Needle Crawler  
A Large Scale Crawler for University Domains  

By  
Sonali Patankar  

A Project  
Submitted to the Faculty of Graduate School of the  
University of Colorado at Colorado Springs  

In Partial Fulfillment of the Requirements  
For the Degree of Master of Science  

Department of Computer Science  
Fall 2006  

______________________________  
Advisor: Dr. Jugal Kalita  

______________________________  
Committee Member: Dr. C. Edward Chow  

______________________________  
Committee Member: Dr. Marijke Augusteijn
Table of Contents

Abstract........................................................................................................................................ 4
Chapter 1. Introduction ...................................................................................................................... 5
  1.1 Problem Statement .................................................................................................................... 5
  1.2 Objectives ................................................................................................................................ 7
  1.3 Summary of Work ..................................................................................................................... 7
Chapter 2. Background Research ....................................................................................................... 9
  2.1 Issues Related to Crawling ........................................................................................................ 12
Chapter 3. Software Design Goals .................................................................................................... 14
Chapter 4. Implementation ............................................................................................................... 16
  4.1 Needle Crawler High Level Architecture ............................................................................... 16
  4.2 The Configuration File ............................................................................................................ 20
  4.3 Modules .................................................................................................................................. 21
  4.4 User Manual ............................................................................................................................ 22
  4.5 Incremental and final log .......................................................................................................... 23
  4.6 Database Design ....................................................................................................................... 24
  4.7 Restart Logic ............................................................................................................................ 29
Chapter 5. Problems faced and Solutions .......................................................................................... 30
Chapter 6. Results ............................................................................................................................. 37
  6.1 100K Crawl ............................................................................................................................ 37
  6.2 500 K Crawl (Aborted after 440 K) ....................................................................................... 40
  6.3 One Million Crawl ..................................................................................................................... 44
  6.4 Restart ..................................................................................................................................... 53
  6.5 Comparison with other published Crawler ............................................................................. 55

7. Conclusion .................................................................................................................................... 57
  7.1 Future Work .............................................................................................................................. 58
  7.2 Considerations for Large Crawlers ......................................................................................... 59

8. References ..................................................................................................................................... 63

Table 1: Time Spent by Activity for 100K Crawl ............................................................................. 37
Table 2: Cumulative Time Spent and Average for 400K Crawl ....................................................... 41
Table 3: Time Spent Distribution for 440K Crawl ......................................................................... 42
Table 4: Content Type Distribution for 440K ................................................................................ 43
Table 5: Return Code Distribution for 440K Crawl ....................................................................... 43
Table 6: Time Spent for 100000 URL intervals in 1 Million Crawl ............................................... 46
Table 7: Restart Performance .......................................................................................................... 53
Table 8: Comparison with published crawlers................................................................................. 55

Figure 1: Pie Chart for 100K Crawl Time Spent Distribution ......................................................... 38
Figure 2: Bar Chart for Cumulative Time Taken for 440K Crawl ................................................ 41
Figure 3: Pie Chart for Time Spent Distribution for 400K Crawl .................................................. 42
Figure 4: Content Type Distribution for 440K Crawl .................................................................... 43
<table>
<thead>
<tr>
<th>Needle Crawler</th>
<th>Master’s Project Report</th>
<th>UCCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 5: Pie Chart for Return Code Distribution for 440K Crawl</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>Figure 6: Content Type Distribution for 1 Million Crawl</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>Figure 7: Incremental Content Type Distribution (1 Million Crawl)</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>Figure 8: Return Code Distribution (1 Million Crawl)</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Figure 9: Bar chart for Text and total download for 100K Crawl Intervals</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>Figure 10: Cumulative and Interval specific time spent for 1 Million Crawl</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>Figure 11: Pie Chart for Time Spent by Activity (1 Million Crawl)</td>
<td>52</td>
<td></td>
</tr>
</tbody>
</table>
Abstract

We may learn a great many things thanks to the abundance of information on the World Wide Web. Due to this abundance of information we are given a new challenge, to find relevant information, in a reasonable amount of time. Search Engines were designed for this challenge earlier on. They are an effort to make information navigation and access easier. Due to the dynamism of the Web, crawling forms the back-bone of Web information retrieval applications.

This report is a detailed review of observations, experiences and lessons learnt, during design and development of the Needle Crawler. Needle is the name of a search engine for educational domains, being built at UCCS. The Needle Crawler is designed to fit into, the Needle Search Engine architecture. To address future needs of the Needle Search a restart feature has been added to make the crawler versatile. This design was developed and tested on a small Linux box, allowing it to achieve reasonable results. Using perl and mysql, Needle has been able to crawl 1 million pages in elapsed time of 739941.266 seconds and the average number of URLs processed was 1.78. However, it was designed so that on a faster machine with higher bandwidth, better results could be attained.
Chapter 1. Introduction

At UCCS, Jing Yang\textsuperscript{1} and Apparao Kodavanti\textsuperscript{2} worked on a separate text and image search engine effort. They each developed separate crawlers, with different architectures. The problems they faced proved valuable, in working on the Needle Crawler effort. Needle is meant to be a search engine for University domains. The Needle Crawler is a piece of the Needle Search Engine. The crawler will navigate the pages, download and extract information/links from them and prepare data for the later stages of search engine processing such as parsing, indexing and the front end. The Needle search engine allows easy configuration and modular enhancement of features, allowing this crawler to be seamlessly integrated into the current architecture.

1.1 Problem Statement

Creating a large-scale crawler for URLs on the Web is a time-consuming and complex task. Although Commercial search engines have efficient crawlers, the documentation is insufficient for research purposes due to the proprietary nature of the engines. When I started working on a large-scale crawler, I faced many problems. The following are problems that had to be addressed during my endeavor to build a crawler for the Needle search engine.

- A lot of duplicate content exists on the Internet. The same files are hosted on multiple sites. Downloading and processing the same content can cause the crawler to work extra without any gain. Using a message digest of the content this extra work can be avoided.

\textsuperscript{1} http://cs.uccs.edu/~jyang/search3.php
\textsuperscript{2} http://dirac.uccs.edu/~image/search.html
In crawler terminology this is referred to as **Content Seen**. It also helps isolate duplicate content on mirror sites.

- Many different Websites have links to the same sites on them. Popular links are referenced in many sites. Once a URL has been downloaded and processed, it does not need to be downloaded or processed again if it appears again in another page. This is referred to as **URL seen** check.

- A Web crawler’s job is to download documents from Web Servers in a systematic manner. A crawler can be considered a browser that sends many requests in very short amount of time. This can cause a large load on the Web server(s) fulfilling the request. The requests must be spaced out at a fixed interval to avoid overloading Web Servers.

- Most of the information on internet is publicly accessible. The crawler can come across URLs that the Web server administrator would prefer were not crawled. The convention is to provide the access rules in a file called **robots.txt**, which is stored at the sub-domain level. This file contains rules the crawler should obey. It should not access files disallowed by the **robots.txt** file.

- A crawler is the search engine’s interface to external world for collecting valuable information. A crawler makes a part of the information it collects such as the type of file and the URL Link structure available to the search engine. A crawler should capture and record this information so that the next stages would not have to repeat a part of crawling.

- Crawling is a time consuming and complex process. The amount of work that a crawler has to do means that it must not repeat work that has already been done. A crawler should be able to continue work from where it was last time when it was terminated. So, an efficient Restart mechanism must be provided for the crawler to provide consistent performance.
• Information available on the Internet is stored in many different kinds of files. A
crawler should be extensible so that future enhancements can be made. New pieces may
be needed for reading new kinds of files and converting them to text if necessary for link
extraction and processing.

1.2 Objectives

Due to the past efforts at UCCS and the problems they faced and were not able to
overcome, the first objective was to overcome these problems. Another major objective
was to enhance the Needle search engine. The Needle search engine is meant to be
scalable, so that it can handle a large number of URLs, each of which must be crawled.
The crawler chosen should be polite to minimize the imposition on webmasters. Since the
World Wide Web has a lot of redundant data, it also should avoid redundancy. The Web
is always evolving, and the Needle Crawler must make it easy to adapt to future
enhancements.

1.3 Summary of Work

Jing's and Apparao's work revealed some potential challenges. With the help of research
papers, Mercator’s design appealed to me, since it was trying to achieve objectives
similar to mine. Using Mercator [1] as a starting point I designed the Needle Crawler to
meet the objectives. Design was just the beginning. As I delved into coding and testing,
things got a lot more challenging. I came across several new issues, some that I was
prepared for, and some that I was not. Apart from a number of small crawls, Needle was
used to crawl 1 million pages and the results were gathered to analyze the performance.
Comparison with published results and previous efforts at UCCS and elsewhere was also made.
Chapter 2. Background Research

The Internet has grown rapidly since its inception in December 1969. The Web is a directed graph where a Web page is a node and a directed link from one node to another is a URL connecting those two pages. Search engines are the key to finding specific information on the vast expanse of the WWW. Without sophisticated search engines, it would be virtually impossible to locate anything on the Web without knowing a specific URL.

While there are a great variety of search engines like Google\textsuperscript{3}, AltaVista\textsuperscript{4}, Yahoo\textsuperscript{5}, they all share the same basic architecture. A search engine consists of three components a robot (crawler), an indexer and user interface.

The robot periodically traverses the WWW and does the following: [5]

(a) Fetches a page

(b) Parses it to extract all linked URLs

(c) For all the URLs not seen before, repeats (a)–(c)

The Parser or the Indexer extracts all the words from each page and records the URL where each word occurred. Page Ranking is performed to assign levels of relevance for the URLs [2]. A User Interface assists in specifying search terms and preferences. Search engines differ primarily in how content is extracted, how much of the WWW they search

\textsuperscript{3} http://www.google.com

\textsuperscript{4} http://www.altavista.com

\textsuperscript{5} http://www.yahoo.com
and how user keywords are matched and ranked. Web crawlers, also known as spiders, robots or wanderers, are the software programs that automatically traverse the Web.

Web crawlers are almost as old as the Web itself. The first crawler, Matthew Gray’s Wanderer, was written in the spring of 1993, for collection of data to measure the growth of the Web [3].

Generally, the crawler gets a list of URL's to visit and process. The crawler doesn't rank the pages, it only goes out and gets copies, which it stores for the search engine to later index and rank according to various criteria. Search engines use crawlers to find what's on the Web; then they construct an index of the pages that were found.

Three prominent algorithmic choices for crawling are [7] Breadth first traversal, Depth first traversal, or Best first traversal. For topic-based crawling breadth first traversal is said to yield better results [3]. The best first strategy refers to the ordering of the URLs based on some priority scheme, for example, using Page Rank as described by Google [2], or based on the number of Hyperlinks coming out or pointing to the page. Depth First Traversal keeps going to the first URL, the page found by it, and the first URL on it and so on, until there are no links, then comes back to the second URL. The data structure used in this case is a stack.

The estimated cost of an entire crawl of the World Wide Web was about US $1.5 Million [11] in 2000, considering just the network bandwidth necessary to download the pages. As a result, it becomes very important to use the network resources efficiently to maximize the crawler throughput and avoid wasting the allocated bandwidth.
To make crawling efficient, caching and cache size are two important factors [5]. A large cache brings little additional benefit. It is difficult to determine the best crawling strategy because of the volume and unavailability of crawler logs from commercial search engines. A cache of 50000 URL entries could achieve hit rate of almost 80%. All caching strategies are equally good but CLOCK [9] or RANDOM [5] is recommended.

A crawler interacts with thousands of Web servers the fact that they are beyond the control of the system, makes a crawler fragile. Google improved, by utilizing resources to their optimum, such as CPU, memory, disk and bandwidth [2]. A search engine is a valuable research tool for a wide range of applications.

Crawlers are an important component of Web search engines [4], and most search engine companies treat their crawling strategy as a trade secret. Therefore, they have not described the crawlers in the literature. In most searches, useful documents can be found within a depth of 4 or 5 links from a site. The depth a crawler needs to dig to is higher in case of blog sites. Dynamically generated Web sites are the future and guidelines are needed for practically crawling the infinite web [4]. The growing size of the Web requires a crawler to be scalable. Being efficient at all levels is of paramount importance [7].

Mercator is an extensible and scalable crawler written in Java. Its scalable design made it easy to configure the crawler for memory varying from 128 MB to 2 GB. Extensibility is achieved by making use of protocol and processing modules [1]. Mercator is currently used as the Web crawler for AltaVista’s American and European search sites [3]. Crawler architectures need special consideration for the environment crawlers work in. DNS
Resolution is one possible area of improvement. Smart crawlers take into account current response from Web servers to decide order of a crawl [8].

At UCCS, Jing Jacobs\(^6\) and Apparao Kodavanti\(^7\), developed 2 separate search engines for text and image respectively. It was a great initial effort, and there is a lot of room for improvement. “Needle” is the name of the integrated search engine effort (for both text and image) currently in progress under advice of Dr Kalita to create a scalable, extensible Search Engine.

### 2.1 Issues Related to Crawling

Although a Web Crawler's function is simple, it must be programmed to handle many challenges. Running a Web crawler is a challenging task. There are tricky performance and reliability issues and even more importantly, there are social issues. Crawling is the most fragile application since it involves interacting with hundreds of thousands of Web servers and various name servers which are all beyond the control of the system. Most of the issues are related to the characteristics of the Web, independent of the Web crawler architecture chosen.

The two most important guidelines given by Koster [10] are (1) that a crawler must identify itself, including an e-mail address for contact, or some Web site administrators will send complaints to the listed owner of the entire originating network segment and (2) a crawler must wait between repeated accesses to the same Web site.

\(^6\) [http://cs.uccs.edu/~jyang/search3.php](http://cs.uccs.edu/~jyang/search3.php)

\(^7\) [http://dirac.uccs.edu/~image/search.html](http://dirac.uccs.edu/~image/search.html)
Repeated access to a Web page can trigger some alarms on the Web server, and complaints from its administrator. Duplicates in the queue must be eliminated to prevent it from fetching the same page over and over.

A major performance problem is DNS lookup. Each crawler can maintain its own DNS cache so it does not need to do a DNS lookup before crawling each document.

A crawler must determine how often to revisit a page. On one hand, it's a waste of resources to re-index an unchanged page. On the other hand, it wants to re-index changed pages to deliver up-to-date results.

The prevalence of mirrored content on the Web is very high. It is estimated in over 30% of cases, the content retrieved is duplicate content. So, it is important find pages without duplicate content [11].

One of the most challenging aspects of Web crawling is how to download pages from multiple sources in a stream of data that is as uniform as possible. This is a problem because Web server response times vary considerably.

Web server up-time cannot be taken for granted. It is common that Web servers are down for long periods of time, even days or weeks, and re-appear later. This is why sometimes “dead pages” are called “comatose pages” [12].
Chapter 3. Software Design Goals

Below are the software design goals for the Needle Crawler.

Modularity - As the Crawler requirements grow, the code is likely to increase in size and complexity. The Needle code should be managed in modules or subroutines. Each module should do work assigned to it, and interface with the calling module. Modularity provides important benefits such as reuse of code, and easy troubleshooting.

Configurability – The crawler crawls the Web. The Web has a variety of documents, both with errors and without. There are lots of variables that go into the design of the Crawler, for what it should and should not do. The Needle crawler should be configurable, so that all these parameters can be provided from a XML file.

Extensibility - By definition, extensibility is the ability to enhance current operation without needing to make architecture level changes. Since the challenges and requirements faced by the Needle crawler can change frequently, it needs to be extensible. This flexibility will allow tailoring the software to specific requirements.

Scalability – Scalability is an application’s ability to cope up with heavy load, and provide consistent performance. In reference to the Needle Crawler, it should process a consistent number of URLs over a period of time. Even though the basic operation of the crawler is to download URL and extract URLs, the number of URLs, the number of links in them can be different for every page. Two main pieces contribute to crawler performance, average download speed and average URL processing speed. Not much can
be done to improve average download speed, though it has a major effect on the final
crawler average. URL processing speed varies due to both number of URLs processed
and the amount of processing needed for each URL. The ratio of URLs processed over
elapsed time provides better insight on how this piece scales up. Evaluating scalability is
looking at both of download speed and processing ratio to see how consistent they were.

**Politeness** - `Robots.txt` is a file that is meant for instructing crawlers what files may
or may not be indexed. The Needle Crawler should follow the rules specified in
`robots.txt`. It should also ensure that the same subdomain is not visited very
frequently. The current time between visits is 10 seconds and is configurable.

**Redundancy Removal** – The Internet is full of information, but unfortunately, there is
plenty of duplicate information. The same URLs are referred to in multiple places, and
sometimes the same content is available under different names. The Needle crawler needs
to recognize redundancy and process accordingly so that processing time is spent on
unique content.
Chapter 4. Implementation

Among the papers I reviewed, Mercator [1] presents a relatively detailed study. My initial design was based on that of Mercator. Even though the basic crawler operation seems trivial, the volume of URLs and variety of situations presented their own challenges. I modified the design to provide desired features. The next diagram shows high level Needle Crawler Architecture.

4.1 Needle Crawler High Level Architecture
Seed URLs are added to the queue. Scheduler will decide which URL is selected from queue. It is passed to Downloader that will make the request for the page and download the page and stores it in storage area (hard disk). Then the downloaded document is passed to Process URL. Process URL consist of extracting links, checking for URL seen, Filter check, Allowed domain Check, robots.txt etc. once the URL passes all the checks it is added to the queue as a candidate for crawling.

The following page has a basic block diagram of the Needle Crawler Operation and provides more details.
Needle Crawler
Process Seed URLs and add them to the queue

Extract the next url to be crawled and submit for download

MediaType Used for Reference
Read configuration file and prepare for crawl

URLLinkStructure
Update Link Structure

RobotRules
If Allowed add to URLs to be Crawled queue and continue to extract

Domain Table

Download Success ?
Retries exceeded ?

Save the file
Disk for Saving Downloaded files
Update Crawler Table

URL
Get RobotRules and Add to RobotRules Table

Content Seen
Add the New Domain to DomainTable

Check Content Type
Does this URL pass the allowed domain filter ?

ReasonCode
Does this URL pass custom

UCCS
Is this a text file ?

Crawler
Extract Links and do the following process for all links

Is this URL already seen ?

Does this URL pass allowed domains

Sonali Patankar
The block diagram given above shows the general operation of the Needle Crawler. The Crawler starts with loading the parameters from the config.xml file. It processes the seed URLs and loads them in appropriate subdomain queues. Then it decides which should be the next URL to be crawled.

After the URL is chosen, it is requested from the server. If the download is successful, it is saved using the url_id. If the download is not successful, the Needle Crawler will check if it should retry the URL. If so it will put the URL back in the appropriate queue. If the download succeeds, the crawler checks if the content has been seen before. If the content has been seen before, it should not process this since this is duplicate content. In this duplicate content case, it should also update the ReasonCode table to reflect this. If this is new content, then check the type of the content. In all cases, the Crawler table is updated. The process continues by checking if the type of content is text, the Crawler will process the downloaded file to extract the links and process them. While processing links, several checks are made to decide if the links should be crawled. The URLSeen check is done to see if this URL has already been seen. The Filter check is done to see if the extension of the file should be crawled. The Allowed domains check will check if the domain in the link should be crawled. If the domain is allowed, DomainTable will be checked to see if the domain is new. If the domain is new, the robots.txt file is checked to see if there are new robot rules to be captured. Then the robot rules will be checked to see if this URL should be crawled. Custom filter checks are done to see if the URL is longer than 255, if it is a potential crawler trap, or if it has in page anchor tags.
4.2 The Configuration File

One of the goals of the Needle Crawler is to make it configurable. A sample configuration file is shown below.

```xml
<config>
  <DB dbname="MySQL">
    <driver>mysql</driver>
    <host>128.198.144.19</host>
    <dbname>dbp_sonali</dbname>
    <user>sonali</user>
    <passwd>sonali74</passwd>
  </DB>
  <RestartCrawler>Y</RestartCrawler>
  <SeedURL>http://www.cusys.edu</SeedURL>
  <SeedURL>http://www.uccs.edu</SeedURL>
  <SeedURL>http://www.colorado.edu</SeedURL>
  <SeedURL>http://www.cudenver.edu</SeedURL>
  <SeedURL>http://www.ucdhsc.edu</SeedURL>
  <IncludeDomain>cusys.edu</IncludeDomain>
  <IncludeDomain>ucss.edu</IncludeDomain>
  <IncludeDomain>colorado.edu</IncludeDomain>
  <IncludeDomain>cudenver.edu</IncludeDomain>
  <IncludeDomain>ucdhsc.edu</IncludeDomain>
  <ExcludeDomain>search.colorado.edu</ExcludeDomain>
  <ExcludeDomain>adsync.colorado.edu</ExcludeDomain>
  <InvalidURLBegin>javascript:InvalidURLBegin</InvalidURLBegin>
  <InvalidURLBegin>mailto:InvalidURLBegin</InvalidURLBegin>
  <TimeOutSeconds>10</TimeOutSeconds>
  <URLLimit note="max number of URL to crawl">100000</URLLimit>
  <ProtocolSupported>http</ProtocolSupported>
  <ProtocolSupported>https</ProtocolSupported>
  <Text>
    <SupportedType>html</SupportedType>
    <SupportedType>htm</SupportedType>
    <SupportedType>shtml</SupportedType>
    <SupportedType>php</SupportedType>
    <SupportedType>txt</SupportedType>
  </Text>
  <Image>
    <SupportedType>jpg</SupportedType>
    <SupportedType>gif</SupportedType>
    <SupportedType>jpeg</SupportedType>
    <SupportedType>png</SupportedType>
  </Image>
</config>
```

Below I discuss th XML tags used in the configuration file.
<dbname> : The name of the database used by Crawler.

<host> : The IP address of the machine name where the database exists.

<passwd> : The database password.

<RestartCrawler> : Yes (Y) or No (N) switch to indicate if the crawler should restart from where it left off, or start fresh.

<URLLimit> : The maximum number of URLs the crawler should crawl for this run.

<SeedURL> : The list of seed URLs to be used by the crawler.

<IncludeDomain> : The list of domains the crawler should consider valid.

<ExcludeDomain> : The list of domains that should be disallowed.

<ProtocolSupported> : The list of protocols the crawler should consider valid.

<SupportedType> : The list of file extensions allowed.

Instead of using a text input file, the Needle crawler uses a XML input configuration file. This makes it easier to edit, which in turn makes it easy to make program changes if more parameters are added.

4.3 Modules

The Needle Crawler is written using Perl. Keeping in mind, the goal of modularity, subroutines were created in order to maintain a fair balance between distribution of work and parameter passing overhead.
4.4 User Manual

To get the crawler working correctly, the parameters in the configuration file need to be updated. The directories used for saving text and image files should be created if they don’t already exist.

To run the crawler in the background on a Linux machine the following command can be executed.

> perl crawlernov08.pl &

The database and tables used by the crawler need to be created. The access privileges for the database and table need to be appropriately specified for the user running the crawler. In case of large crawls, it is more useful to run it in pieces. The crawler writes a log into file crawlerrun.log every 500 URLs crawled.

The crawler program is stored on an unnamed machine on Dr Kalita’s subnet of machines. The IP addresss of the machine is: 128.198.144.17 and all files exist under sonali/prodrun/

configuration file location: sonali/prodrun/config.xml, This file is used to provide parameters like if we want to restart or not, domains we want to include and domains we want to exclude etc. It also has information about the database used by the crawler.

crawler program: sonali/prodrun/crawlernov08.pl
Incremental logfile: `sonali/prodrun/crawlerrun.log`. It has log after every 500 urls crawled

logfile: `sonali/prodrun/crawler.log` (this file contains final cumulative log for the current crawl)

All pages that are requested are downloaded and saved under either `downloadip/image` or `downloadip/text` depending upon if it is image file or text file.

### 4.5 Incremental and final log

Final Log Example (`crawler.log`)

time spent in restart is: 145.273861
total time spent in domain was: 1341.83602200009
total time spent in domainipadd was: 8.8732269999927
total time spent in url filter was: 54.3641779999922
total time spent in robotrule was: 0.014248
total time spent in allowed domain was: 69.6137740000156
total time spent in checkrobotrule was: 879.698188999939
total time spent in addseed:
total time spent in download was: 38287.3151900003
total time spent in CS-download was: 2155.920978
total time spent in check mimmtype was: 4.46703299999831
total time spent in urlhash was: 156.829462992683

*** extract url calculation

total time spent in extracturl was: 36747.5136739998
total time spent in extract links was: 13015.036461
total time in extr1 was: 13038.1983790001
total time spent in extr2 was: 3266.01940199979

total time spent in extr3 was: 4548.422728017

total time spent in extr4 was: 9109.60592200829

total time spent in extr5 was: 5950.64450501003

total time spent in extr6 was: 4.1996089999992

final total is: 35917.0905450352

the difference is: 830.423128964627

time spent before downloadurl starts executing: 150.258224
time spent in complete downloadurl: 100618.212114
time spent in processing download failurls: 2.093887
time spent in processing download succurls: 38645.30881
Time spent in waiting for Web servers to respond : 1547.973026
total time spent in content hash was: 102.120332000002
total time spent in addtoqueue was: 844.3188149999784
total time spent in extractfromqueue was: 20807.0240040001
total delay time is: 17592.7676525116
total valid urls extracted: 313001
total urls extracted: 5211562
number of urls crawled are: 100000
elapse time: 100768.470738
end time 16 20 28

time missing1 = 972.444631000007
time missing2 = 313.62667399997

Incremental Log Example

145.273861 , 1336.4192990001 , 8.83477299999914 , 54.1135069999922 ,
0.014248 , 69.3088310000154 , 876.234226999939 , , 38104.5221600004 ,
2144.777792 , 4.44454799999834 , 156.236171992759 , 36594.2646369998 ,
12954.009692 , 12977.0667100001 , 3251.71972199978 , 4532.08638701681 ,
9073.40754500841 , 5928.42621300994 , 4.17938299999993 ,
35766.885960035 , 101.798332000002 , 840.710438999788 ,
20675.7273330001 , 17477.6704494953 , 311640 , , 5192420 , 99500 ,
100286.187201 , 16 , 12 , 26 , 1547.973026 , 2.093887 , 38483.5807639999
, 967.751406000008 , 312.06616899997

The incremental log can be easily loaded into Microsoft Excel to analyze the performance of various modules within the crawler. At the end of the crawl, the crawler writes a file crawler.log with complete statistics for the entire crawl including the time spent in different activities in the crawl.

### 4.6 Database Design

The Needle search engine table design uses a modular approach and attempts to save only the data absolutely necessary. When the crawler is in operation, a lot of data is being captured. It is important to avoid redundant data and make table design and indexes suitable for faster operation of the crawler.

**Crawler**

The Crawler table will keep details about every URL crawled.
### DomainTable

The DomainTable will maintain details about every new subdomain encountered.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Default</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain_id</td>
<td>int(10)</td>
<td>No</td>
<td></td>
<td>Unique auto increment identifier for domain</td>
</tr>
<tr>
<td>domain_name</td>
<td>varchar(255)</td>
<td>No</td>
<td></td>
<td>Domain name</td>
</tr>
<tr>
<td>IP_address</td>
<td>varchar(255)</td>
<td>No</td>
<td></td>
<td>IP address when available.</td>
</tr>
<tr>
<td>time_of_req</td>
<td>decimal(16,6)</td>
<td>Yes</td>
<td>NULL</td>
<td>Last time a request is made for this domain</td>
</tr>
<tr>
<td>qsize</td>
<td>int(10)</td>
<td>No</td>
<td>0</td>
<td>Number of URLs in the queue for this domain</td>
</tr>
</tbody>
</table>
Switch for blocking the domain. If set to Y, the domain can be blocked from crawling while the crawler is running.

Number of URLss crawled from this domain

**MediaType**

The **MediaType** is a reference table to associate an id with detailed media type.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Default</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>doc_type_id</td>
<td>tinyint(3)</td>
<td>No</td>
<td></td>
<td>Unique auto increment identifier</td>
</tr>
<tr>
<td>doc_type</td>
<td>varchar(20)</td>
<td>No</td>
<td></td>
<td>Description for doc type such as text or image</td>
</tr>
</tbody>
</table>

**RobotRules**

The **RobotRules** Table will store the robot rules for every sub domain encountered.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Default</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>robot_rule_id</td>
<td>int(10)</td>
<td>No</td>
<td></td>
<td>Unique Auto Increment identifier for robot rule</td>
</tr>
<tr>
<td>domain_id</td>
<td>int(10)</td>
<td>No</td>
<td>0</td>
<td>Domain id for the robot rule</td>
</tr>
<tr>
<td>Permissions</td>
<td>char(1)</td>
<td>No</td>
<td></td>
<td>A or D for Allow or Disallow</td>
</tr>
<tr>
<td>directory</td>
<td>varchar(255)</td>
<td>Yes</td>
<td>NULL</td>
<td>The directory that is being allowed or disallowed D in permission and * in directory would mean nothing is allowed and A * would mean everything</td>
</tr>
</tbody>
</table>
ReasonCode

The ReasonCode table will hold counters for the various errors URLs have, such as filter failed, domain failed etc.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Default</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>reason_id</td>
<td>Tinyint(3)</td>
<td>No</td>
<td></td>
<td>Unique Auto Increment Identifier</td>
</tr>
<tr>
<td>reason_code</td>
<td>Char(2)</td>
<td>No</td>
<td></td>
<td>Two byte reason code identifying the type of error</td>
</tr>
<tr>
<td>counter</td>
<td>int(11)</td>
<td>No</td>
<td>0</td>
<td>Number of occurrences</td>
</tr>
<tr>
<td>reas_desc</td>
<td>varchar(50)</td>
<td>No</td>
<td></td>
<td>Detail Description of the reason code</td>
</tr>
</tbody>
</table>

URL

The URL Table will store the details about the URL.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Default</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>url_id</td>
<td>int(10)</td>
<td>No</td>
<td></td>
<td>Unique Auto increment identifier for URL</td>
</tr>
<tr>
<td>url</td>
<td>varchar(255)</td>
<td>No</td>
<td></td>
<td>url text</td>
</tr>
<tr>
<td>doc_type_id</td>
<td>tinyint(3)</td>
<td>No</td>
<td>0</td>
<td>Type of the URL such as , text, image or other</td>
</tr>
<tr>
<td>Container_url</td>
<td>int(10)</td>
<td>Yes</td>
<td>NULL</td>
<td>The url_id of the container URL</td>
</tr>
</tbody>
</table>
### Table 1: Field Descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Default</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>cluster_id</td>
<td>int(10)</td>
<td>Yes</td>
<td>NULL</td>
<td>Used for page ranking</td>
</tr>
<tr>
<td>url_hash</td>
<td>varchar(32)</td>
<td>No</td>
<td>NULL</td>
<td>32 byte URL hash</td>
</tr>
<tr>
<td>retry_attempt</td>
<td>int(10)</td>
<td>Yes</td>
<td>NULL</td>
<td>Number of retry attempts</td>
</tr>
<tr>
<td>crawled</td>
<td>char(1)</td>
<td>No</td>
<td>NULL</td>
<td>Y, N or E switch to signify if it has been crawled, not crawled or an error</td>
</tr>
<tr>
<td>err_detail</td>
<td>char(2)</td>
<td>No</td>
<td>NULL</td>
<td>More detail of the error if crawled = E</td>
</tr>
<tr>
<td>udomain_id</td>
<td>int(11)</td>
<td>No</td>
<td>0</td>
<td>Domain id for the URL</td>
</tr>
</tbody>
</table>

### URLLinkStructure

The **URLLinkStructure** table stores the link structure seen as the pages are crawled.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Default</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>link_id</td>
<td>int(10)</td>
<td>No</td>
<td>NULL</td>
<td>Unique auto increment identifier for the link id</td>
</tr>
<tr>
<td>from_url_id</td>
<td>int(10)</td>
<td>No</td>
<td>0</td>
<td>From url id</td>
</tr>
<tr>
<td>to_url_id</td>
<td>int(10)</td>
<td>No</td>
<td>0</td>
<td>To URL id</td>
</tr>
<tr>
<td>anchor_text</td>
<td>varchar(100)</td>
<td>Yes</td>
<td>NULL</td>
<td>Anchor text for the link</td>
</tr>
<tr>
<td>update_date</td>
<td>date</td>
<td>No</td>
<td>0000-00-00</td>
<td>Update date</td>
</tr>
</tbody>
</table>
4.7 Restart Logic

In the initial restart implementation, I kept track of all the URLs processed whether failed or successful. They were kept in separate tables and to find out the ones to be crawled in case of a restart, a join was needed. In the case of a small number of rows, the restart worked, but as stated in Results section (Chapter 6. Results) this approach failed to give results in case of large crawls.

In an attempt to rectify this issue, the table design was changed to make the information needed for restart available in one table. This made the process of restart a simple select.
Chapter 5. Problems faced and Solutions

During the development of the Needle Crawler, I came across some unique challenges. Some situations did not seem like problems, but ignoring them could degrade the Crawler performance. Other problems encountered included dealing with differences in how the response differed in case of different Web servers. In this section, I discuss how the problems were recognized and overcome.

Crawler Traps

I discovered a couple of unintentional crawler traps, the typical example being radio.uccs.edu. This site has a calendar of events that has events until 2022 but because of no explicit solution the crawler attempted to crawl all these URLs.

Crawler traps are always very hard to find. Most unintentional traps we faced are calendars. While dynamic Websites can generate endless dynamic pages, the crawler needs to be intelligent to decide what pages are important and what pages are not. On investigating the calendar issue further, URLs I found use the word calendar and webcal as part of URL. I decided to use this information to my advantage and also gave consideration to the time window we want to focus on. The time window I chose was 1 year before and one year after the current date. This helped me avoid the crawler trap. One more change I made was to keep track of how many URLs are being crawled for every domain. This would help to identify crawler traps which may not have otherwise been discovered. To solve the crawler trap the problem, in the Custom filter I am checking if it has the word calendar or webcal in it.
**MimeType/Content Type**

By using HTML::Tokeparser\(^8\) links are extracted. Links are of type image or text. To recognize the correct media type, I tried using LWP::MediaTypes package. This package has methods that guess media types but the problem with this was that if it cannot recognize the media type, it returns application/octet-stream as the type. It recognizes .txt, .html, .gif, and .jpg, but does not recognize .shtml, .htm, .png, and .php. So I added the mime types that were not included to include all mime types for my crawler. But this package doesn’t allow adding the new mime types even though it says so. Hence, I decided to get the mime type by using the content type returned at the time of download. If the content type was other than text or image then the default content type of 3 would remain.

**Timeout**

For accessing and downloading URLs, I used LWP::UserAgent\(^9\). I came across a peculiar problem when the crawler accessed domains that would not respond. I had the timeout set for the UserAgent package, so that it aborts the request if the web server did not respond in 10 seconds. But some Web servers took 3 minutes and 10 seconds to timeout in my experience. When I searched for more information, I found that the 10 second time provided in LWP gets its effect only after the connection is established. Some servers took 3 minutes, and still the connection could not be established and the crawler waited for 10 seconds as per timeout. While there seems to be no way to get the web server to respond to us quickly, I decided to use this observation to benefit the crawler. I created a column in the DomainTable to signify problematic domains. When a domain takes

---

\(^8\) HTML::TokeParser(CPAN package) allows processing of a html document by splitting it into tokens. Each token can be processed to extract links and associated information.

\(^9\) LWP::UserAgent (CPAN package) is a web user agent class. It facilitates retrieval of documents from Web servers.
more than 100 seconds to respond, I mark the domain blocked so that in the URL selection, this domain will be ignored for the time being. This helped me improve Crawler performance significantly.

**In Page Anchor Tags**

While crawling, I came across a problematic situation. There was far too much content-seen recorded, so I looked at URL table and found that we had lot of in page anchor tags (over 300 in page anchor tags), e.g. (http://cs.uccs.edu/%7Ecs301/mysql/manual.html)

On this site we have many links on a page which are ‘#’ links (mainly for navigation). While processing the links initially the crawler thought that each of these links was unique, and hence they were finding their way into the queue. But upon downloading, the content had already been seen.

To solve this problem, whenever the hash (# sign) occurs in a URL, the part before the hash is checked to see if the URL is seen before. If seen, we consider it as an In Page Anchor Tag error. If it is not seen, we insert it without ‘#’ and crawl it so next time if it comes we will find it.

**Finding URLs that created problems for the crawler**

Some URLs caused the crawler to crash. In the beginning, it was difficult to discover, which URL may have caused the crawler to go down. To obtain this information reliably and efficiently, I added a column to the URL table called crawled. When the URL is considered for download the crawled field will be set to C. After link extraction and processing the crawled field will be set to Y.

This helped me to identify the URL the crawler was processing when it went down.
**Intermediate Logs**

In a long crawl, capturing what is going can be tedious when there are problems. It can be difficult to tell based on table statuses alone when the performance is worse than expected. In order to solve this we can log some information. All information about a URL cannot be written out because of two problems. One is that it consumes quite a bit of operational time and the other is that it can produce a lot of data that is difficult to analyze. To get more insight into the process, I only write logs after 500 URLs are processed, telling me how much time was spent for those and what is the time distribution. This let me look at where the time taken was growing, and validating if there is a legitimate reason for the same.

**Restart Issues**

My initial restart design joined the URL table, Crawler Table and Failed URL table. I used this combination to find which URLs were not crawled and not in the Failed Table. For the small runs, the restart was tested, and performed reasonably well. But after a 100K crawl, when I attempted to restart, the join became overwhelming, and the process hung for a long time. For a 100K crawl the approximate number of rows in URL were little under a million, 100 thousand rows in Crawler and about 2 million rows in FailedURLs. To do an efficient restart I used the column in URL table that now told me, what was not crawled, so it made it a simple select rather than a join. With a simple select, the program now needs to look at 1 byte field to decide whether this should be considered while doing restart. This reduces the complexity by selecting from one table rather than joining three tables.
Space

In one of the test crawls, for 100K URLs, the crawl worked, but since we were saving all the files to disk, the machine dirac started running out of space and nearly crashed. Since I anticipated crawling more than 100K, I changed the program to delete the file after it has been processed. This helped me perform larger crawls so that I could test my crawler more completely. Later on, a machine was provided that had a larger hard disk and then the code was changed again, to retain the files.

I also created two separate directories for text and image files so that it was easy to identify which one was which, and it was less overwhelming than storing everything under one directory.

Long URLs

I had a problem where the crawler was trying to download a URL and the crawler could not find it in the URL table. When the URL was put in queue, it was full length, but the table was expected to only store 255 bytes of URL. These URLs were longer than 255 bytes. Research suggested that URLs longer than 255 are more likely when a lot of parameters are being passed to a dynamic page. There is a potential Crawler Trap as well in here. I decided to mark URLs longer than 255 bytes as error, to solve this issue. Another consideration here was that a URL is stored in a varchar of size 255. If it were stored in TEXT format it would not be as efficient.

DNS Issues

When a fetch URL request is made, a call to DNS is made to find its IP address. This can put substantial burden on DNS servers. To lower the load and improve the performance, I cache the IP addresses of the domains when I first see them, and then use the cached IP
addresses to request URLs from that domain in the future. This reduced the amount of processing and waiting for the DNS server to respond. Waiting for the DNS server to respond can become overwhelming, due to number of requests being made.

In an ideal world every Website would have returned an IP address. But I came across domains where they would not return the IP address or when the IP address is shared by more than one domain. In these cases, storing the IP created more confusion than benefit. So I altered the algorithm further, to first get an IP address for a URL and then check, if we perform a reverse-look up, or lookup up the domain name using the IP address, if it returns the same domain name. To decide whether to use IP address for such sites or not is based on the reverse lookup.

**Debugging**

In the initial phase of the crawler development, the focus was more on feature development than testing and performance. After most features were developed, I made a crawl for 100K URLs. I noticed during this crawl that the extracting links from the page and processing them was taking more than double the time of download. Then, I tried to figure out what part was taking more time in extracting links and found quite a bit of code that was being executed when it was not really necessary. The code was processing results which were not used in future processing. Switches were introduced to avoid this processing. After fixing that we were able to cut down the extracturl time significantly.

While working with the UCCS domain I noticed that after crawling 30k+ URLs only one subdomain remains that has URLs to be crawled. With due respect to politeness requirements imposed on the crawler, the crawler needs to wait before next request is made, so the crawler slows down to follow the rules. This problem will automatically go away if there is more than one domain to pull URLs from.
Multithreading Consideration

I noticed that my `extracturl` subroutine was taking the most time of the entire crawling process. So I decided to make only that subroutine multithreaded. While the documents are downloaded sequentially, it creates a new thread that will extract links and perform filtering, i.e., checking prefixes and suffixes, checking if the domain is allowed or not, retrieving the robots rules and checking those robots rules to see if crawler is allowed to crawl that url. This new thread will save on the elapsed time and it will also be able to use available bandwidth more efficiently. But, since my crawler is running on a single machine, all the threads would compete with each other along with existing processes, and would not necessarily improve the performance, as I expected. So I continued to improve on the single threaded crawler.
Chapter 6. Results

Crawling is a time consuming operation. To be able to see how efficient the crawler is, it is important to keep track of time spent in various activities. It is also important to analyze the results to see where an improvement is possible, or if we have a major problem. While writing the crawler program, I tested it several times for many small crawls. The purpose of this testing was to see if the written functionality works. So the focus was on getting the crawler working rather than on the performance. As the features got implemented, the focus shifted to volume processing. Writing just the features themselves presents its own set of problems, while dealing with volume processing presents an entirely different set of problems.

6.1 100K Crawl

One of the first major crawls I did was for 100,000 URLs. This crawl was done using only www.uccs.edu as the seed. The crawl took 127171 seconds (35.33 hours). The download time was 39 thousand seconds (10.83 hours). Even though the crawl finished successfully it took a long time. So, there was a lot to learn from. This data helped me modify the crawler in future runs.

I was expecting to see the majority of the time would be spent downloading pages. But my results showed otherwise. Below is the distribution of the time spent and a pie chart.

Table 1: Time Spent by Activity for 100K Crawl

<table>
<thead>
<tr>
<th>Activity</th>
<th>Time Spent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Download</td>
<td>39011</td>
</tr>
<tr>
<td>ExtractUrl</td>
<td>83314</td>
</tr>
<tr>
<td>Other</td>
<td>4846</td>
</tr>
</tbody>
</table>
Figure 1: Pie Chart for 100K Crawl Time Spent Distribution

The actual rate of processing was $100000 / (127151 – 54(\text{delay})) = 0.78$ URLs per second

Using this rate, if it stayed consistent, which looked unlikely, it would have taken $780,000$ seconds or 9 days to crawl 1 million URLs. As the load increased, it was probable that this average time could have gone down. Even though the numbers were not promising, it provided a perspective, as how the volume processing of URLs should be done.

The numbers indicated that there was something not quite right. Extracting links from documents and processing them was taking about 65% of the elapsed time. This showed that a lot more time was being spent on processing than the download. The delay time spent was not significant. I took this information and reviewed the program, to see if
there was any excessive processing or any inefficiency. I tried to remove every place that
was performing any excess processing.

Other than the program issues, there were several issues after this crawl. This crawl saved
all the files it downloaded, and it ran on dirac.uccs.edu. The dirac machine was
very low on available space and also had a small hard disk. When I started the crawl
about 79% was used. By the time the crawl finished machine was 91% full, and few
applications could run. So I had to get rid of all the downloaded files and the database,
since nobody was able to work on the machine. Unfortunately, due to this it was not
possible to obtain the details, as much as I would have liked to.

In this crawl, I was maintaining the Failedurls table which held the URLs that had
failed. One of the main reasons this table got so large was the URLseen field, and this
resulted in excessive entries in Failedurls. The restart process was tested for small
crawls before and restarted in a reasonable amount of time. Now it was time to test the
restart after a large crawl. The restart involved a join between URL, Crawler and
FailedUrls tables. The sheer volume of rows in the 3 tables was so high that the
restart hung up the process. So, it was apparent that all was not well with the restart, and
restart logic had to be rethought. Even though most of the time was accounted for, very
little time (4%) of the time was not spoken for.

Taking lessons from this 100K crawl, I modified the process to make it more efficient.
Meanwhile I was also coming across several problems that are fully explained the
Problems Faced and Solutions Section (Chapter 5. Problems faced and Solutions). Trying
to get restart working, the focus was to reduce the number of rows in failedurls. This required the addition of a counter in that table. The restart was tested on short crawls and was again working, but still needed to be tested on a large crawl. I also made modifications to the program to delete a file after processing is completed so that large crawls could be performed without having to worry about bringing a machine down.

### 6.2 500 K Crawl (Aborted after 440 K)

After doing several small crawls with the modified program, it was time to do a large crawl. This time the crawl was done for not just uccs.edu but using five seed URLs from other University of Colorado (CU) Domains:

- [http://www.uccs.edu](http://www.uccs.edu)
- [http://www.colorado.edu](http://www.colorado.edu)
- [http://www.cusys.edu](http://www.cusys.edu)
- [http://www.cudenver.edu](http://www.cudenver.edu)
- [http://www.ucdhsc.edu](http://www.ucdhsc.edu)

Contact was established with CU Domain Webmasters to seek their approval for crawling. CU Boulder had me do a trial crawl for 50 thousand URLs, to check how much load it would put on their system. After all the requirements were met, the approval was given for a larger crawl. The crawl for 500K URLs was started. Learning from the 100K Crawl, intermediate logs were incorporated to view the incremental progress.

The following sections detail the results obtained from this crawl.
Table 2: Cumulative Time Spent and Average for 400 K Crawl

<table>
<thead>
<tr>
<th>URLs</th>
<th>Total Time Taken</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>34145</td>
<td>2.928686</td>
</tr>
<tr>
<td>200000</td>
<td>79566</td>
<td>2.513636</td>
</tr>
<tr>
<td>300000</td>
<td>148463</td>
<td>2.020705</td>
</tr>
<tr>
<td>400000</td>
<td>295723</td>
<td>1.352617</td>
</tr>
<tr>
<td>410000</td>
<td>336591</td>
<td>1.218096</td>
</tr>
<tr>
<td>420000</td>
<td>390053</td>
<td>1.076777</td>
</tr>
<tr>
<td>430000</td>
<td>480314</td>
<td>0.895248</td>
</tr>
<tr>
<td>440000</td>
<td>584699</td>
<td>0.752524</td>
</tr>
<tr>
<td>440500</td>
<td>590570</td>
<td>0.74589</td>
</tr>
</tbody>
</table>

Figure 2: Bar Chart for Cumulative Time Taken for 440K Crawl

At its start, for 100,000 URLs it took only 34145 seconds (9.49 hours). This was almost one third of the time taken for the previous 100K Crawl. This time the crawl was on a much larger set of domains than the previous crawl. So it was apparent that a lot of deficiencies and redundant processing were removed. The crawler continued to perform reasonably until almost 300,000 although the average went down from 2.92 to 2.02 URLs per second.

As it progressed, the processing slowed down very quickly, and by the time it had crawled 440,000 URLs it had taken almost twice the time to process half as many URLs as before. Between 440,000 and 440,500 URLs, it took approximately 6000 seconds, or close to 2 hours. The time growth rate was deeply worrisome, and hence the crawl was aborted. Even though the crawl took a lot longer, putting the numbers under analysis showed that some progress was made, but it was not nearly enough.
The average rate of processing was \( \frac{440500}{590570 - 442}\) (delay) = 0.74 URLs per second. Since the rate was deteriorating, it would have taken a lot more than 746448 seconds.

Looking at the distribution of time spent, I was able to find some new issues, which were not apparent in the previous crawl.

**Table 3: Time Spent Distribution for 440K Crawl**

<table>
<thead>
<tr>
<th>Activity</th>
<th>Time spent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Download Time</td>
<td>181966</td>
</tr>
<tr>
<td>ExtractUrl Time</td>
<td>278144</td>
</tr>
<tr>
<td>Other</td>
<td>130460</td>
</tr>
</tbody>
</table>

**Figure 3: Pie Chart for Time Spent Distribution for 400K Crawl**

Compared to the previous crawl, the amount of time in `extracturl` was reduced from 65% to 47%, but the other time jumped from 4% to 22%. I investigated the possible causes for it, and could not find any in the program. Researching the way indices are created in `mysql`, provided some insight and helped me to partially resolve this.

For load checking on `ExtractUrl` the following table and chart is useful.
Table 4: Content Type Distribution for 440K

<table>
<thead>
<tr>
<th>Number of URLs</th>
<th>Document Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>329475</td>
<td>Text</td>
</tr>
<tr>
<td>110864</td>
<td>Image</td>
</tr>
<tr>
<td>370</td>
<td>Other</td>
</tr>
</tbody>
</table>

Content Type Distribution for 440K Crawl

Figure 4: Content Type Distribution for 440K Crawl

This shows that 75% of the pages downloaded were text and needed `extracturl` processing. It was encouraging to see that the percentage of time was reasonable. But the fact still remained that the URL processing time was still higher than download, which suggested that there was still room for improvement. The next table shows return code distribution for this crawl.

Table 5: Return Code Distribution for 440K Crawl

<table>
<thead>
<tr>
<th>Number of URLs</th>
<th>Return Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>382810</td>
<td>200</td>
</tr>
<tr>
<td>1173</td>
<td>400</td>
</tr>
<tr>
<td>776</td>
<td>403</td>
</tr>
<tr>
<td>55950</td>
<td>404</td>
</tr>
</tbody>
</table>
More successful URLs meaning Crawler spent more time doing useful operation.

6.3 One Million Crawl

After the 440K crawl, I researched indexing methods to achieve best results for the least amount of effort. Cardinality is the number of unique index values. The Cardinality measure is extremely important in measuring performance. One should use the least number of bytes in the index to achieve the highest cardinality. We can always get the highest cardinality by using the entire field as index, but we get better performance by using a part of the field and get the same cardinality. I used the database created for the 440K crawl to do ‘what if’ analysis for the cardinality. The results showed that by using less than 25% of the size of the field we can have the same cardinality as provided by the entire field. I used this approach to reduce the content hash and URL hash index size. The program now had to lookup at 75% less data than before and the benefit was that it provided consistent performance in the 1 million crawl. It helped me achieve scalability in terms of spending similar amount of time for similar amount of load, irrespective of how many URLs were already crawled. It also helped identify and reduce the other category time, which had gone up to 22% in the 440K crawl.
Among other improvements, I created a better model of keeping track of failed URLs. This reduced redundancy and helped better manage the statistics to keep. The new model was a ReasonCode Table. It can hold different error codes, which each have a counter to show the number of occurrences of a particular error.

The following table shows details about time spent in individual 100 thousand URL crawl. The first column shows the incremental URL number. Due to politeness restrictions, as we progress in crawling URLs, the numbers of subdomains continue to get exhausted, and when we are left with very few subdomains, the crawler has to enter a delay period where it cannot request anything from any domain. This effect is seen since the delay starts increasing as we go down the table. One would normally think that more data to download should take more time and less data to download should take less time. My observations are somewhat different. The amount of data downloaded in a second varies on parameters such as the geographic location of the server, the time of the day, the load on the server fulfilling request, the load on the network behind which crawler is running, and the number of documents to download. Each one of this has some effect on the download time.
Table 6: Time Spent for 100000 URL intervals in 1 Million Crawl

<table>
<thead>
<tr>
<th>EndUrl</th>
<th>Elapsed Time</th>
<th>Delay Time</th>
<th>Elapsed – Delay</th>
<th>Download Time + Waiting Time</th>
<th>Download Time</th>
<th>Waiting (failed download URLs time)</th>
<th>ExtractURL Time</th>
<th>No of Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>33241.598</td>
<td>44.35</td>
<td>33197.25</td>
<td>18301.05</td>
<td>14406.45</td>
<td>3624.59</td>
<td>12133.00</td>
<td>3,315,208,754</td>
</tr>
<tr>
<td>200000</td>
<td>36450.488</td>
<td>47.76</td>
<td>36402.73</td>
<td>18793.44</td>
<td>17249.07</td>
<td>1544.37</td>
<td>14126.00</td>
<td>2,791,684,449</td>
</tr>
<tr>
<td>300000</td>
<td>53334.000</td>
<td>974.00</td>
<td>52360.00</td>
<td>28275.00</td>
<td>23026.00</td>
<td>5249.00</td>
<td>19925.00</td>
<td>3,091,750,305</td>
</tr>
<tr>
<td>400000</td>
<td>65025.000</td>
<td>7343.15</td>
<td>57681.85</td>
<td>30778.77</td>
<td>28640.12</td>
<td>2138.65</td>
<td>21303.25</td>
<td>2,816,543,234</td>
</tr>
<tr>
<td>500000</td>
<td>77234.080</td>
<td>13020.18</td>
<td>64213.90</td>
<td>35374.88</td>
<td>31148.03</td>
<td>4226.85</td>
<td>23384.82</td>
<td>2,517,962,504</td>
</tr>
<tr>
<td>600000</td>
<td>90065.840</td>
<td>21927.99</td>
<td>68137.85</td>
<td>37486.18</td>
<td>36061.84</td>
<td>1424.34</td>
<td>27892.05</td>
<td>1,754,020,958</td>
</tr>
<tr>
<td>700000</td>
<td>95837.119</td>
<td>24612.86</td>
<td>71224.26</td>
<td>36775.84</td>
<td>36195.77</td>
<td>580.07</td>
<td>28469.21</td>
<td>1,567,771,821</td>
</tr>
<tr>
<td>800000</td>
<td>93594.030</td>
<td>20515.20</td>
<td>73078.83</td>
<td>36721.17</td>
<td>35506.94</td>
<td>1214.23</td>
<td>30429.00</td>
<td>1,797,046,330</td>
</tr>
<tr>
<td>900000</td>
<td>94390.640</td>
<td>22907.77</td>
<td>71482.87</td>
<td>37558.12</td>
<td>36391.15</td>
<td>1166.97</td>
<td>27798.07</td>
<td>1,480,009,870</td>
</tr>
<tr>
<td>1000000</td>
<td>100768.470</td>
<td>17592.76</td>
<td>83175.71</td>
<td>39835.28</td>
<td>38287.31</td>
<td>1547.97</td>
<td>36747.51</td>
<td>1,870,878,602</td>
</tr>
</tbody>
</table>
As we can see here, the download time can increase even if the total amount of data downloaded is reduced, e.g., for 100000 URLs the data downloaded exceed that for 200000 URLs but the time taken for download and waiting is greater. On the other hand, if the data downloaded increases, it can get downloaded in less time. An example of this case is in the 700000 URLs row where the content downloaded is less than the 800000 row but the time taken for download (for 800000) is less. When the crawler requests a page from the Web server, it may succeed or fail. The case of success is beneficial to the crawler in terms of quantity of data downloaded. In case of failure, it has been observed that all Web servers are not equal. Some took a very long time to inform the crawler that the content is not available. This causes worse performance of the crawler, since during the time it takes for the Web server to respond, the crawler has not done any useful work during that time frame. This time was initially not recorded, hoping that it would not be substantial. After this was analyzed, it is easy to see how significant it is, in terms of download time.

The following content distribution diagram shows that 79% of the content downloaded was text. That results in `extracturl` processing for 79% of the content.
Figure 6: Content Type Distribution for 1 Million Crawl

Extracturl time increased for each iteration, per the table given above (Table 6: Time Spent for 100000 URL intervals in 1 Million Crawl). The time continued to increase. The next figure, Figure 7: Incremental Content Type Distribution (1 Million Crawl) shows the distribution of content for every 100000 URL intervals. It shows that the number of text pages received began at 49000 in first 100 thousand pages and continued to increase until the end of 1 Million crawl. It means that extracturl processed more documents for every iteration.
Needle Crawler

Master’s Project Report

UCCS

Figure 7: Incremental Content Type Distribution (1 Million Crawl)
The next Figure 8: Return Code Distribution (1 Million Crawl) pie chart shows the return code distribution for crawl of 1 million URLs.

![Return Code Distribution](image)

**Figure 8: Return Code Distribution (1 Million Crawl)**

It shows that 94% of the downloads were successful. However, this does not tell why the total time of crawling increased in every subsequent 100000 URLs. The next chart sheds some light on it. As the crawl progresses, the number of successful retrievals kept increasing which makes the crawler do more work than the prior iteration.

The next chart shows the percentage of text content in terms of bytes of the total download. It was already seen that number of text pages increased as the crawl progressed, this chart shows that the text content became a greater part of the total download. This results in not only more documents to process, but the size of each document also increased.
Figure 9: Bar chart for Text and total download for 100K Crawl Intervals

These different views of the result data help us find answers to why time for extracturl increased. The next chart shows the total time elapsed plotted along with other significant times.
Figure 10: Cumulative and Interval specific time spent for 1 Million Crawl

We can see that the cumulative total time increased, but it appears to be a linear increase.

The percentage of time spent in various activities is shown in next pie chart.

Figure 11: Pie Chart for Time Spent by Activity (1 Million Crawl)
The percentages of time shown in the above chart, reflects reality more accurately as downloading takes more time than everything else. One new entry in this chart is delay time, and it is important to note how significant delay time is in the total elapsed time for crawl.

One of the lessons learnt from this exercise was that, it is important to capture time taken in all activities, large or small, important or not. All of the times become significant when large crawls are done. The analysis of time helped me to find where the deficiencies were and take appropriate measures to fix them.

**6.4 Restart**

Having done a lot of crawls was a good introduction to dos and don’ts for restart. After accumulating too many entries in FailedUrls Table and trying to do a join, it was evident that the join was an inefficient way to go. Even after introducing a counter in the FailedUrls to reduce the number of rows, the join requirement did not go away. The fact is that gathering all of the information was necessary, but storing it in too much detail caused too much data processing. Additional logic was developed and table changes were made to the information stored within the tables while crawling so that the need for computing a 3 table join was no longer necessary. Then it became a simple select which would load the URLs in the queue. The results scaled very well as shown in the below Table 7: Restart Performance. During the 1 Million crawl, the restart was done more than 10 times. The data below show how many URLs were loaded into queues and the time taken for the same.

<table>
<thead>
<tr>
<th>Number of URLs loaded in restart</th>
<th>Time Taken</th>
<th>Average URLs processed per second</th>
</tr>
</thead>
</table>

Sonali Patankar  
Page 53 of 64  
Fall 2006
As we can see the time taken was linearly proportional to the number of URLs loaded. As the load increased, the time required stabilized. The variations in the ratio can be attributed to the amount of memory available at the time of restart since this operation loads the URLs into the queue structure. The maximum time spent for 1732762 URLs was only 145 seconds.
### 6.5 Comparison with other published Crawlers [13]

<table>
<thead>
<tr>
<th>Crawler</th>
<th>Needle</th>
<th>Google</th>
<th>Mercator</th>
<th>Internet Archive</th>
<th>Polytechnic</th>
<th>Thailand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Config</td>
<td>Intel P4 1.8 Ghz, 1 GB RAM, 225 GB Hard disk</td>
<td>Not available</td>
<td>Four Compaq DS20E 666Mhz Alpha Servers</td>
<td>Not available</td>
<td>3 Sun Ultra 10 Workstations and a dual processor Sun E250</td>
<td>Four sets of AMD Athlon XP 1500+, 786 MB RAM, six 35GB Hard Disks</td>
</tr>
<tr>
<td>Data structure for URLs</td>
<td>Queue</td>
<td>Mem hash table disk sorted list</td>
<td>Bloom Filter per domain</td>
<td>Mem Red-Black tree disk sorted list</td>
<td>AVL tree of suffixes</td>
<td></td>
</tr>
<tr>
<td>DNS Solution</td>
<td>Stored locally in database</td>
<td>Local cache</td>
<td>Custom</td>
<td>Adns</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programming Language</td>
<td>Perl</td>
<td>C++/Python</td>
<td>Java Rewrote Java</td>
<td>C++/Python</td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>Parallelism per machine</td>
<td>1</td>
<td>300</td>
<td>100</td>
<td>64</td>
<td>1000</td>
<td>300</td>
</tr>
<tr>
<td>System Size</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Crawl Order</td>
<td>1 Million</td>
<td>24 Million</td>
<td>891 Million</td>
<td>100 Million</td>
<td>120 Million</td>
<td>400 Thousand</td>
</tr>
<tr>
<td>Crawl Rate (pages/sec)</td>
<td>1.78</td>
<td>48</td>
<td>600</td>
<td>10</td>
<td>140</td>
<td>618</td>
</tr>
<tr>
<td>Effective Crawl Rate (1*1) = 0.04 pages per second</td>
<td>48 / (4*300) = 0.15 pages per second</td>
<td>600/(4*100) = 1.5 pages per second</td>
<td>10/64 = 0.15 pages per second</td>
<td>140/(3*1000) = 0.04 pages per second</td>
<td>618 / (4*300) = 0.51 pages per second</td>
<td></td>
</tr>
</tbody>
</table>

In the Fall 2005, Jing Yang[^10] wrote a text search engine with an integrated crawler at UCCS which crawled UCCS domain. No detail data is available on that crawler. For the summary data, the crawler took about a week to crawl 50,000 URLs and crawled only html files.

[^10]: http://cs.uccs.edu/~jyang/search3.php
Most published crawlers in the Table 8: Comparison with published crawlers are distributed. They run hundreds of threads on multiple machines. The timeline for the various results varies from 1997 to 2001. Needle crawled 1 Million URLs and was able to achieve effective crawl rate of 1.78 per second. While this looks better than other’s effective ratio, with due respect to them, they have crawled many more million pages and would have faced additional challenges to mine. In comparison, Needle Crawler shows promise to provide competitive crawler performance for crawling large number of URLs.


7. Conclusion

During this project, I developed and studied a Web Crawler from both feature and performance perspectives. Programmatically, the perl language is well suited for getting the design of crawler started, without having to worry too much about low level details. Although there are many documents available about crawlers, they provide high level design. Algorithms and detailed design are mostly kept as a trade secret.

I adapted some ideas from Mercator [1] for general design. There were several situations where I faced unique problems and developed ideas on my own to deal with the problems at hand. It was a great learning experience, which helped me realize intricacies of Web Crawlers, and how a large search engine operates.

My study shows that the performance of the crawler I designed and developed scales up reasonably well. The crawler completed the same amount of work in same time throughout the timeline of the 1 million crawl. The overall average depends on both the download average and extracturl average. In the 1 million crawl, the average number of documents downloaded per second continued to decrease. The Extracturl processing average varied within a reasonable range and improved when the load reduced. As with every process, there is always room for improvement from approach and processing standpoints.
7.1 Future Work

Presently, the Needle crawler processes text, html, php and image files. The amount of information available increases everyday and it is stored in additional formats such as PDF, Word, etc. To capture links from such documents, a reader module can be written to convert the information in them to text and passed back to the needle crawler core. With this the Needle Crawler can provide higher coverage.

The Needle Crawler downloads the files in their entirety. For the 1 million URL crawl, the downloaded data was over 20 GB. While that does not sound a lot in the age of large hard disks, archival techniques can be applied to retain data for a longer time. This feature can also be used to provide a cached feature to the Needle search engine front end with minimal usage of disk space.

Crawling is a time consuming operation. The Distribution of time spent over various subtasks in crawling is very important in finding bottlenecks. The more detailed the time keeping gets, the more complicated it becomes to extract useful information. During all the crawls I did, I manually analyzed the numbers to understand how pieces of the crawler were performing. A tool to analyze the results can help achieve the same results faster and also can do a more detailed analysis.

Needle is a single threaded crawler. Results indicate that downloading a page takes the majority of time. To make better use of bandwidth and CPU, link extraction can be executed in thread(s), so that crawler makes itself available to download more pages.
Another possible improvement can be, to queue up link extraction processing and perform it in the delay time when applicable.

It will also be nice to provide a user interface for modifying parameters in the configuration file \texttt{config.xml}. Refer Chapter 4. Implementation

\section*{7.2 Considerations for Large Crawlers}

I learnt a lot from the number of crawls performed for this project. It helped me understand what the important considerations are for industrial strength crawlers. I provide a summary of items which should always be analyzed before undertaking a large crawling project.

\textbf{High and Dedicated Bandwidth and CPU} – A Crawler downloads a lot of data. Hence the bandwidth becomes a very important consideration that helps dictate the download time taken. During the Needle crawls, the maximum bandwidth (1435Kbps) achieved was about four times the minimum (307Kbps). Obtaining higher bandwidth is a very important consideration.

\textbf{Code Reuse, language selection} - Crawler programs run for a long time due to the nature of work. The language used for the crawler program and operating system it runs on are the immediate candidates for analysis. While languages like Perl offer package support for many of the tasks carried out by a Crawler, low level languages can help achieve higher efficiency. The tradeoffs for using low level languages are low portability, difficult maintenance and difficult upgrade. Code reuse can help the load modules become smaller and run faster.
Main Memory – To speed up some of the operations the crawler needs to store information in working storage variables. Most crawlers, Needle included, maintain a data structure to hold the URLs to be crawled and keep them in main memory. These structures can get very large for larger crawlers and can have performance implications. Due consideration should be given to keep the usage of main memory to a minimum so it does not affect performance.

Permissions – When a very large site is crawled, it becomes apparent to the webmaster that many pages are being requested in a short amount of time. In today’s world, with a lot of hackers trying to bring systems down, this can be construed as an attack, if the Webmaster has no knowledge that this was a crawl and not an attack. Permission must be obtained from the target domains and communication information provided, so that in case of heavy load on the target domain, a crawl can be deferred to a more suitable time.

DNS – Every large crawler must consider storing DNS information for the URLs it crawls. Maintaining this information helps the crawler, not go through the process of obtaining it for every URL every time. The Needle crawler stores the IP address in its database, and uses it to request URLs from previously seen domains. This can reduce load on the local DNS server and also the overall time consumption.

Large set of Domains – Large crawls can mean large number of URLs or large number of domains which may not have large number of URLs. In the Needle crawl, I observed that having a large number of domains to work with is extremely important. It helps in load leveling and politeness compliance.
**Efficient Restart** – Some crawlers take snapshots of the crawler operation at regular intervals to help in restart process. Determination of how much detail to capture and at what interval is very important. One might end up with too much or too little information. Since there are many ways to keep this information, the focus should be on organization if it. Retrieval techniques to get the crawler back in operation from when it aborted or stopped are crucial. The Needle crawler maintains this information as part of its operation, so the databases are up to date, and hence facilitate easy restart.

**Dynamic Control** – A large crawl puts a crawler through many unforeseen situations. Sometimes it may try to obtain URLs from a server that is down. Some non-operating servers can take more time than others. While the crawler can get unsuccessful return codes, it causes quantity and quality reduction of crawler data capture. A Crawler can be made smart, to recognize which domains have a large failure rate, and defer their crawl to a later time.

**Time Keeping** – A lot of activities a crawler performs are similar from one URL crawl to another. But that’s where the similarity ends. While downloading a URL, the crawler is operating in slightly or sometimes very different conditions from all other URL crawls. Some of the issues may have to do with slow response from server, or too many links to process, or the database inserts, updates and reads taking too much time. Detailed time keeping at regular intervals allows the developer to look at how much time the crawler spent in individual activities. This can sometimes help correct a deficiency that may be external to the crawler, but may significantly impact crawler performance.
**Large and fast Storage** – The size of the data downloaded require large and fast storage devices. This can have major implications on crawler performance.
8. References


http://citeseer.ist.psu.edu/cache/papers/cs2/266/http:zSzzSzwww.cs.ucsd.eduzSz~dboswe