast and economical restarting mechanism for the machines in the system and a low time to allow for resilience of the system.

More information about crawlers and information discovered are discussed in the second chapter.

1.2 Objectives and Design Goals

The following are the aspects to be considered for the design of a crawler:

- Politeness: The crawler should be polite to Web servers in a way that it does not over burden the Web servers with frequent requests in a short amount of time. There should be enough delay between consecutive requests to the same server.
- Rule-abiding: The robot rules in robot exclusion files (*robots.txt*) on the servers should be followed without any exceptions. The crawler should not fetch documents which it (or any other spider) is not allowed to access.
- Scalability: Performance of the crawler should not degrade if there is an increase in the number of domains to be crawled or an increase in machines assigned to the task of crawling.
- Easy but exhaustive configuration: The configuration of a crawler should be easy and intuitive. Almost all the aspects of the crawler should be configurable but the amount of options should not be daunting. There should not be any two mutually exclusive options.
- Speed: The crawler should be able to gain a good speed for crawling URLs and should be able to maintain that speed even if the number of URLS increase or there is an increase in the number of domains to be crawled.
- Priority based frontier: Frontier is the part of a crawler which holds the URLS to be crawled. The crawler should make a sound choice regarding which URL should be crawled next. The URLs will have a weight or a rank, based on which URL will be chosen next. The URL with the highest rank will be crawled before any other URLs. Logic defines how the rank of a URL is determined. Any URL which is encountered in other URLs but not yet crawled gets a higher rank and will be shifted nearer the front of the frontier.

- Detection of non-functional servers: A good crawler should detect that a Web server is offline and is not processing requests or takes a long time to process the requests.
- Selective crawling of documents: The crawler should crawl documents where there is a possibility to find links. i.e. documents having content-type *text/html*, *text/javascript* and others and should have logic to avoid the download of audio/ video files or files which might not contain links.
- Load Balancing: The task of crawling should be equally distributed among the machines based on their performance.

1.3 Summary of Work:

The crawling system was built from scratch. Design logic was developed and implementation of the system was changed a many times as problems became evident. The ideal properties of a crawler, as mentioned above, were kept in focus for building the system. When the system was updated, crawled pages were not discarded. Experiments were performed starting with four machines. Issues like caching of Robot Rules, distribution of sites amongst crawlers, a quick and economical restart mechanism and others were solved. One main problem faced was the processing speed of the machines. These and other issues are mentioned in detail in the fourth chapter.

Chapter 2: Background Research

2.1 Related work:

The distribution of a Web crawling task not only faces the yard-stick of getting extracted information but also have to handle other issues such as network usage, distribution of sites to be crawled and synchronization of downloaded data with previous data, duplication of data and merging of results.

In [CGM 2002], the authors discuss various criteria and options for parallelizing the crawling process. A crawler can either be centrally managed or totally distributed. A crawler can be designed as to ignore overlap of pages that are downloaded while taking care of network load or vice versa. The authors define the *quality* of a crawler as its ability to download "important" pages before others. For a parallel crawler, this metric is important as every crawling process focuses only on local data for marking pages as important. The authors mention that distributed crawlers are advantageous than multi-threaded crawlers or standalone crawlers on the counts of scalability, efficiency and throughput. If network dispersion and network load reduction are done, parallel crawlers can yield good results. Their system utilizes memory of the machines and there is no disk access. They did not store their data to prove the usefulness of a parallel crawling system. Their target was a small set of news sites.

Mercator is a scalable and extensible crawler, now rolled into the Altavista search engine. The authors of [HNM 1999] discuss implementation issues to be acknowledged for developing a parallel crawler like traps and bottlenecks, which can deteriorate performance. They discuss pros and cons of different coordination modes and evaluation criteria. The authors summarize their work with very good performance statistics. In brief, they concur that the communication overhead does not increase linearly as more crawlers are added, throughput of the system increases linearly as more nodes are added and the *quality* of the system, i.e. the ability to get "important" pages first, does not decrease with increase in the number of crawler processes. Fetterly and others in their work [FMN 2003] accumulated their data using the Mercator crawler. They crawled 151 million HTML pages totaling 6.4 TB over a period of 9 days.

Shkapenyuk and Suel of [SSD 2002] produce statistics similar to the ones produced by Heydon and Najork of [HNM 1999]. Their architecture employs one machine for each modular task. To avoid bottlenecks while scaleup, they had to deploy more machines for a specific task. Therefore, the number of machines increased with a greater curve for adding more crawler machines. The network communication load increased drastically due to increased coordination work amongst machines with similar tasks.

The architecture of Ubicrawler as described in [BCS 2002] is fully distributed in terms of agents. A single point of failure does not exist but at an expense of heavy intra-system communication and duplication of data. Their system is a fault-tolerant, platform

independent and all data is saved to files. The agents push URLs amongst themselves in order to download them. If a node is busy, another node is contacted. Much latency can creep into the system if a free node is hard to find at correct time. This utilizes more resources for coordination amongst agents than necessary. Manual tuning of allocation of nodes to agents is needed to avoid bottlenecks.

In [LKC 2004], the authors implement a distributed system of crawlers in a point-to-point network environment. Data to be communicated to and fro amongst systems are stored in dynamic hash tables (DHTs). The crawler is an on-the-fly crawler. It fetches pages guided by a user's search. Even when the search triggers the crawling task, there is no pruning of URLs and exclusion pages based on content. This can result in poor quality of results returned. They implement URL-redirection amongst nodes to balance the load but the criteria for redirection of URLs is unclear. The usage of memory due to DHTs and the coordination is not recorded. But they claim to reach download speed of around 1100 kBps per second with 80 nodes.

Authors of [PSM 2004] have implemented a location-aware system where globally distributed crawlers fetch pages from servers which are nearest to them. The system is resilient to failures but it has long resilience time and the feature is detrimental to the organization of data. When a crawler goes down, the domains it was crawling are transferred to other crawlers. This results in a large overlap of data. A heavy heartbeat protocol is needed to choose the nearest crawler. Moreover, a crawler nearer to many

Web servers might be overloaded whereas a server at a little bit more distance may be sitting idle.

Cho et. al. in their work [JHL 1998], describe the importance of URL re-ordering in the frontier. If a page exists in the frontier and it is linked in by many pages that are crawled, it makes sense to visit it before others which are linked from a few number of pages. PageRank is used as a driving metric for ordering of URLs and three models of crawlers are evaluated. They conclude that Page Rank is a good metric and pages with many backlinks or ones with a high PageRank are sought first.

In [JHE 2000], Cho and Molina evaluate an incremental crawler. A collection is the repository where crawled pages are stored. They describe a periodic crawler as a crawler which re-visits the sites only after crawling all the sites in an iteration. On the other hand, an incremental crawler incrementally crawls and re-visits pages after a target amount of pages are crawled and stored in the collection. This target is specified by a *page window*. A page window is the number of pages when crawled; the crawler has to crawl them again. This way, an incremental crawler indexes a new page as soon as it is found as opposed to the periodic crawler which indexes new pages only after the current crawling cycle is complete. The way an incremental crawler crawls is called *active crawling*, where the crawler is aggressive on getting more recent copies of pages. A periodic crawler's crawling process is called *passive crawling*. The researchers crawl a small amount of the Web and discover that more than 70% of the pages over all domains remained in their page window for more than one month and it took only 11 days for

50% of the .com domain to change, while the same amount of change took almost 4 months for the .gov domain. The authors state that the rate of change of pages is best described by a Poisson process. A Poisson process is often used to model a sequence of independent and random events. The authors describe two techniques to maintain the collection. With the first one, a collection can have multiple copies of pages grouped according to the crawl in which they were found. For the second one, only the latest copy of pages is to be saved. For this, one has to maintain records of when the page changed and how frequently it was changed. This technique is more efficient than the previous one but it requires an indexing module to be run with the crawling module. The authors conclude that an incremental crawler can bring fresher copies of pages more quickly and keep the repository more recent than a periodic crawler. However, they conduct their experiment for a limited period of time.

A recent paper, [JHH 2006] was discovered late during my research but it is worth mentioning here as the architecture of my system and their system have a lot in common. In their work, the authors describe the Stanford's WebBase system which is the academic version of Google! repository and search engine before it was commercialized in late 1990's. The component of interest is the crawler module. They implement 500 processes dedicated to the task of crawling. The process of crawling is parallelized amongst the machines. A site with a fully qualified domain name is taken as an atomic unit for the crawling process and it should not be crawled by more than one machine for a crawling cycle. The authors state good points on why a site should be treated as independent components. For example, it makes management of crawlers much easier

requiring little coordination amongst crawlers and the URL data structures can be stored in the main memory. One can reinforce site-specific crawling policies too; like scheduling the time of the crawl. But they crawl all the sites at the same time (9 A.M. to 6 P.M.PST). This can be improved upon where the location of servers is taken into consideration. One can crawl the sites at night time according to the location of servers or during the wee hours of mornings when the servers expect lesser load. During the day, one can crawl in a non-intrusive way with a large delay between consecutive fetches. The authors refer to this delay as *courtesy pause* and it ranges from 20 seconds for smaller websites to 5 seconds for larger websites. The courtesy pause and courtesy policies like respecting the robots exclusion protocol play a major role in the design of a system. The system has a threshold of 3000 pages to be crawled per site, on a day. The crawlers crawl the pages which belong to one domain at a time. There are two types of crawling techniques described. One, there is a process dedicated to crawl each site and a process which crawls multiple sites simultaneously. The authors discuss advantages and trade-offs between two processes. Their system implements a combination of both types of processes. The system translates domain names to IP addresses for the *seed URLs*. IP addresses of sites are used to fetch more pages from sites. The authors state that if one does not resolve domain names, one could experience a performance hit and accuracy loss due to high DNS queries. But the technique fails if there are multiple virtual servers implemented by hosting services or if a domain exchanges service providers. Additionally, robots.txt files on the sites hosted on virtual servers cannot be retrieved. This and other factors make the choice of resolving a domain name during the starting of

a crawl more expensive and difficult. For the purpose of crawling, the authors conclude that treating a site as one unit elevates parallelization of a crawling process.

Using a single process for the task of crawling in her work [SNC 2006], Patankar crawled 1 Million pages in a period of nine days. Data was loaded into the main memory from the tables and a number of queues were used to represent the *frontier*. HTML pages and images were crawled from five *.edu* domains.

2.2 Politeness and Robots Exclusion Protocol

There are two criteria for a Web crawler to fulfill to be a polite crawler:

1. Wait for some amount of time before fetching another page from one server and not having many simultaneous connections with the same server at any given point of time in order to avoid a high load for the servers it is crawling.
2. Respect the rules in the robots exclusion file: robots.txt hosted on the server's root directory.

A detailed explanation of both the points is as follows:

Crawlers automate the process of requesting data from the servers. The automated scripts can perform the job very quickly, resulting in a crippling impact on the performance of a Web server. If a single crawler is performing multiple requests per second and/or downloading large files, and there are multiple crawlers trying to crawl sites, there will be a daunting load on a server due to requests from multiple crawlers.

Even if Web crawlers can be used for a number of tasks as noted by Koster in [KMR 1995], but they come with a price for the general community. Web server performance can be hit due to answering crawlers and the ways the performance is affected are:

- Network resources are consumed greatly, as crawlers require considerable bandwidth to operate and they generally function with a high degree of parallelism for a long time span.
- The number of requests coming in is high and at a high frequency resulting in steep server overload.
- If a crawler is poorly written or the inherent logic is inefficient, it can crash servers or routers by sending too many requests in a small period of time, or download a huge number of pages that they cannot handle.
- If too many users stated using personal crawlers and deployed them around the same time, they can disrupt networks and Web servers.

The first proposal for the interval between two requests was given in [KGR 1993] and was proposed to be a 60 seconds value. However, if pages were downloaded at such a rate from a large website having more than 100,000 pages over a perfect, zero latency connection with infinite bandwidth, it would take a crawler more than 2 months to crawl only that entire website; also, the Web server would be sitting idle most of the time. In addition, the crawler would never be able to get current pages with latest information from that Web server. This norm does not seem practical, let alone be acceptable.

Cho and Molina in their work [JGM 2003] use 10 seconds as an interval between two page accesses, and the implementers of the WIRE crawler [BYC- 2002] use 15 seconds as the default time gap between fetching two pages. An adaptive politeness policy was followed by the Mercator Web crawler [HNM 1999]: if downloading a document from a given Web server takes t seconds, the crawler should wait at least for (10 x t) seconds before downloading the next page from that particular Web server. Dill et al. the their work [DKR 2002] used just 1 second but that seems to be too short of an interval because generally, a default connection time-out for a Web server is 30 seconds.

Anecdotal evidence from Web servers' access logs show that access intervals from known Web crawlers range anything from 20 seconds to 3–4 minutes. It is worth noticing that even when being very polite, and taking all the safeguards to avoid overloading Web servers, some complaints from Web server administrators are received.

The following is a classic example on why a crawler needs to be polite while crawling pages. Grub, a search engine acquired by LookSmart was based on a distributed computing algorithm. According to grub.looksmart.com, the Grub crawling project is not operational right now. Quite a few webmasters were opposed to being crawled by a Grub crawler for its apparent ignorance of sites' robots.txt files. Because Grub cached robots.txt, any changes to the file could not be detected. Webmasters encountered that

Grub did not understand newly created robots.txt files blocking access to specific areas of their site for all crawlers. Grub's distributed architecture resulted in a huge amount of server overload by keeping open a large number of TCP connections — the effects of this were essentially similar to a typical distributed denial of service (DDOS) attack.

Koster in [KRE 1996] proposed a partial solution to problems stated above called initially as the robots exclusion protocol, also known as the robots.txt protocol and it is now a way to communicate by administrators to the crawlers about what parts of their Web servers should not be accessed. Even though an interval between requests plays an important role on the Web server's load, its efficiency and its availability, this standard does not include a suggestion for the interval of visits to the same server. A non-standard robots.txt file can use a "Crawl-delay:" parameter to indicate the number of seconds to pause between two requests, and some commercial search engines like Google, MSN, Yahoo!, are already considering and respecting this norm.

2.2.1 A simple robots file

Even if you want all your directories to be accessed by spiders, a simple robots file with the following may be useful:

```
User-agent: *
Disallow:
```

With no file or directory listed in the Disallow line, you imply that every directory on your site can be accessed by any crawler. At the very least, this file will save you a few bytes of bandwidth each time a spider visits your site (or more if your 404 file is large); and it will also remove robots.txt from your web statistics bad referral links report.

```
User-agent: *
Disallow: /cgi-bin/
```

The above two lines, informs all robots - since the wildcard asterisk "*" character was used) that they are not allowed to access any page in the cgi-bin directory and its sub-directories. That is, they are not allowed to access cgi-bin/any_file.cgi or even a file or script in a subdirectory of cgi-bin, such as /cgi-bin/any_sub_directory/any_file.cgi.

It is possible to exclude a spider from indexing a particular file. For example, if you don't want Google's image search robot to index a particular picture, say, some_personal_picture.jpg, you can add the following lines to your robots.txt:

```
User-agent: Googlebot-Image
Disallow: /images/some_personal_picture.jpg
```

2.2.2 Advantages of a robots.txt file

1. A robots.txt can avoid wastage of Web server’s resources.

Dynamic pages on a site which are coded in scripting languages (such as CGI/ PHP/ Python/ Perl) are not indexed by many of the search engine spiders. However, there are crawlers of some search engines that do index them, including one of the major players, Google.

For robots or spiders that do index dynamic pages, they will request the scripts just as a browser would, complete with all the special characters and request parameters. Generally, the scripts serve no practical use for a search engine as the values change depending on the request parameters, location of the browser or type of the browser – Web developers can create different pages to be viewed with specific browsers so that their site is browser-compatible and has a consistent look and feel. But here, we are talking about scripts that check site-navigation, error-reporting scripts, scripts that send emails, scripts that implement business logic and so on. It does not make sense for a search engine to crawl such types of scripts. - It can just crawl the direct links that use the scripts. Generally, scripts do not produce useful information when called on their own and many times, they do not produce links leading to other pages. An administrator might want to block access for spiders from the directories that contain scripts. For example, spiders can be blocked from accessing scripts in a CGI-BIN directory. Hopefully, this will reduce the load on the Web server that occurs due to execution of scripts.

Of course there are the occasional ill-behaved robots that hit your server at a high frequency. Such spiders can actually crash your server or at the very least hit its performance and as a result, slow it down for the real users who are trying to access it.

An administrator might want to keep such spiders from accessing their sites. It can be done with a robots.txt file. Unfortunately, ill-behaved spiders or ones with bad logic often ignore robots.txt files.

2. It can save a good amount of bandwidth

If one examines a site's access logs, there will surely be a fair amount of requests for the robots.txt file by various spiders. If the site has a customized 404 document instead of a default one, the spider will end up requesting for that document if a robots.txt file does not exist. If a site has a fairly large 404 document, which the spiders end up downloading it repeatedly throughout the day, one can easily run into bandwidth problems. In such a case, having a small robots.txt file may save some, even though, not significant bandwidth. It would also save some considerable bandwidth during a high time of the day when the server runs gets a huge amount of requests from users and it is being crawled at the same time.

Some spiders may also request for files which one feels they should not do so. For example, there is one search engine which is on a look out for only graphic files (.gif, .jpg, .png, .bmp etc.). If one does not see the reason why s/he should let any crawler index the graphics on his/her site, waste their bandwidth, and possibly infringe their copyright, they can ban it and other spiders from accessing their graphic files directory through a rule in a robots.txt file.

Once a page is indexed, a spider is likely to visit the page again to get a fresh copy, just in case the page was updated after its previous visit. If one catches a spider coming too often and creating bandwidth problems, one can block such a spider from accessing any document in the entire site.

3. Refusing a particular robot for copyright reasons.

Sometimes you don't want a particular spider to index documents on your site because you feel that its search engine infringes on your copyright or some other reason. For example, Picsearch[1] will download images and create a thumbnail version of it for people to search. That thumbnail image will be saved on their Web server. If, as a webmaster, you do not want this done, you can exclude their spider from indexing your site with a robots.txt directive and hope that the spider obeys the rules in that file and stop making a copy of documents from your site.

2.2.3 Shortcomings of using robots.txt

1. It is a specification, not a rule.

As mentioned earlier, although the robots.txt format is listed in a document called "A Standard for Robots Exclusion", not all spiders and robots actually bother to heed it. These days, generally all the bots obey it but some bots created specially to extract and

---

[1] http://www.picsearch.com

harvest information from your site do not respect the robots.txt file. Listing something in your robots.txt is no guarantee that it will be excluded. If you really need to protect something, you should use a .htaccess file or other access authentication methods.

2. Sensitive directories should not be listed

Anyone can access your robots file, not just robots. For example, type http://www.google.com/robots.txt on the address bar of your browser and it will get you Google's robots.txt file. It doesn't make sense for webmasters who seem to think that they can list their secret directories in their robots.txt file to prevent that directory from being accessed. Far from it, listing a directory in a robots.txt file often attracts attention to the directory! In fact, some spiders (like certain spammers' email harvesting robots) make it a point to check the robots.txt for excluded directories to spider. There is less chance that your directory contents are not accessed if you do not advertise it or link it from public area of your website. A better way is to password-protect the directories.

3. Only One Directory/File per Disallow line

Robots Exclusion Standard only provides for one directory per Disallow statement.Do not put multiple directories on your Disallow line. This will probably not work and all the robots that access your site will end up ignoring that line and accessing the directory and/or pages listed on that line.

2.2.4 Strengthening the defense against ill-behaved spiders:

If one has a particular spider in mind which one wants to block, one have to find out its name. To do this, the best way is to check out the website of the search engine. Respectable engines will usually have a page somewhere that gives you details on how you can prevent their spiders from accessing certain files or directories. One other way is to look at the access logs. One can also use tools like Webilizer, Mint, or other statistical tools that construct information out of server's access logs.

When one has such spiders in mind, one can create a daemon to keep these spiders at bay. This process can be executed as a proxy server. The algorithm for such a daemon process is as follows:

1. Load the robot rules in robots.txt from Web server's root directory.
2. For every request that comes in, check it against the set of rules.
    a. if the request fails: issue a restricted-access error page.
    b. else: pass the request to the Web server to be processed.

Another way is to use a combination of *RewriteCond* and *RewriteRule* directives provided by *mod_rewrite* module for the Apache server to keep spiders away from the website or parts of it. The mod_rewrite module can also be used to rewrite URLs as a part of search engine optimization techniques but this functionality is out of context here.

In connection with the F(orbidden) flag, the *RewriteCond* directive can contribute to keeping down the server strain caused by robots, for example, with dynamically generated pages or parts of the site which contain sensitive data. The following configuration snippet shows a *mod_rewrite* method to keep off such a robot, called "BadSpider". It is identified by its IP address. One cares for the robot's name to ensure that the Web server doesn't block out normal users who may be working from the same computer. Additionally, reading the starting page is allowed, so that the robot can read this page and enter it in its index.

```
#if the HTTP header user agent starts with "BadSpider" ...
RewriteCond %{HTTP_USER_AGENT}  ^BadSpider.*
#and requesting IP addresses are a.b.c.d and a.b.c.e,
RewriteCond %{REMOTE_ADDR}  ^a\.b\.c\.[d-e]$
#then prevent access to pages which start with /dir/sub-dir
RewriteRule ^/~dir/sub-dir/.+   -  [F]
```

A detailed explanation of the last technique of blocking spiders is available in an article entitled *URL manipulation with Apache* at http://www.heise.de/ix/artikel/E/1996/12/149/ by Ralf S. Engelschall and Christian Reiber.

Chapter 3: Implementation

This chapter explains the setup of the system. There is one machine which works as the scheduler machine. Its task is to assign seed pages or domains to the crawling machines. The crawling machines get the seed URLs and the domains they need to crawl. These then crawl the pages and follow links on the crawled pages.

Following topics are discussed in this chapter:

- Software and packages used
- Perl modules used
- Explanation of WWW::RobotRules::MySQLCache module.
- The configuration file
- System architecture and user manual
- Database implementation
- The Algorithm used.

3.1 Software and packages used:

Perl:

Perl (version 5.8.8) is used as the programming language. A fast interpreter, its features to handle and manipulate strings and relatively small memory signatures of its modules make it an ideal language for this project.

MySQL:

The database is designed and implemented using MySQL v3.23.58 and v4.1.1. MySQL is free, scalable and Perl has a rich API for MySQL.

PhpMyAdmin:

It is a good and intuitive front-end user interface for the MySQL database. Many features are provided to create, manipulate and manage databases and users in the MySQL environment. One can also see and adjust MySQL environment variables.

jEdit:

jEdit is a powerful IDE for coding in multiple languages. It is free of cost. Its features consist of syntax highlighting, auto code indentation, matching of braces, code folding, error checking, ability to handle multiple secure remote connections, ability to remember open documents while restarting and a very powerful search and replace facility with regular expressions.

Apache:

An Apache server v2.0.54 is used for the machines to communicate using the CGI module.

Software used for documentation:

Microsoft Visio was used for the diagrams and Microsoft Word was used to write the project report.

3.2 Perl modules used:

The following CPAN (Comprehensive Perl Archive Network) modules were used for the project.

1. LWP: It is a comprehensive module for Web related implementation. LWP::Simple is a small sub-module used for fetching pages. For more information, visit the CPAN website (http://cpan.perl.org).
2. File::Path: It is used to implement an operating system independent file system to store fetched pages. One can create, modify, list, read and delete directories and files and create, follow, modify and delete symbolic links (if it is available as a feature in the operating system).
3. DBI: DBI is a rich database independent interface for Perl. DBI::DBD is a Perl DBI Database Driver Writer's Guide and DBD::mysql is a MySQL driver for the Perl5 Database Interface (DBI). These modules are not threads compliant and database handles cannot be shared amongst Perl threads.
4. Threads: To start numerous machines, a multi-threaded implementation was designed so that the crawler machines can be started at the same time and while one machine is started, other machines do not have to sit idle.
5. CGI: It is used for communication among different machines. Crawler machines are started when a CGI script is called which starts the crawling script as a background process.

6. Digest::MD5: The md5_hex() function from this module is used to create a hash of the retrieved pages. This can be checked for changes in a page when it is crawled the next time.
7. HTML::Element: This module represents HTML tag elements. It is used to define tags by HTML::SimpleLinkExtor when it parses a downloaded Web page.
8. HTML::SimpleLinkExtor: This module extracts links from a Web page. One can specify what types of links are to be extracted. Links can be extracted from different tags like <a>, <img>, <form>, <link>, <script>, <frame> and others.
9. URI::URL is used to construct objects of URLs from links encountered in a Webpage.
10. URI is used to canonize relative and absolute URI objects provided by the above module.
11. HTTP::Headers is used to extract individual headers from a Web server's response.
12. LWP::RobotUA is a package for implementing a browser object. The parameters are discussed in the explanation of the configuration file.
13. WWW::RobotRules::Parser: This module is used to parse a robots.txt file.
14. WWW::RobotRules::MySQLCache: This is a extension for maintaining a cache of multiple robots.txt files in a MySQL database. Its working is explained next.
15. Carp: This module is used to issue warnings and errors at different intensity levels for problems occurring in modules.
16. DateTime::Format::Epoch is used to convert time from epoch seconds to current time format in year, hours, minutes and seconds and vice versa.
17. Data::Dumper is used to print data as strings for debugging purposes.

18. Config::Crontab: To manage crontab entries.
19. Time::HiRes: This module was used to keep track of time for different activities during the crawl

3.3 Explanation of WWW::RobotRules::MySQLCache module:

This module was created to persistently store robot rules extracted by the WWW::RobotRules::Parser module in a MySQL Database. The reasons to create this module are explained in Chapter 4. A description of MySQL tables used by this module and their relation is in the database section of this chapter.

The functions provided by this module are as follows:

1. new($databaseServer,$username,$password,$database):

Creates a new object of type WWW::RobotRules::MySQLCache. Database connection parameters are similar to those used in DBI.

a. database server to connect to.

b. username

c. password

d. database to be used.

2. create_db():

Creates two tables – location and rules to store the information.

location: Stores location of *robots.txt*.

rules: Stores rules extracted from *robots.txt* file(s).

3. *load_db($location):*

Loads rules from a *robots.txt* if we don't have them in the database. If the copy of *robots.txt* in database is not consistent with the one just fetched from the server, the database is updated.

4. is_present($location):

Checks if a particular *robots.txt* is present or not.

Returns 0 if

a) $location/robots.txt does not exists database OR

b) $location/robots.txt is not a recently updated file

Returns 1 otherwise.

5. is_allowed($user_agent, $url_or_folder):

Checks of a *userAgent* is allowed to fetch a URL or access pages in a folder.

Returns 1(allowed) or 0(disallowed).

3.4 The configuration file:

The file *config.txt* holds the parameters for the configuration of the system. There are configuration parameters for the scheduler machine, the crawling machines and the *userAgent* object. It is saved as a text file so that the crawling machines can fetch it from scheduler machine's Apache server as a text file. The crawling machines then include the local copy of *config.txt* as a file containing Perl scripts. An explanation of the parameters is as follows:

1. *weight_of_machine*: This is a multiplication factor for the weight of machines and the domains assigned to the machines. Possible values:
    a. a numeric digit
    b. number of domains * weight of machines field in the database.
2. location of log files: The directory relative to the scripts to store the log files.
3. *max_error_pages*: The maximum number of error pages to crawl before declaring a server to be non-operational.
4. *delay*: courtesy pause to be obeyed between consecutive fetches from one server.
5. *max_count_per_domain*: The maximum number of pages to be crawled from one domain in one cycle.
6. *max_time*: The maximum time in seconds devoted to one domain in one cycle.
7. *number_of_retries*: The number of retries for failed URLS.
8. *restrict_to_base_domain*: This parameter is used as a flag to report URLs from other domains not assigned to the crawling machine or not to report such URLs.
9. *threads_per_crawler*: For a multi-threaded version of the crawler; number of threads to be run simultaneously. Each thread runs for one domain.
10. *crawl_content_types*: This is a list of content_types of documents from which links are extracted and followed.
11. *save_content_types*: A list of content_types of documents to be saved on the disk.
12. *save_crawled_files_here*: The location or folder where the downloaded documents are to be saved.
13. *tag_types*: List of tags to be extracted from a downloaded Web page.
14. *store_mailto_links*: A flag parameter to save email addresses found on Web pages.

15. *max_redirects*: The maximum number of redirects to be followed for one URL.
16. *time_out_secs*: The timeout for the userAgent object while trying to fetch a page from a server.
17. *time_out_sec_on_fetch*: This is the maximum time for a page-fetch from a server. If the process takes more time than this parameter, the server is temporarily added to list of blocked servers.
18. *allowed_protocol_schemes*: It is a list of protocol schemes to be followed. Currently, this is set to *http*.
19. *depth_of_crawl*: This is an unused parameter. Web pages located not more than the depth of number of sub-directories are to be crawled.
20. *max_urls_per_site*: This is an unused parameter. This number indicates the maximum number of pages to be crawled from a Website.

Note: The two parameters above can be used if one experiences a server that has a page that leads to an unending dynamically generated pages to trap a spider.

21. *spider_name*: Name of the *userAgent* or a browser object.
22. *responsible_person*: Email address of a person to be contacted by administrators of Web sites for problems experienced due to the spider.
23. *query_method*: Default query method to be used to fetch Web pages. The values can be ‘get’ or ‘post’.

3.5 System architecture and user manual:

The system consists of one main machine called the scheduler and multiple crawler machines. Each machine has a local MySQL database, a Web server and Perl installed. Seed URLs are added to the database on the scheduler machine. The crawlers register themselves, i.e., add themselves to the database on the scheduler. A high level system diagram is shown in Figure 1. The working of the system is explained in the next section.

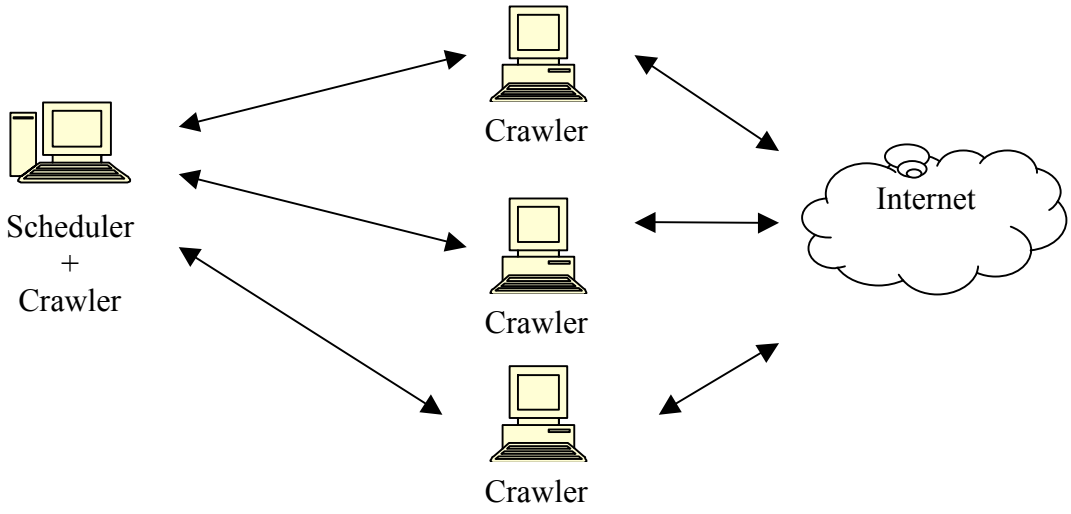

Figure 1: System Architecture.

The following are the steps to add a machine to the database and prepare it to crawl sites:

1. Move scripts from scheduler machine to the new machine. The folder *cgi-bin* is to be copied.

2. From browser on the new machine, call http://*<location of scheduler>*/~ankur/register_form.html and submit the form Pass phrase is academic_crawler. This will add the new machine to the database.

3. Change database connection parameters (database, username, password) in *cgi-bin/local_db_connection.pl* for local database connection.

4. Start the crawler manually by running perl crawl_all.pl *<ip address of current machine> <crawl_id of the current crawl> <ip address of scheduler machine>*. *crawl_all.pl* is in *cgi-bin* directory. The crawler does not need to be started manually. It will be started by a *cron* job on the scheduler machine.

To add domains to be crawled, fill out the form using a browser at *<location of the scheduler>*/~ankur/domains_register_form.html. Enter a seed page for the domain. Generally this is the root page of the domain.

To start a crawl, call the script *start_crawl.pl* on a command line. This will assign domains to the crawlers and start the crawlers.

To end a crawl, call the script *end_crawl.pl*. It will collect the information from databases on crawler machines and stop the crawler processes. It will also mark the current crawl as completed.

3.6 Database Implementation:

An explanation of fields in the tables is provided. Data from these tables will be shared among researchers for the Needle project. An explanation of relationships amongst the tables follows.

3.6.1 Tables on the scheduler:

Table s_domain:

Field	Type	Null	Default
total_pages	int(11)	Yes	*NULL*
size	bigint(20)	Yes	*NULL*
base_url	varchar(255)	No	
domain	varchar(255)	Yes	*NULL*
time_stamp	date	No	0000-00-00
start_time	time	Yes	*NULL*
stop_time	time	Yes	*NULL*
delay_high	int(2)	Yes	*NULL*
delay_low	int(2)	No	0

Table 1: *s_domain* table.

Table 1 contains information about the domains to be crawled. *total_pages* and *size* are two fields to be filled in during the crawl. These are updated at a regular interval. *base_url* is the seed url for one domain.

- *start_time* and *stop_time* define the range when the server load is expected to be minimum. These times are local times for the crawler. For example, a server in Arizona is expected to have low load from 1:00 AM in the morning to 7:00 AM, then the *start_time* for its record would be 00:00:00 and *stop_time* would be 06:00:00.
- *delay_low* is the delay between two consecutive page fetches from one single server when the Web server is expecting a lot of load, i.e., during the day.

- delay_high is the delay between two page fetches when the server load is expected to be low, for example, in the wee hours of mornings. *delay_high* is the delay when local time is in the range from *start_time* and *stop_time*.

Table s_machine:

Field	Type	Null	Default
machine_id	varchar(15)	No	
weight	int(2)	Yes	*NULL*
database	varchar(255)	Yes	*NULL*
username	varchar(255)	Yes	*NULL*
password	varchar(255)	Yes	*NULL*
scripts_loc	varchar(255)	No	

Table 2: *s_machine* table.

Table 2 contains information about the machines where crawler processes will run.

- *machine_id* is the IP address of the machine
- *weight* pertains to the number of domains the machine can crawl simultaneously. There is one crawling process running per domain.
- *database*, *username* and *password* are MySQL connection parameters to connect to the datatabase on the machine.
- *scripts_loc* is location of scripts on the crawler machines. This field was added so that different users can have a crawling process.

Table s_crawler:

Field	Type	Null	Default
crawl_id	int(11)	No	
start_time	timestamp(14)	Yes	*NULL*
end_time	timestamp(14)	Yes	*NULL*

Table 3: *s_crawler* table.

Table 3 contains information about each crawling cycle.

- *crawl_id* uniquely identifies a crawl. This field is also used to identify data belonging to each crawl.
- *start_time* is the start time of the crawl and *end_time* when the crawl ended.

Table s_assignment:

Field	Type	Null	Default
crawl_id	int(11)	Yes	*NULL*
machine_id	varchar(20)	Yes	*NULL*
base_url	varchar(255)	Yes	*NULL*
time_stamp	timestamp	No	0
crawl_done	tinyint(1)	No	0
c_pid	int(11)	Yes	0
delay	int(3)	No	0
last_crawl	bigint(20)	Yes	0

Table 4: *s_assignment* table.

Table 4 connects the machines, domains and crawling processes. Seed urls: *base_urls* of one or more domains from *s_domain* table are allotted to machines according to the *s_machine.weight*. This is done for every crawl process identified by *s_crawler.crawl_id*.

- *delay* is either *delay_low* or *delay_high* from the *s_domain* table.

- *crawl_started* and *crawl_done* mark the start and end of crawl for each domain.
- *c_pid* is the process id for the crawling processes on machines identified by *s_machine.machine_id*.
- *last_crawl* is the number of URLs processed the previous time pages were fetched from a particular domain.

Table s_machine_crawler:

Field	Type	Null	Default
crawl_id	int(11)	No	0
machine_id	varchar(20)	No	
url_count	int(11)	No	0

Table 5: s_machine_crawler table.

- Table 5 - *s_machine_crawler* is another relationship table between crawlers, machines and domains. But here, it is a count of URLs allotted to each machine for one crawling period.

Table s_linkage:

Field	Type	Null	Default
base	varchar(255)	Yes	*NULL*
link	varchar(255)	Yes	*NULL*

Table 6: *s_linkage* table.

Table 6 is a copy of *c_linkage* except that the links are normalized to be uniquely recognized across machines and across crawls.

Table s_url:

Data from crawlers is dumped into this table at the end of each crawling period. The table structure is the same except *url_id* is normalized to be able to uniquely identify a url across machines and across crawling cycles. The fields are explained below for the *c_url* table, i.e., table 7.

Field	**Type**	**Null**	**Default**
url	text	No	
content_type	varchar(100)	Yes	*NULL*
time_stamp	timestamp(14)	Yes	*NULL*
time_taken	integer(5)	Yes	0
crawled_at	datetime	Yes	0000-00-00 00:00:00
valid_till	datetime	Yes	0000-00-00 00:00:00
server	varchar(255)	No	
size	int(11)	Yes	*NULL*
url_id	varchar(255)	No	
crawl_id	int(11)	No	0
crawled	tinyint(1)	No	0
update_freq	int(11)	No	0
check_sum	varchar(50)	Yes	*NULL*
rank	int(11)	No	0
retries	tinyint(1)	Yes	0

Table 7: *s_url* table.

Table s_interdomainlinkage:

Field	Type	Null	Default
from_link	text	No	
to_link	text	No	
crawl_id	int(11)	No	0
time_stamp	timestamp(14)	Yes	*NULL*
seen_flag	int(11)	No	0

Table 8: s_interdomainlinkage table.

Links leading to pages hosted on servers outside the domain currently being crawled are dumpled into *s_interdomainlinkage* table, i.e., table 8.

- *seen_flag* is implemented as a flag for the scheduler to check if links are working or not. Following are the possible values for the *seen_flag*:
    - 1: If the link works
    - 0: If the link is not checked.
    - -1: If the link is broken.

Table s_server:

Field	Type	Null	Default
server_id	varchar(255)	No	0
server	varchar(255)	No	

domain	varchar(255)	No

Table 9: s_server table.

Table 9 contains list of servers and domains gathered from multiple crawlers

3.6.2 Tables on the crawlers:

Table location:

Field	Type	Null	Default
robot_id	int(11)	No	
location	varchar(255)	No	
created_on	datetime	No	0000-00-00 00:00:00

Table 10: *location* table.

- *location* table stores the location of *robots.txt* files.
- *robot_id* is the primary key for the records.
- *location* holds the domain where the *robots.txt* is hosted.
- created_on is the date when the *robots.txt* was created. This field is used to test if the copy of *robots.txt* cached on the crawler is a fresh or not.

Table rules:

Field	Type	Null	Default
robot_id	int(11)	No	0

userAgent	varchar(255)	No
rule_loc	varchar(255)	No

Table 11: rules table.

- *rules* contain the rules extracted from a *robots.txt* file. This table is used to check if crawlers are allowed to fetch a URL from the server or not.
- *userAgent* is the *userAgent* field in the *robots.txt*. Generally, this contains a * meaning a rule is set for all robots or spiders. Sometimes there are names of particular spiders.
- *rule_loc* contains part(s) of the website or page(s) that are not to be visited.

Table c_server:

Field	**Type**	**Null**	**Default**
server_id	int(11)	No	0
server	varchar(255)	No	
domain	varchar(255)	No	

Table 12: c_server table.

- *server* hold different servers from where the URLs are fetched.
- *domain* is the list of domains.

Table c_linkage:

Field	**Type**	**Null**	**Default**
base	int(11)	Yes	*NULL*
link	int(11)	Yes	*NULL*

Table 13: *c_linkage* table.

Table 13 holds the connection between the urls.

- *base* is the url_id of the containing page.

- *link* is the url_id of the linked page.

Table c_url:

Field	Type	Null	Default
url	text	No	
content_type	varchar(100)	Yes	*NULL*
time_stamp	timestamp(14)	Yes	*NULL*
time_taken	integer(5)	Yes	0
crawled_at	datetime	Yes	0000-00-00 00:00:00
valid_till	datetime	Yes	0000-00-00 00:00:00
server	int(11)	No	
size	int(11)	Yes	*NULL*
url_id	varchar(255)	No	
crawl_id	int(11)	No	0
crawled	tinyint(1)	No	0
update_freq	int(11)	No	0
check_sum	varchar(50)	Yes	*NULL*
rank	int(11)	No	0
retires	tinyint(1)	Yes	0

Table 14: *c_url* table.

Table 14 serves as a holder for the information gathered by the crawlers as well as a frontier.

- *url* is the URL of the pages
- *url_id* is the primary key. It uniquely identifies the URLs.
- *content_type* holds either the content_type of the pages, i.e., *text/html*, *application/pdf* and others or holds the HTTP error code returned by the server if the url is broken i.e. 404, 403, 500 and others.

- *time_taken* is the time taken to process the URL which consists of fetching the content type of content of the URL, fetching the document from the server and/ or extracting links from the page.
- *crawled_at* is the time when the process discussed was completed.
- *crawl_id* is the identity of a crawling cycle.
- *crawled*: This field determines the status of a URL. Possible values are as follows:
    - 0 if the URL is not crawled
    - 1 if the URL is crawled
    - -1 if fetching the URL results in an error
    - -2 if just a *head request* was performed on the URL.
    - -3 if the URL was ignored due to robots exclusion or a protocol mismatch.
- *update_freq*: When a record is shifted from a crawler to the scheduler, checksum of the page is checked with its checksum in previous crawls. This field holds the number of times the page was updated.
- *check_sum* holds the *MD5* checksum of the content fetched from a Web server for a URL.
- *rank*: When a URL is not crawled, i.e., it is still a part of the frontier, *rank* is implemented as a counter and holds the number of times it is linked from other pages. A rank is used to prioritize the order in which the URLs in the frontier are processed. URLs with a higher rank are crawled earlier than their counterparts with low rank.
- *retires*: This field holds the counter for number of retries on failed fetches. Failed fetches have -1 as value for the crawled field. Maximum number of retires is a configuration option in the *config.txt* file.

- *server*: It is the P*rimary key* of the server where the URL is hosted. Domain and server information is stored in the *c_server* table.

3.6 Database Relationships:

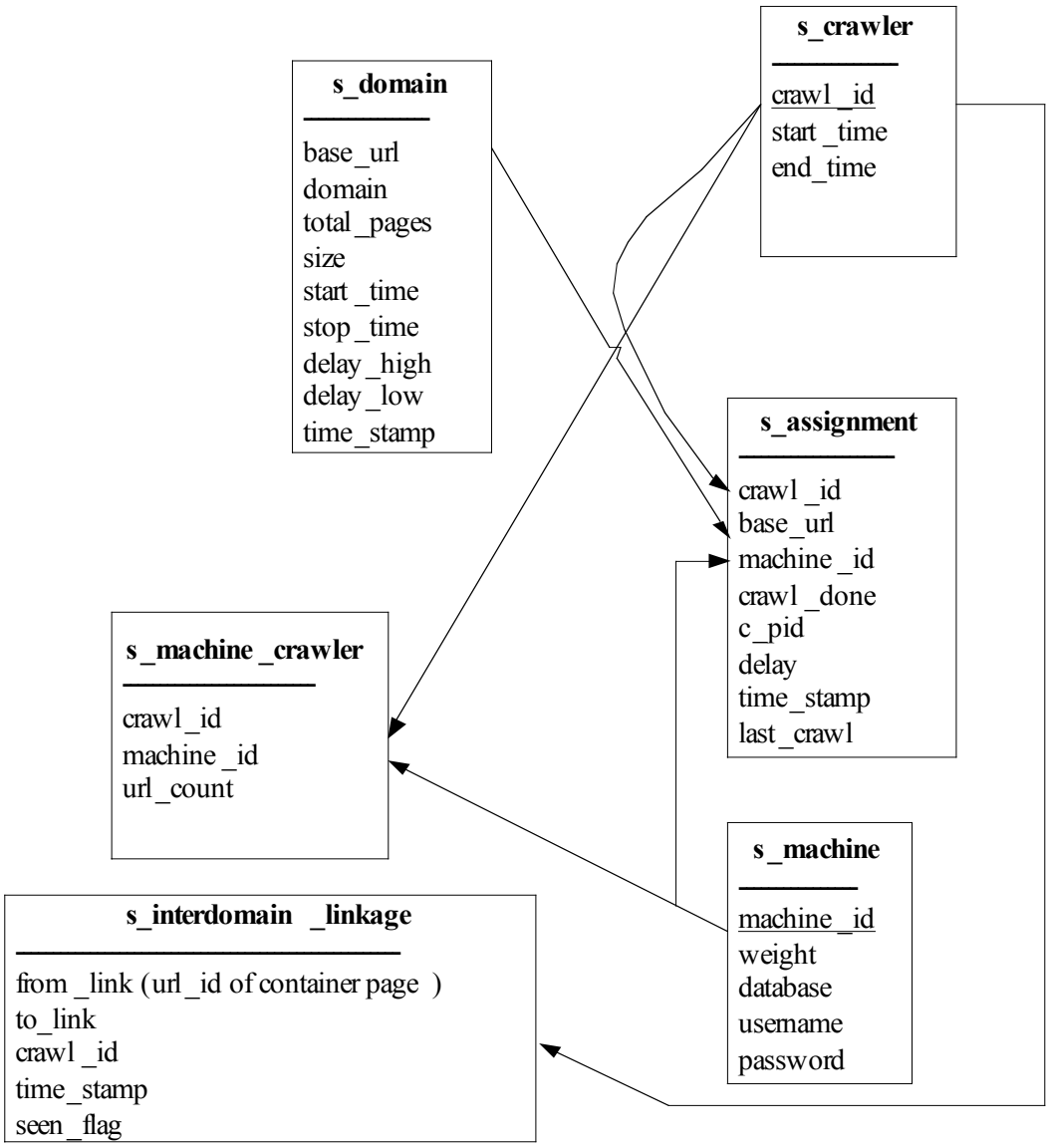

Figure 2: Relationships amongst tables on scheduler

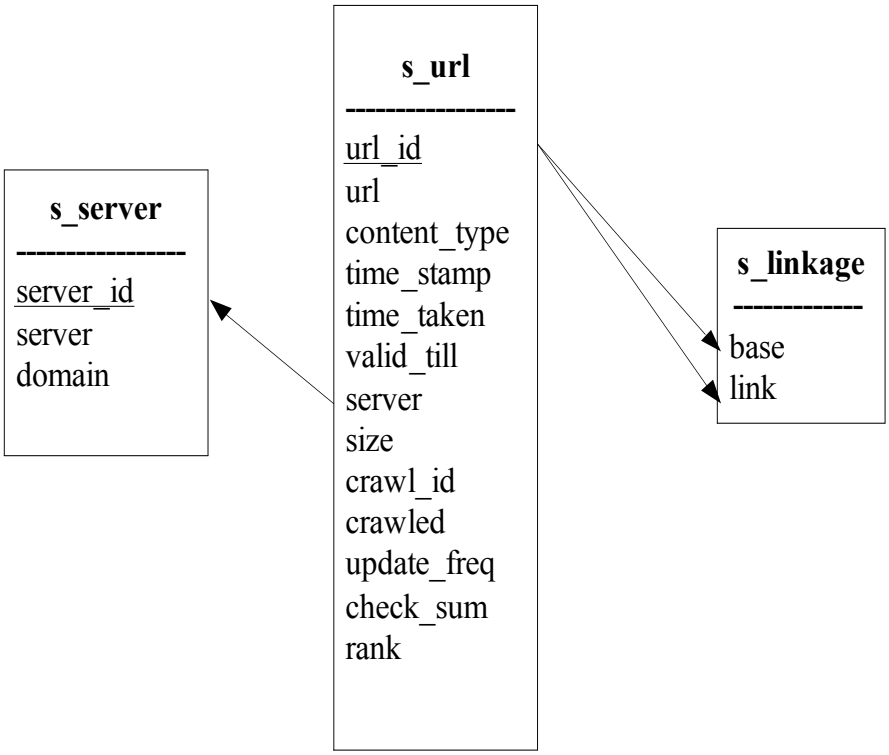

Figure 3: Relationship amongst tables on Scheduler for dumping data from crawler machines.

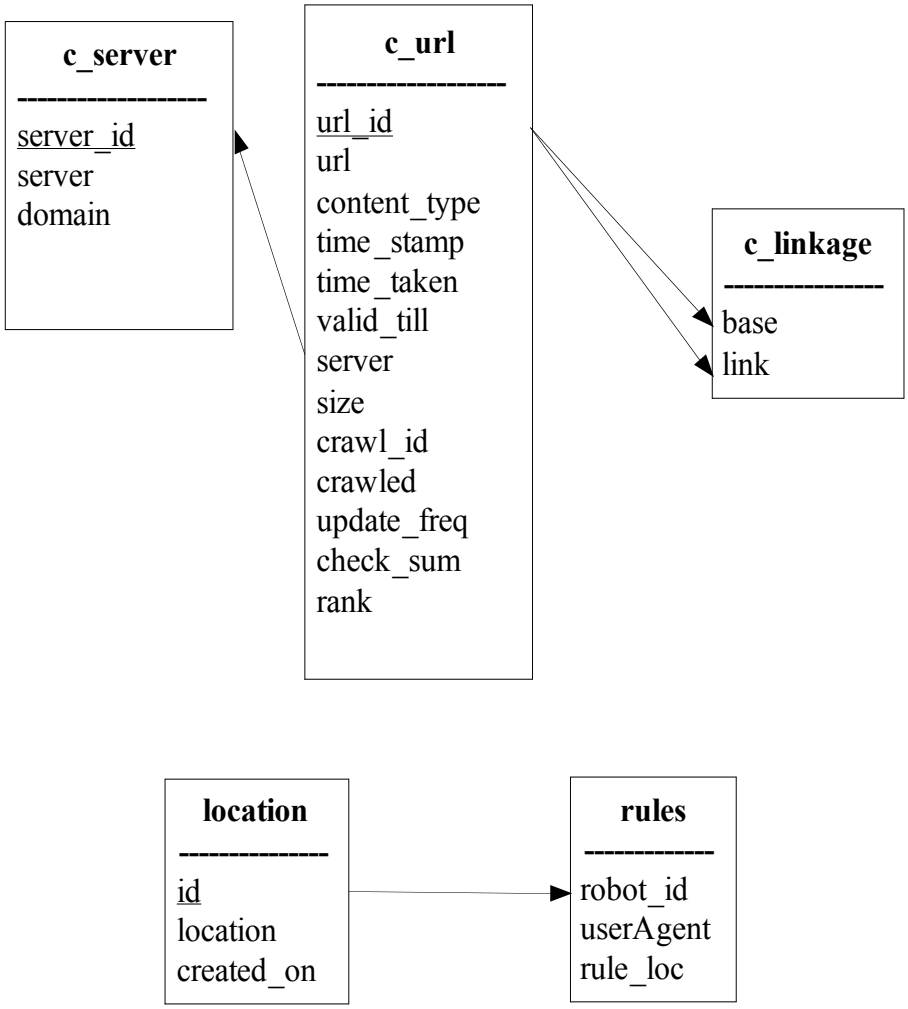

Figure 4: Relationships amongst tables on crawler machines

3.7 The Algorithm used:

3.7.1 Scripts on the scheduler machine:

a) Starting a crawl:

1. Start a new crawl by inserting a new row in *s_crawer* table.

2. Fetch unassigned domains from *s_domain* table.
3. Fetch machines from *s_machine* table. Note: It is assumed that the machines are registered in the database.
4. Assign domains to machines and save the assignment in *s_assignment* table.
5. Count domains per machine and store the information in *s_machine_crawler*.
6. Write *cron* jobs to start crawling machines (b: crontab entry 1) and fetch statistics (c: crontab entry 2).

b) crontab entry 1: starting or restarting crawling processes

1. Check for unassigned or new seed pages in the *s_domain* table.
2. Check for new machines in the *s_machine* table.
3. If both of above values are greater than 0, assign the domains to machines and re-calculate the number of domains per machine.
4. Fetch machines from the *s_machine_crawler* table.
5. For each machine, do the following:
    a. Kill the current crawling process if the *c_pid* field > 0.
    b. Start a new crawling process on the machine.

c) crontab entry 2: fetch statistics and check if the crawling process is running

1. Fetch machines in the *s_machine_crawler* table.
2. For each machine, perform following tasks:
    a. Try to ping it.
    b. Check if the crawling process is running or not.

c. Try to connect to its database.

i. If success, collect number of URLs crawled, uncrawled, ignored, which had just the head request performed and store this information into the *s_crawl_stats* table.

3. If any of sub-steps of 2 fails, email the administrator.

d) End a crawl

1. Get machines that participated in the current crawl from *s_machine_crawler* table.
2. For each machine, perform following tasks:
   a. End all crawling processes on the crawler machines.
   b. Append information in *c_url* and *c_linkage* to *s_url* and *s_linkage* after normalizing the *url_id* for each URL.
3. Mark the current crawl as finished by adding a *stop_time* in the *s_crawl* table.

3.7.2 Scripts on the crawling machines:

a) Crawling

1. Fetch a new copy of *config.txt* and *database.txt* from the scheduler machine. The file *database.txt* holds database connection parameters like host name, the name of the

database, the user name and the password for the database on the scheduler machine. Local database connection parameters are stored in *local_db_connection.pl*.

2. Get seed pages for domains assigned to this machine.
3. For each seed page, do the following:
    a. If the seed URL is found on the local database, restart the crawling process for the domain. Get distinct servers for the domain and the count of uncrawled URLs for each server. Also get the URLs with error pages but retries field is less than *number_of_retries* in the configuration file.
    b. If the seed URL is not found, fetch the page and extract links from it. Store the links in the database with a crawled flag value of 0. See extract_links (algorithm section b) for steps to extract links from a downloaded page. Mark the seed URL as crawled. Store the server where uncrawled links are located and the count of URLs for each server into main memory.
4. While the count of URLs for servers in the memory is greater than zero, loop through the servers and fetch a page that belongs to the server and has the highest rank from the database. Perform the steps mentioned in extract_links(algorithm section b) for each URL. Perform this step for all the servers having count of URLs greater than zero and if the server does not exist in the list of blocked servers.
5. Mark all the pages as error pages which are marked as not yet crawled but the number of retries is equal to the corresponding value in the configuration file.
6. Mark a domain as crawled if none of its pages are to be crawled.
7. If you run out of servers, go to step 3.

b) Extract links from a downloaded page:

1. Try to *ping* the server if server's *ping* status is not set
    a. If the result of the *ping* request is success, continue
    b. Else increase retries field by one for all the URLs hosted on the server and exit from the function. Add the server to the list of blocked servers.
2. Note down current time.
3. Perform a head request on the URL.
    a. If the head request is a success,
        i. continue for pages that have *content_type* similar to *crawl_content_type* entry in the configuration file. On success, continue to step 4.
        ii. Else update the record by saying that a head request was performed but the page was not downloaded.
    b. Else mark the URL as an error page and increment the number of error pages of the server by one.
4. Reset the number of error pages for the server.
5. If the number of error pages for a server is equal to its corresponding value specified in the configuration file, add the server to the list of blocked servers. Exit the function.
6. Perform a page download.
    a. If the download is a success, go to step 7.
    b. Else mark the URL as error and increment its retries value by one.

7. Extract links from the downloaded document. The tags to be extracted are specified in the configuration file.
8. Remove duplicates from the extracted list.
9. Remove links which are navigational tags on the same page.
10. Ignore email links. Save these links in a file if specified in the configuration file.
11. For each link, canonize a relative URL and perform following steps:
    a. If the link exists in the database and if it is uncrawled, increment its rank by one and add the linkage: from the current URL to a link in the *c_linkage* table.
    b. If the link exists in the database and it is crawled, just add linkage.
    c. If the link does not exist in the database, add it to the database and add the server to the list of servers.
12. Mark the current URL as crawled.
13. Note the current time and if the difference between this value and the one noted in step 2 is greater than the value of *max_delay_on_fetch* specified in the configuration file, mark the server as too slow and add it to the list of blocked servers.

3.7.4 Calculate performance of a machine and balancing the load amongst crawlers.

To calculate the performance of a machine and balance the load, following steps are implemented:

1. If the crawl is restarted, calculate number of pages explored.
2. Calculate total pages crawled for each hour and a cumulative average of pages crawled over the period.

3. If the current total of pages is less than the total of pages explored in the first hour by more than 70% for more than 50% of crawlers in the system,
    a. Stop the crawlers
    b. Count the number of servers under each domain which have uncrawled URLs and calculate the average. Sort the domains according to this average in a descending order.
    c. Calculate the percent of pages processed by each machine after the last restart.
    d. Sort the machines according to the percentage of pages processed.
    e. Assign the domains to the machines based on the above number.
    f. Restart the crawlers.

Chapter 4: Problems faced and Solutions

Even if the project differs from work done by previous researchers – Zhang [YDI 2006] and Patankar [SNC 2006], there were common issues which were kept in mind like scalability of the system, efficiency of the crawlers, a good restart policy for the crawlers and analyzing the time taken by different parts of the crawler i.e. selection of next URL to crawl, download of a Web page, testing of the links encountered and updating the database.

Following is a list of problems faced during the development of the crawler. Also listed are the methods to solve them and the solutions.

4.1 Starting the system, a heartbeat protocol and communication amongst the machines.

The first problem was to devise a way for the machines to communicate effectively over the network connecting them, not overloading the network at the same time. One of the goals of the project was to minimize network overhead due to intra-system communication. There are many options for machines to communicate like p2p connection, using ports and TCP-level connection and alternatively using the message passing interface. These options seemed to be cumbersome as all that is needed is a persistent connection, which in turn hogs the network. There should be a seamless way for the machines to communicate only when needed and the communicating machines should be able to report on a non-functional parts of the machine, i.e., the machines

should be able to check if the program is not working or the database could not be contacted or the other machine could not be found on the network. The previously mentioned methods do not work at the application level and it is difficult to test programs or database if the underlying connection is not working. If a machine goes offline, the administrator or the person in charge should be notified and other machines should be able to acknowledge that a machine has failed. The main problem was to start the crawlers via a single switch on the main machine and devise a non-intrusive heartbeat system for machines to check on each other. A heartbeat system or protocol is used to check on other machines if they are working or not.

Solution:

One of the options explored for kick starting the crawler machines was to use *SSH* connection which is not better than a *TCP* connection although the connections are secure. It did not work because for each connection to be made for a password had to be typed in manually. It would work great to monitor all the crawlers but there was no way to tell if the scheduler stopped working. Also like the *TCP* connection, a script had to be started on each crawler which listened for incoming connection requests at a particular port. That would require manually going to each crawler and starting the script. This would not fulfill the criteria of having a single startup switch.

Remote Procedural Calls (RPC) was tried too. These also needed a program running at one end, i.e., the provider's end.

Using message passing interface was not an option as it required the machines to be in a cluster. One of the design goals was to add and remove machines from the system as easily as possible and keep the system loosely coupled.

For the current system, there are crawlers which crawl non-intersecting domains and the only machine they contact is the scheduler machine. This design needs 2 * n unidirectional connections or n bi-directional connections to be maintained for n machines. The communication needs to be at the application level and asynchronous to be non-intrusive and non-persistent. CGI worked well for the problem of starting the system. A script on the scheduler would call LWP->get() for a CGI script on the crawlers which start a background crawling script local to that machine. This setup worked as the communication need not stay persistent and the life of the communication was maximum 5 minutes after which the CGI would give up. The connection can be broken after the background process is started but as the crawling process is implemented using threads; the CGI cannot end the connection by itself.

In one implementation, the scheduler machine tried to connect to the database of the crawlers for getting the statistics of the crawler like number of URLs the machines have explored. As discussed in the implementation, the number of URLs explored is the sum of the number of the URLs ignored, the URLs that could not be downloaded, the URLs that were ignored based on the content type and number of URLs downloaded and crawled. If the scheduler is not able to connect, it notifies the administrator about the failed machine. All the machines contact the database of the scheduler to report links to

pages which are outside the domains been crawled resulting in a non-intrusive, uni-directional asynchronous communication method that also worked as a heartbeat protocol and minimizes the exchange of URLs amongst the machines. If the crawler machines could not reach the scheduler machine, they would email the administrator and shut down because links outside the assigned domains could not be reported any more. A faster method to restart the machines is discussed later.

For an implementation under consideration, databases on the crawler machines were dropped and only the crawling process was checked upon if it is working or not. Crawlers would fetch URLs from the scheduler machine and crawl the pages. They would then update the database on the scheduler machine. A problem with this design is discussed next.

4.2 Decrease in URL exchanges amongst the crawlers

For a distributed system, the network usage is of prime importance. As mentioned in the background research chapter, researchers state that a large number of URL exchanges can hog the network.

Solution:

The design of the system was developed in a way such that one can have individual control over the machines performing the task of crawling. These machines behave as independent crawling systems except when they want to start or restart crawling the domains. The scheduler machine assigns domains to the crawlers and

regularly checks if they are accessible and the crawling process on the machine is working. The machines report URLs outside the domains they are crawling. The reporting would fail if the scheduler machine itself is not working or the database is not accessible. This condition is however; assumed not acceptable. In such a case, the crawling machines would stop their processes and notify the administrator.

One other design to be considered is to avoid having a database on the crawler machines and to have all the crawler machines connected to the database on the scheduler machine. This idea was dropped for the same reason that there would be a huge number of URLs (all the URLs) being transmitted from the crawler machines to the main machine. If one calculates database usage for the current database design, the database is used numerous times for each URL:

- For insertion of a new URL
- For selection of a URL to be crawled.
- Updating the record for a crawled URL
- Multiple searches if a URL is seen before or not and,
- If the URL undergoing a check is in the frontier, its rank is updated.

The current design of having autonomous crawlers being part of the overall system solves the problem of multiple URL exchanges amongst the machines.

4.3 Caching of Robot Rules.

The rules in robots exclusion files had to be cached somewhere in order to avoid searching for a *robots.txt* file and downloading it frequently. It would be easier if most of the data would be in one consistent format and easily updated if the *robots.txt* file on a server was updated.

Solution:

WWW::RobotRules just parsed the incoming *robots.txt* file but did not cache the rules. There is a persistent version of the above module called *WWW::RobotRules::AnyDBM_File*. It is persistent in the sense it stores the *robots.txt* in a disk file. However, it didn't use the previously saved copies of a *robots.txt* file. Every time a robot rule had to be checked; a new copy was downloaded and appended to the disk file. This resulted in retrieving multiple copies of the same file. There was no way of removing the duplicates and the disk file's length grew very fast adding in little functionality.

A module was needed which had similar functionality as *WWW::RobotRules::Parser* such as parsing the fetched *robots.txt* file and being able to query the set of rules if a *userAgent* object is allowed to crawl a URL or not. The module should be able to store robots exclusion files from multiple servers but should have one copy of the file. It should check if the cached copy of a *robots.txt* file is current or not; and if there exists a new copy of robots.txt file on the server, the module should be able to replace the cached copy of rules with the new one. There should be a way to see if we have a copy of a *robots.txt* file or not. It should also provide a functionality to check if a

single URL is allowed to be crawled by my *userAggent* regardless of server or location of the URL.

A module was created with above mentioned functionality and uploaded on CPAN (Comprehensive Perl Archive Network). The module is called *WWW::RobotRules::MySQLCache* and it uses *DBI*, *WWW::RobotRules::Parser*, *WWW::LWP::Simple* modules available on CPAN. One can create the required MySQL tables given the connection parameters to a MySQL database with a function call. One can check if a *robots.txt* file is present in the database before inserting another copy. The function will return false in two cases: *a robots.txt* file does not exist in the database or a *robots.txt* file is not a recently updated,. i.e., there exists a fresh copy of *robots.txt* on the server and the module has an outdated copy. The module works on two tables: location and rules. The working of the module is explained in chapter 3. *WWW::RobotRules::MySQLCache* is freely available and can be downloaded from CPAN at http://search.cpan.org/~apatwa/WWW-RobotRules-MySQLCache-0.02/lib/WWW/RobotRules/MySQLCache.pm. The version is subject to change without prior notice and can be redistributed and/or modified under the same terms as Perl .

4.4 Filter on URLs to be crawled:

There are huge application files, like Flash files, audio and video files which might not have any links and the browser takes a long time to download them. This induced a major hit on the performance of the crawler. The crawler process would

occasionally halt or take a long time trying to fetch such files. The time taken to download such files was observed to be up to 10 minutes.

Solution:

A list of content-types was created to narrow the types of files the crawler would crawl. Generally, *text/html* is the content-type for pages having links leading to other pages. A time consuming but necessary step of finding out the content type of the page before crawling was added to the crawling system. Pages having content types other than *text/html* like PDF files, audio and video files, text files were filtered out and were not crawled. This method worked for most of crawl but it failed during certain pages which were listed as dynamic pages with an extension of *.php* having the application file as a parameter in the URL. The delivered content of such URLs consisted of video files.

For another method, the content-type of the URL was predetermined by implementing a *head request* for the URL. The *userAgent* object would perform a head request on the URL and determine the content-type. This is a fool-proof way to determine the content type but it comes with a penalty of time. The request goes all the way to the server and fetches the information. This adds a delay of about 0.67 seconds to the task of processing of each URL but it does work every time without an exception. The implementation of fetching header information before download of a page was dropped to increase the throughput of the crawlers.

4.5 Increasing the efficiency of the database:

The data type of the *URL* field in the database was set to *BLOB*; which in MySQL is *TEXT*. One cannot define a hashed index on such a data type resulting in slow retrieval during searches. Patankar in her work [SNC 2006] experienced the same problem.

Solution:

A *FULLTEXT* index was defined on the column and searches were performed using *match* function available for searching strings on a *FULLTEXT* field in a boolean implementation. The non-boolean version returns multiple rows and a rank stating how close the match was. During a search using the *match* function, every character of each row is matched till a perfect hit is found. This increased the processing time of each search. But the cardinality of the index was too low for the index to be efficient.

The data type of the URL field was changed to *VARCHAR* with a length of 255 characters (maximum allowed length) and a hashed index was defined for the field.

Moreover, a limit on results was added to searches where just one row was expected. For example, when the crawler wants to get a URL to crawl, it just needs one URL. For such a query, adding *LIMIT 1* at the end of the query improved the performance of the query as the database engine would stop looking after a matching row was found. Indexes were introduced on the fields on which a search query was executed.

4.6 Decreasing amount of DNS queries:

In previous works, the authors stated that DNS lookups can be slow. In [JHH 2006], the authors resolve DNS names before starting the crawl. But it may be possible that servers change IP address; though not frequently. There has to be a way to find the IP addresses of servers during execution of the crawl.

Solution:

When a crawl starts for a number of servers, a *ping* request is sent to the servers to find out their IP addresses. The server names for URLs of corresponding servers are replaced by their IP addresses. The IP addresses are not stored but rediscovered for each time a crawl is started. This adds a small delay of about 10 seconds for the first URL of every server. However, this method results in avoiding DNS lookups for consequent URL fetches by the crawler. This method of dynamically resolving DNS names is transparent to change in IP address of the servers. This method was dropped after auditing log files. The log files indicated a rise in the number of error pages. These pages had a *temporarily moved* error status. Moreover, if a cluster of server exists, the page for residing on those servers will be processed by a different server. A crawler should not fetch pages from the same server and should be able to acknowledge such servers.

4.7 Configuration of delay between fetches per server.

The delay between fetches from the same server is called *courtesy pause*. This value has to be optimum. If it is too high, the crawler's throughput is low. If it is too low, one ends up in overloading a server with frequent visits.

Solution:

In previous works, researchers have set the value of courtesy pause between 10 and 30 seconds. Authors of [HNM 1999] have a dynamically set delay based on time it took to fetch previous page.

It is observed that during night, based on the location of servers, the servers experience considerable low load than their load during day time. Two period of hours were set for each server dividing 24 hours – the *high tide* where the courtesy pause is low and there can be more fetches from the same server compared to the *low tide* where there are few fetches and the courtesy pause is greater. There were two different delays set; one for each period of time. During the day, there can be a larger courtesy pause; which can be decreased during the night time. Keeping a high delay during day time has two advantages. First, the crawler can perform some fetches; which better than nothing; and second; there is a guarantee that the servers are not getting overloaded resulting in no or fewer complaints from their administrators. The length of the periods can vary. One can set the high tide to be from midnight to early morning if the administrators are contacted and they want their servers to be crawled during such hours.

4.8 Detection of an off-line or a slow server.

Sometimes, the servers can be offline for maintenance purposes or other reasons. If a server experiences high load of requests, it can take more time to respond to each URL request. The crawler should be able to detect such servers and ignore them for a time.

Solution:

There are two lists of servers kept in the memory. One list contains the servers from which URLs are to be fetched and two, a list of servers to be ignored for some time. Time taken to process each URL is recorded. Servers of URLs taking more than 60 seconds were shifted from the first list to the list of ignored servers. It was found that URLs taking more than 60 seconds took 70% of the crawling time. There were 4% of such URLs. A configuration parameter is provided in the *config.txt* file.

URLs of servers previously ignored are processed next time the crawler process is restarted. It has been observed that the time when the server is experiencing peak load is not more than a couple of hours. The crawler process is forced to restart every 6 hours.

A count of pages resulting in error pages for each server is recorded. This count is updated for every URL processed. It is reset to zero on a successful page download for a URL and it is incremented every time an error page is discovered. Assuming any page on

a server does not have more than ten broken URLs, if the error counter reaches a value of five, the server is added to the list of ignored servers.

4.9 Increasing the throughput of the crawlers:

During the first implementation of the crawler, a thread was assigned the tasks to crawl one domain. There were two such processes running on each machine. But due to a low number of servers from each domain, the program ended in no activity till the amount of time of *courtesy pause* was observed for every server.

Solution:

The multi-threaded crawling method was dropped and the crawling process was brought down to one process per machine. This process cycled through servers of all the domains assigned to it by the scheduler. Due to a large number of servers to crawl, the program didn't have to wait for courtesy pause to end for a server. It can fetch a page from the next server while *courtesy pause* for one server was observed. Patanker discusses the same strategy in her work [SNC 2006]. But this method broke the logic of site-level management scheme implemented in the multi-threaded implementation.

Queries were analyzed using the *explain* tool. Adding indices on fields used in the searches brought down the search time to one-third of time consumed before for the same task. For the *c_url* table, the *server* field was changed to an integer referring to *server_id*

in the *c_server* table. An index was added on the server field in the *c_url*. The idea of redefining indices can be found in [SNC 2006].

A huge amount of time was consumed in checking if a URL was present in the database. For this problem, solution used by Patankar in [SNC 2006] was implemented. *MD5* hash of the URL is stored in the database and for checking if a URL is present or not, a runtime hash value of the URL is checked in the database. This reduces the number of bytes MySQL engine has to compare for each URL. *MD5* produces a hash value of 32 bits. An index of 6 bytes was found to be sufficient to get a cardinality of the index equal to the number of rows in the database.

Chapter 5: Experiments and Results

A total of 4,518,808 pages were downloaded over a period of 12 days with a down time for the machines of about a day resulting in 4.5 GB of downloaded data from 2,219,436 pages. A total of five machines were used for the task of crawling. The pie chart in figure 5 shows the distribution of pages according to their category.

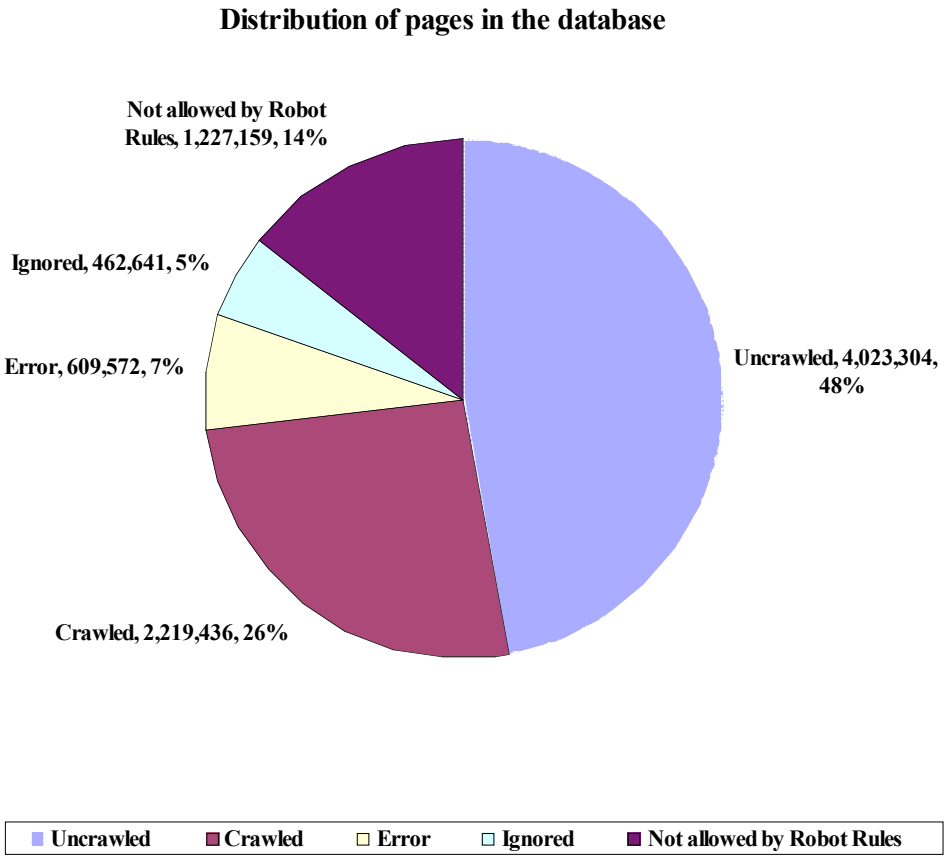

Figure 5: Distribution of pages in the database

Optimizing the tables:

A boost in performance was observed after optimizing the tables and redefining indices over the fields which were included in crawls. Figure 6 compares the increase in number of pages explored per hour on one of the machines. The hours are four hours before and after the task was performed

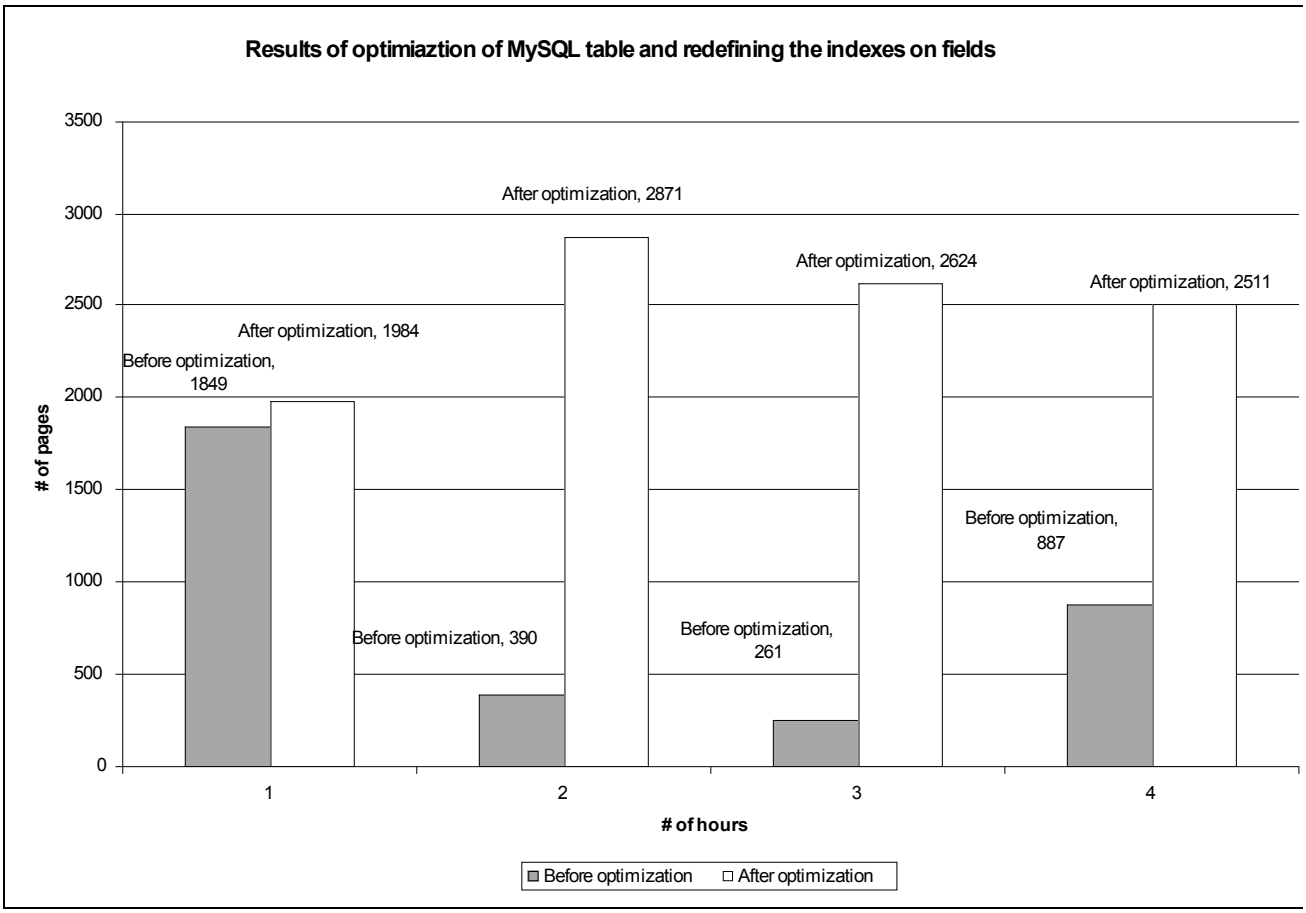

Figure 6: Results of optimization of MySQL table and redefining indexes on fields

However, similar results were not observed for all the machines. It is advisable to check the cardinality of the indexes at a period of three to four hours depending on how many new rows are added. Optimizing the table using MySQL *optimize table* should be performed at regular intervals for maintaining the efficiency of the database engine.

Load Balancing

Load distribution was performed for the domains which lasted for 1441 seconds; relocating 73 out of 92 domains.

Figure 7 shows the difference in performance of a machine for four hours before and after the activity of load balancing.

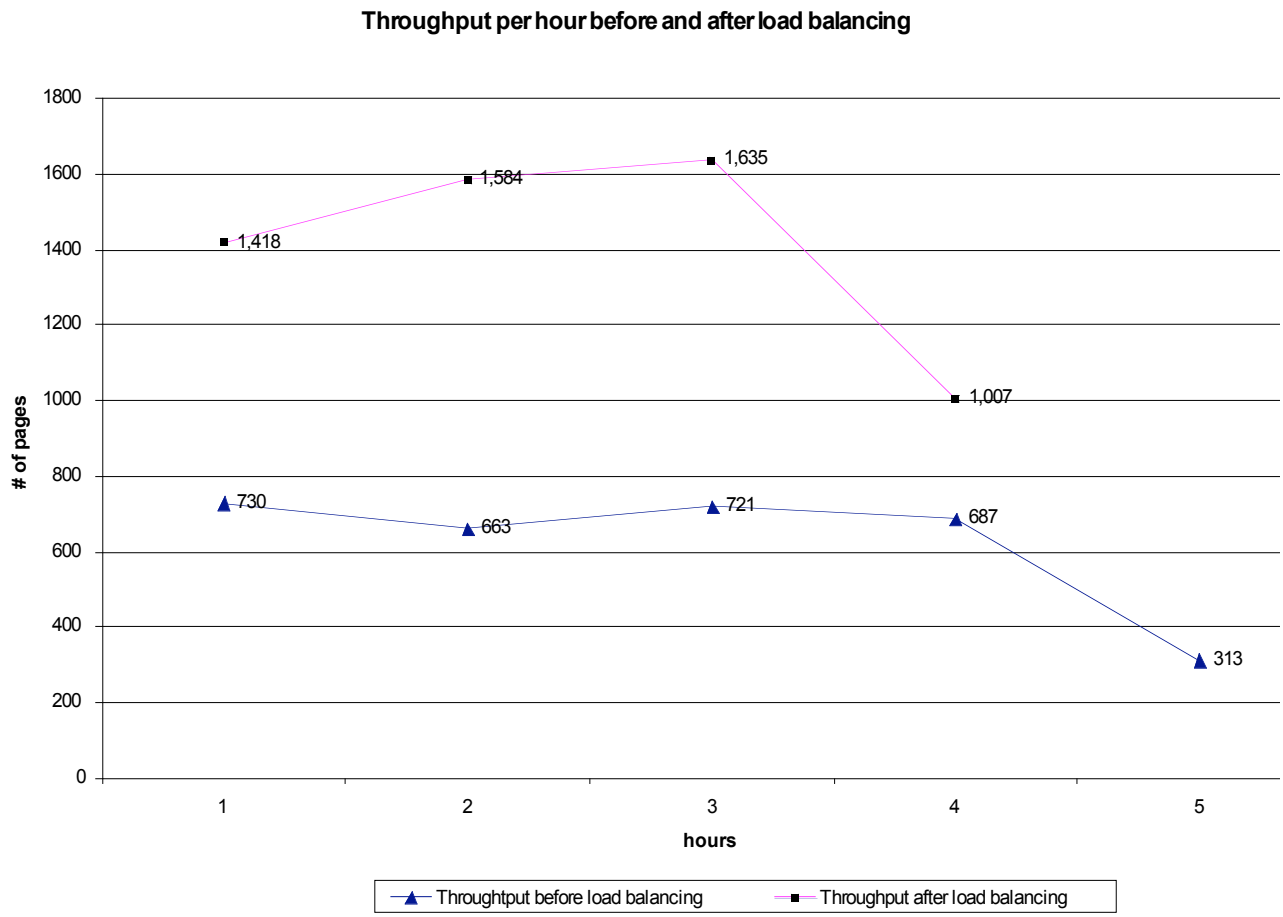

Figure 7: Throughput per hour before and after load balancing

The throughput decreases by 30% during the duration of four to five hours. Considering an average of 665 pages an hour, 226 pages were not crawled by a machine. After load balancing, the average throughput is 1411 pages per hour for 4 hours. The gain in throughput is 440 pages per hour. Similar increase in performance was observed on other machines. It is assumed that load balancing should be performed every six to eight hours depending on the throughput of the machines. A machine's performance over a period is to be logged and the number of domains to be assigned should be calculated accordingly. A machine's performance can take a dive if it is assigned servers which have a high response time and a number of these servers are ignored at the same time

Factors that affect the performance of the system are:

- Low processing power of one or more machines.
- A machine is assigned relatively small domains and the task of crawling is complete.
- A machine is assigned a domain that has few servers. In this case, the machine has no task to perform for the machine for the minimum period of *courtesy pause* while going through the list of servers.

.

Time distribution for activities:

Figure 8 shows time taken by different activities. This is the time taken to process the latest 100,000 URLs processed over four machines, out of which 66,400 URLs resulted in a download of documents. Total time to process 100,000 URLs = 62184.78 seconds with a speed of 1.6 URLs per second. This data is taken over time when all four

machines were fully functional. Down time for the machines is not considered. This is the performance attained after the code was optimized.

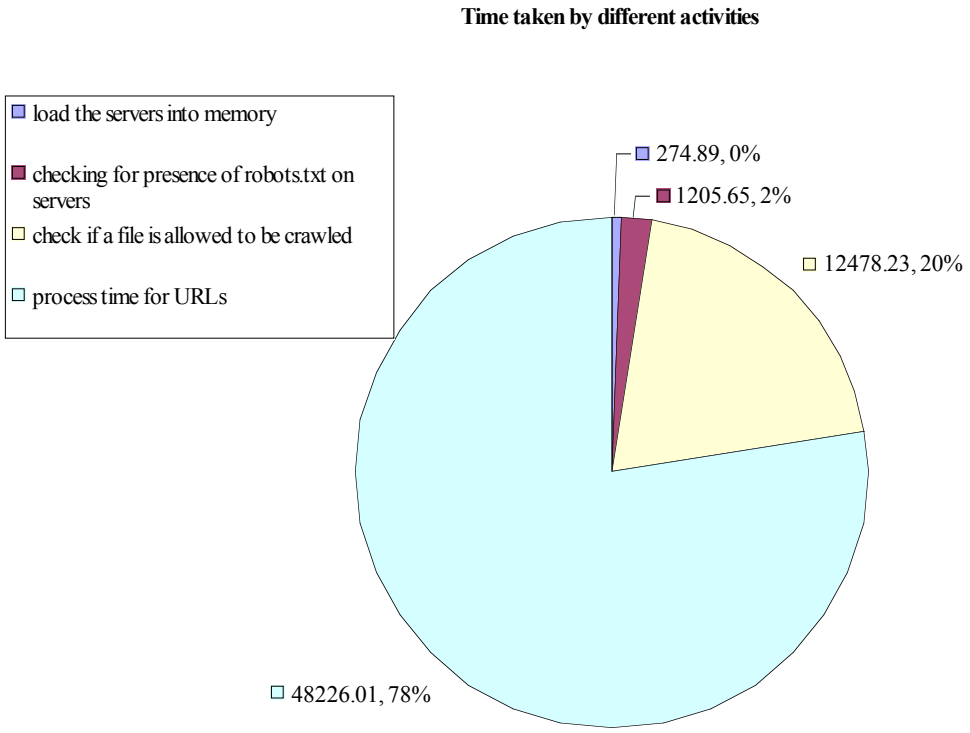

Figure 8: Time taken by different activities

Scaleup performance:

An ideal scaleup curve is a horizontal line. The scaleup graph for five machines is as shown in figure 9. The time to load servers and time to check if a robots.txt file exists on a server in the database are not increasing drastically. Time taken to check if a URL is allowed to be crawled depends on the number of servers assigned to the machine. These

three times are highly independent. The number of links downloaded is dependent on what time of the crawl the data is captured.

With this data, if one wants to achieve the performance acquired by Google as shown in table 15, i.e., 24 Million downloaded pages using 5 machines at the rate of 2.12 pages per second; it would take 131 days to complete the task.

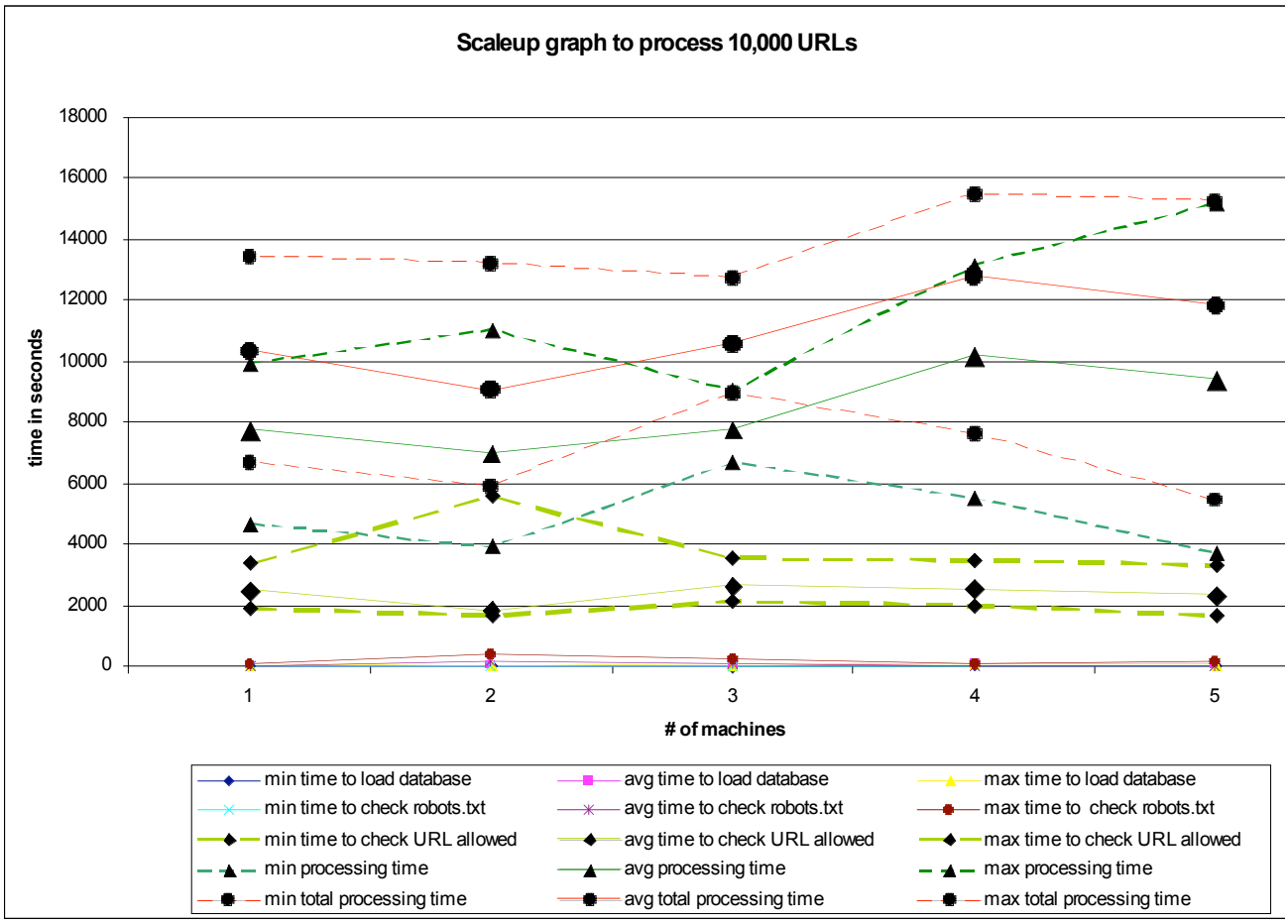

Figure 9: Scaleup graph to process 10,000 URLs.

Figure 9 graphs the time taken by different processes per machine to process 10,000 URLs. The X-axis represents number of machines in the system and Y-axis represent time in seconds. The solid lines represent average time taken for each activity

and the dashed lines with same color and markings above and below it represent maximum and minimum time taken for the activity respectively.

The red lines with circles as points represent the total time taken to process 10,000 URLs. This is the sum of time taken to load the servers into the main memory from the database, time to check the presence and fetch new copies (if available), of robots.txt on the servers, time to check if a URL is allowed to be crawled and processing time of the URLs which includes download of the page, extract URLs, check if the URLs are present in the database and/ or add the URLs to the database. As noted, the total processing time varies in congruence to the processing time for the pages.

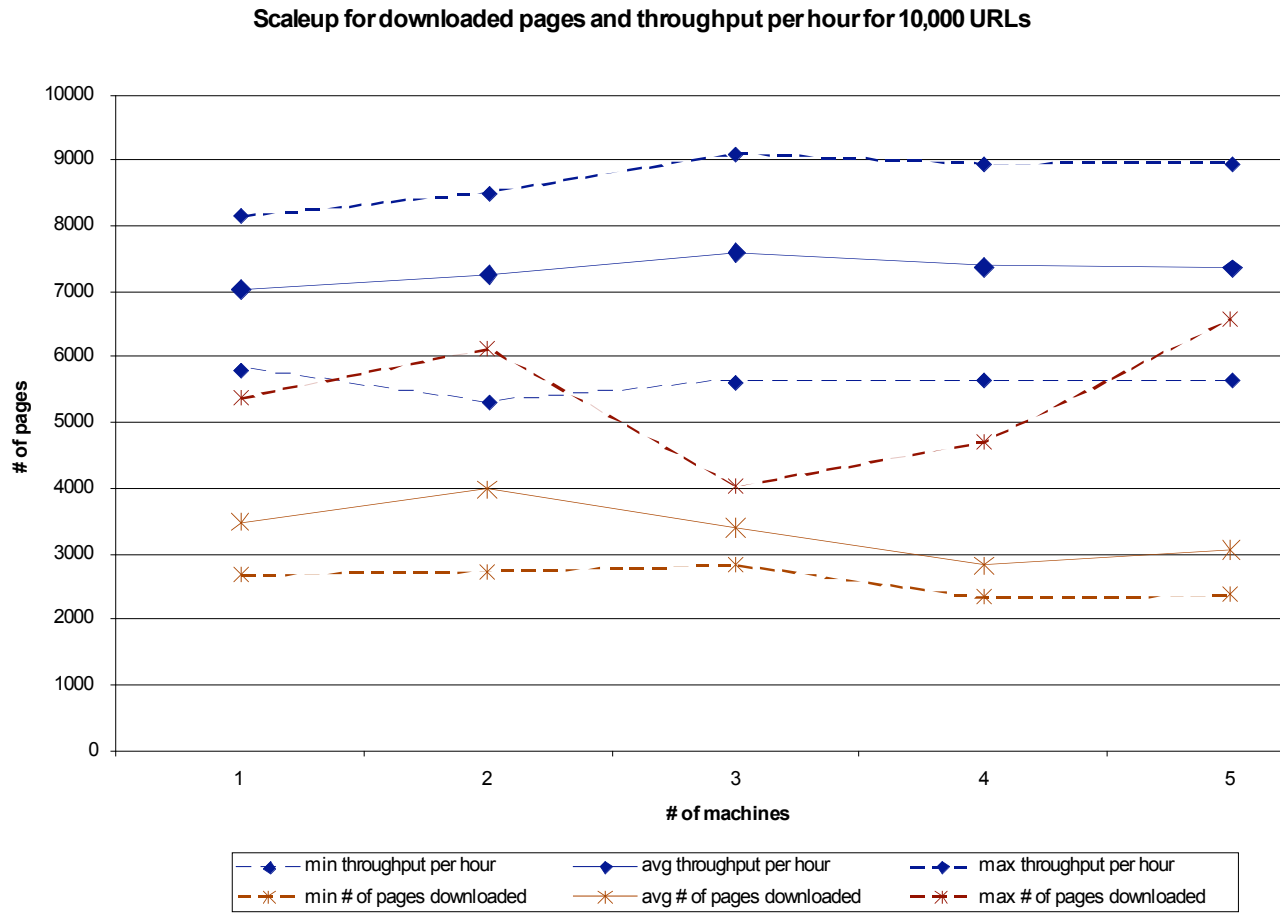

Figure 10: Scaleup graph for downloaded pages and throughput per hour for 10,000 URLs.

Figure 10 shows the throughput per hour per machine and the number of pages downloaded per machine in the system for different number of machines in the system. The data is captured for processing 10,000 URLs on each machine. The X-axis shows the number of machines in the system and Y-axis is the number of pages. The solid lines represent the average number of pages while the dashed lines in same color and same style of markings above and below the solid lines represent maximum and minimum readings respectively. As seen, the number of pages downloaded does not depend on the

throughput of the systems. The throughput can vary on adding more machines but the number of downloaded pages for each machine does not fluctuate much.

Comparison of performance with other systems:

The data for other systems is taken from [SNC 2006].

Crawler	Distributed Needle	Needle	Google	Mercator	Internet Archive
Year	2006	2006	1997	2001	-
Machine Configuration	Intel P4 1.8 GHz, 1 GB RAM, 225 GB Hard disk	Intel P4 1.8 GHz, 1 GB RAM, 225 GB Hard disk	-	4 Compaq DS20E 666 MHz, Alpha servers	-
Data structures for URLs	Perl Scalar	Queue	-	Memory Hash table, disk sorted list	Bloom Filter per domain
DNS Solution	-	Stored locally in database	Local Cache	Custom	-
Programming Language	Perl	Perl	C++/ Python	Custom JVM	-
Parallelism per machine	1	1	-	100	64
# of crawling processes in the system	5-6	1	500	-	-
System size	5-6	1	-	4	-
Number of pages	1 Million	1 Million	24 Million	151 Million	100 Million
Crawl Rate (pages / sec)	3.85	1.7	48	600	10
Effective Crawl Rate (pages / second) = crawl rate/ (system size x parallelism)	0.77	1.7	-	600/400 = 1.5	10/64 = 0.15

Table 15: Comparison of different systems.

Chapter 6: Conclusion and Further Work

6.1 Conclusion

A parallel crawler is implemented and many design issues related to a distributed environment are learnt. Almost all of the issues are resolved but there are some of them where there exists an opportunity for improvement or enhancement. Key problems faced are mentioned in chapter 4. Section 5.3 discusses such issues. These are also the issues when resolved; can enhance the performance and accuracy of the crawling task.

6.2 Further Work

1. Parsing of non-HTML files:

During the crawl, the crawler looks for *content-type* of the page to be crawled. It crawls pages with a *content-type* of *text/html* and ignores *PDF*s, *Javascript*, *CSS* and other text or application files. If one can come up with a parser for such documents, it can be plugged into the code and various documents can be crawled and information extracted from various documents. Right now, *extract_links* is the function to parse HTML documents. One can easily add functions to crawl and parse different types of pages.

2. Segregation of the polling function and processing of a fetched document:

The crawler chooses a server to fetch next URL to crawl. This depends on the configuration switch of the *courtesy pause* to be observed between two fetches from the same server. This is done to avoid overloading one particular server with multiple requests in a short time.

The current method selects a server and crawls a URL hosted on that server. But the URL can be unreachable or might have errors. This leads to a delay of period of time-out set for the *LWP::UserAgent* object. Currently, the time-out is set to 30 seconds and the *courtesy pause* is set to 10 seconds. Thus, the crawler can try to fetch just one where it could fetch three pages. Moreover, due to large amount of links on a page and checking each link with the database to see if it is seen before or not; some pages require more than a minute to be processed.

If the function of polling can be independent from the function of fetching documents, it would boost the crawler's performance significantly. This can be achieved by implementing the fetching function as threads. The crawling function was initiated as different threads making it hard to create threads for fetching a document. This is a limitation of not being able to distinguish between different levels of threads in the thread pool. Occasionally, a number of fetched pages take a long time - more than a minute; to be downloaded and processed. There can be a possibility of multiple threads downloading from the same server. If we have for example, five servers from which the crawler fetches pages, and if the userAgent object fetches pages simultaneously from all of the five servers, it would take a longer time to download the pages, there can be

numerous threads in their running state simultaneously and more threads being added at a gap of ten seconds. Moreover, the *userAgent* object is not implemented to obey the timeout and it runs in a blocking mode resulting in waiting by the main program while a page is being downloaded. This can be changed to implement a non-blocking *userAgent* and a different polling function which initiates fetching of a page from a server as soon as the program has waited for the number of seconds as indicated by the *courtesy pause* after a previous page download is complete.

3. Caching of DNS queries:

A DNS query to resolve the domain name happens for each URL. The crawler does not have the capability to save results of a DNS query. In one implementation, the domain names of the URLs were not resolved. This resulted in minor delay for resolving each URL but it adds up to significant delay for the entire crawl. In another implementation, a server's domain name was resolved only once, at the start of the crawl or when a URL was encountered from a different server. But the mapping was not stored. It would be good to have a module that stores DNS query results for servers and minimize the delay in time for resolving each URL. Instead of resolving all domain names dynamically, one can save previous results and periodically check for their updates. This way, one can track how many times a server's IP address was changed. The change in IP addresses of servers in the *.edu* domain is almost negligible but it can be significant for *.com* or *.org* domains where domains are maintained by service providers.

The new module can either be attached to the current system or the server field in the *c_url* tablecan be replaced by the IP addresses of the Web servers and mapping of IP addresses and domain names can be stored in a different table.

4. Context based sorting of URLs in the frontier:

The sorting of URLs is a dynamic function. The URLs in the frontier are sorted in the order of their frequency of discovery in already fetched documents. A rank is already provided to each URL and a URL with the highest rank is crawled first. A function which calculates rank according to the context of the link or its position from the seed page or some parameter based on link structure of the URLs and add it the current rank can classify more accurately how important a URL is and what its position is in the frontier queue.

5. Getting around a *Splash page*:

The first page, the index page for a domain is taken as a *seed* for that domain. Some domains have a flash video on the main page. Such pages are called *Splash pages*. Such a page does not have any URLs and the navigation starts inside the flash video with words “Skip Intro”. Sometimes, there is a HTML link but not always. A crawler cannot get to other pages in the domain and the domain is marked as crawled after crawling the index page.

There are two solutions for this problem. One, as mentioned above, one can create parsers which look for links inside flash videos which is a formidable task. This way, the crawler can follow the “Skip Intro” link and find other pages in the domain. This method

can also help in discovering all pages in a domain if the site is implemented in Flash. Two, one can query a search engine and get a few URLs from the domain. The crawler can start crawling the domain with these URLs and discover other pages.

References

[BCS 2002] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, *Ubicrawler: A scalable fully distributed web crawler*. In Proceedings of AusWeb02 - The Eighth Australian World Wide Web Conference, Queensland, Australia, 2002, http://citeseer.ist.psu.edu/boldi02ubicrawler.html

[BYC- 2002] Baeza-Yates and Castillo, WIRE Project 2002, http://www.cwr.cl/projects/WIRE/

[CGM 2002] J. Cho and H. Garcia-Molina, *Parallel crawlers*. In Proceedings of the Eleventh International World Wide Web Conference, 2002, pp. 124 - 135, http://oak.cs.ucla.edu/~cho/papers/cho-parallel.pdf.

[DKR 2002] Dill, S., Kumar, R., Mccurley, K. S., Rajagopalan, S., Sivakumar, D., and Tomkins, *A Self-similarity in the Web*. ACM Trans. Internet Technology, 2002, 2(3):205–223.

[FMN 2003] D. Fetterly, M. Manasse, M. Najork, and J. Wiener. *A large-scale study of the evolution of web pages*. In Proceedings of the twelfth international conference on World Wide Web, Budapest, Hungary, pages 669-678. ACM Press, 2003. http://delivery.acm.org/10.1145/780000/775246/p669-fetterly.pdf?key1=775246&key2=5281066411&coll=portal&dl=ACM&CFID=70588468&CFTOKEN=69016913.

[HNM 1999] A. Heydon and M. Najork, *Mercator: A scalable, extensible web crawler*. World Wide Web, vol. 2, no. 4, pp. 219 -229, 1999., http://citeseer.nj.nec.com/heydon99mercator.html

[JGM 2003] J Cho and H Garcia-Molina, *Effective Page Refresh Policies for Web Crawlers*, ACM Transactions on Database Systems, 2003

[JHE 2000] J. Cho and H. G. Molina, *The Evolution of the Web and Implications for an incremental Crawler*, In Proceedings of 26th International Conference on Very Large Databases (VLDB), *Septem*ber 2000.

[JHH 2006] J. Cho, H. G. Molina, T. Haveliwala, W. Lam, A. Paepcke, S. Raghavan and G. Wesley, *Stanford WebBase Components and Applications*, ACM Transactions on Internet Technology, 6(2): May 2006.

[JHL 1998] J Cho, H. G. Molina, Lawrence Page, *Efficient Crawling Through URL Ordering,* Computer Networks and ISDN Systems, 30(1-7):161-172, 1998.

[KGR 1993] Koster, M. Guidelines for robots writers. 1993, http://www.robotstxt.org/wc/guidelines.html.

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

[KRE 1996] Koster, M, *A standard for robot exclusion*. 1996, http://www.robotstxt.org/wc/exclusion.html.

[LKC 2004] B. T. Loo, S. Krishnamurthy, and O. Cooper, *Distributed Web Crawling over DHTs*. Technical Report UCB-CS-04-1305, UC Berkeley, 2004, http://www.eecs.berkeley.edu/Pubs/TechRpts/2004/CSD-04-1305.pdf

[NWB 2001] M. Najork and J. Wiener, *Breadth-first search crawling yields high-quality pages.* The Tenth International World Wide Web Conference*,* May 2-5, 2001, Hong Kong ACM 1-58113-348-0/01/0005, http://www10.org/cdrom/papers/208/.

[PSM 2004] O. Papapetrou and G. Samaras, *Minimizing the Network Distance in Distributed Web Crawling*. International Conference on Cooperative Information Systems, 2004, pp. 581-596, http://softsys.cs.uoi.gr/dbglobe/publications/coopis04.pdf

[SNC 2006] Sonali Patankar, *Needle Crawler: A Large Scale Crawler for University Domains*, Master's project, UCCS Computer Science Department, 2006.

[SSD 2002] V. Shkapenyuk and T. Suel, *Design and implementation of a high-performance distributed Web crawler*. In Proceedings of the 18th International Conference on Data Engineering (ICDE'02), San Jose, CA Feb. 26--March 1, pages 357 - 368, 2002, http://ieeexplore.ieee.org/iel5/7807/21451/00994750.pdf?isnumber=&arnumber=994750

[YDI 2006] Yi Zhang, *Design and Implementation of a Search Engine With the Cluster Rank Algorithm*, UCCS Computer Science Master's Thesis, 2006.