

# **YIOOP FULL HISTORICAL INDEXING IN CACHE NAVIGATION**

A Writing Report

Presented to

The Faculty of Department of Computer Science

San José State University

In Partial Fulfillment of

The Requirements for the Degree

Master of Science

By

Akshat Kukreti

May 2013

© 2013

Akshat Kukreti

ALL RIGHTS RESERVED

# **SAN JOSÉ STATE UNIVERSITY**

The Undersigned Project Committee Approves the Project Titled

YIOOP FULL HISTORICAL INDEXING IN CACHE NAVIGATION

By

Akshat Kukreti

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

---

Dr. Chris Pollett, Department of Computer Science

Date

---

Dr. Chris Tseng, Department of Computer Science

Date

---

Dr. Soon Tee Teoh, Department of Computer Science

Date

## **Abstract**

### **Yioop Full Historical Indexing In Cache Navigation**

This project adds new cache-related features to Yioop, an Open Source, PHP-based search engine. Search engines often maintain caches of pages downloaded by their crawler. Commercial search engines like Google display a link to the cached version of a web page along with the search results for that particular web page. The first feature enables users to navigate through Yioop's entire cache. When a cached page is displayed along with its contents, links to cached pages saved in the past are also displayed. The feature also enables users to navigate cache history based on year and month. This feature is similar in function to the Internet Archive as it maintains snapshots of the web taken at different times. The contents of a web page can change over time. Thus, a search engine caching web pages has to make sure that the cached pages are fresh. The second feature of this project implements cache validation using information obtained from web page headers. The cache validation mechanism is implemented using Entity Tags and Expires header. The cache validation feature is then tested for effect on crawl speed and savings in bandwidth.

## **Acknowledgements**

I would like to express my sincere gratitude to my Masters Project advisor, Dr. Chris Pollett. Without his guidance, constant encouragement and support, the project would not have been completed on time.

I would also like to express my appreciation to Dr. Chris Tseng and Dr. Soon Tee Teoh for their time, and for being part of my CS298 committee.

# Contents

YIOOP FULL HISTORICAL INDEXING IN CACHE NAVIGATION .....	1
1. Introduction .....	9
2. Background.....	11
2.1 Web Crawlers.....	11
2.1.1 The Crawling Process .....	11
2.2 Web Caches And Search Engines.....	12
2.3 The Internet Archive and WayBack Machine .....	14
2.3.1 The Internet Archive.....	14
2.3.2 The WayBack Machine .....	14
2.4 Yioop Search Engine Software .....	17
2.4.1 Yioop's Model.....	17
3. Implementing The History Feature .....	20
3.1 Modifying Yioop's Cache Request And Output Code .....	20
3.2 Modifying Links Within Cache Pages.....	21
3.3 History UI.....	23
4. Entity Tags And Expires Headers.....	24
4.1 Entity Tags .....	24
4.1.1 Experiment With Entity Tags, PHP, and cURL .....	25
4.2 Expires Header.....	26
5. Modifying Yioop's Fetcher .....	27
6. Disk Access Experiment.....	29
7. Data Structure for Cache Page Validators.....	31
7.1 B-Trees.....	31
7.2 B-Tree Implementation in PHP .....	33
7.3 Augmenting Yioop's Queue Server .....	34
7.4 Implementing the Cache Page Validation Feature.....	35
7.4.1 Queue Server pseudo-code for storing Cache Page Validators .....	35
7.4.2 Queue Server pseudo-code for Lookup of Cache Page Validation Data when producing fetch batches.....	36
7.4.3 Fetcher pseudo-code for Fetcher for handling Cache Page Validation Data .....	37
8. Experiments.....	38

8.1 Testing the effect on fetch batch creation time.....	38
8.2 Determining effect of cache page validation on bandwidth .....	41
9. Conclusion .....	43
Bibliography .....	44

## List of Figures

Figure 1: Example of robots.txt Source: <a href="http://www.robotstxt.org/robotstxt.html">http://www.robotstxt.org/robotstxt.html</a> .....	11
Figure 2: Google's cache link Source: <a href="http://google.com">http://google.com</a> .....	13
Figure 3: Section of a cached page Source: <a href="http://google.com">http://google.com</a> .....	13
Figure 4: WayBack Machine interface Source: <a href="http://archive.org">http://archive.org</a> .....	15
Figure 5: Yioop crawl in a multiple Queue Server setting .....	18
Figure 6: Searching within the cache .....	21
Figure 7: Modified links within cached pages .....	22
Figure 8: Redirection within the cache .....	22
Figure 9: Yioop's History feature .....	23
Figure 10: ETag extraction using PHP cURL .....	25
Figure 11: ETag headers in PHP cURL.....	26
Figure 12: Yioop Work Directory .....	27
Figure 13: Cache page validation data written by Yioop's Fetcher .....	28
Figure 14: Disk access experiment results .....	30
Figure 15: A B-Tree with minimum degree 501 .....	32
Figure 16: B-Tree node for cache page validation feature .....	33
Figure 17: Cache page validators written by Queue Server.....	34
Figure 18: B-Tree nodes for cache page validators .....	34
Figure 19: Table comparing fetch schedule write times .....	38

# 1. Introduction

The idea behind this project is to enhance Yioop's cache related functionality in terms of displaying cached versions of web pages and maintaining cache freshness. The web is an important source of information and is ever changing. Web pages that are active today may become inactive tomorrow. The Internet Archive takes regular snapshots of the web so that along with the current versions of web pages, previous versions are also accessible. It does not however, support full text search on the archived pages. It would be an improvement; if, along with the search results displayed by a search engine, users are also able to access cached copies of the page taken at an earlier date, and also, be able to perform full text search on cached pages. This is the motivation behind the first feature of this project which enables users to view all cached copies of a web page. The feature keeps track of cached pages during search time. A search engine may re-crawl a set of web pages after a certain number of days. If the content of the page has not changed, bandwidth is wasted on downloading the page again. Bandwidth can be saved by checking if a web page is still fresh. For example, Google uses If-Modified-Since header for validating cached pages. The motivation behind the second feature of this project is to experiment with a different way of validating cached versions of web pages that uses Entity Tag and Expires header. The second feature keeps track of cached pages during crawl time.

This report describes the work done in my Masters project, and consists of the following sections. The first section describes the background and preliminary work where I explored Yioop! and the Internet Archive. The second section describes my experiments with Yioop!'s cache request and output mechanism, modification of links within cached pages and, implementing and testing a history feature that enables users to view all cached versions for a web page. The third section describes my experiments with PHP, cURL and Entity Tags. The fourth section describes how I modified Yioop!'s Fetcher program to separately send cache-

related data to the Queue Server. The fifth section describes my experiments for estimating a lower bound on the lookup of ETags. It also describes my implementation the B-Tree data structure for storage and lookup of ETags, and integration with Yioop!'s Queue Server for scheduling. The sixth section describes experiments with the modified Yioop! code and their results. The last section of the report is conclusion and future work.


## 2. Background

### 2.1 Web Crawlers

Search engines use web crawlers to build their search index of the web. Web crawlers are programs that visit links, download the web pages for those links, extract links from the downloaded web pages, and repeat the process for the extracted links (Stefan Büttcher, 2010). The responsibilities of a web crawler include de-duplicating web pages, re-crawling web pages to ensure that the search index is up to date, assigning priority to web pages so that the most important web pages are re-crawled before the less important ones. Web crawlers maintain a priority queue of URLs that have already been seen to determine which URLs should be re-crawled first (Stefan Büttcher, 2010).

#### 2.1.1 The Crawling Process

The crawling process starts with an initial set of URLs. The web crawler begins by selecting URLs from the priority queue of URLs. Next, the web crawler performs Domain Name Translation to determine the IP address for the URL. Before downloading a web page, a web crawler must check if it is allowed to download that web page. Web site owners use "robots.txt" file to indicate to web crawlers, the pages of the web pages it is allowed to crawl (About /robots.txt). A "robots.txt" file contains directions along with paths to resources that the web crawler may not download.



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

Figure 1: Example of robots.txt Source: <http://www.robotstxt.org/robotstxt.html>

Figure 1 shows an example of a robots.txt file. The "User-agent: crawler-name" is at the start of the file. It denotes the names of the crawlers for which, the instructions that follow, apply. In the example, the \* next to User-agent denotes that the instructions must be followed by all web crawlers (Stefan Büttcher, 2010). The "Disallow: path" indicates that the web crawler must not download the resource. In the example, Disallow: / indicates that all the web crawler must not download any resource associated with the URL.

After making sure that a page can be downloaded, the web crawler downloads the page. The web crawler then processes the page to extract information about the page which includes URLs. The page is then saved so that it may be added to the search engine's index. The URLs extracted during processing of the downloaded pages are then added to the priority queue of URLs. The web crawler then again extracts URLs from the priority queue, and the process continues.

## **2.2 Web Caches And Search Engines**

A web cache is used to store copies of web pages accessed by the program maintaining the web cache. Maintaining web caches is useful as recent copies of web pages can be accessed without actually visiting the link for that particular web page. If a web page containing information of interest is not active, but the web cache contains a recent copy of the web page, the desired information can be obtained by accessing the cached copy (Web Cache).

Search engines also make use of web caches for saving copies of documents downloaded by their web crawlers. Commercial search engines like Google display the link to the cached version of the web page along with the results of the search query. Figure 2 shows how Google displays links to the cached version of a web page. Figure 3 shows what a cached page looks like. The top of the page contains the date at which the web page was cached.

**noarchive:** The noarchive directive can be used to tell the web crawler that a particular web page must not be archived. It can be specified using a <meta> tag within the contents of a web page, or specified using the X-Robots tag in the page headers.

For example, <meta name="robots" content="noarchive"/>

HTTP 1.1 200 OK

(...)

X-Robots-Tag: noarchive

In this project, the new features modify how Yioop handles cached copies of web pages during search time, and add a new method for validating cached pages during crawl time.

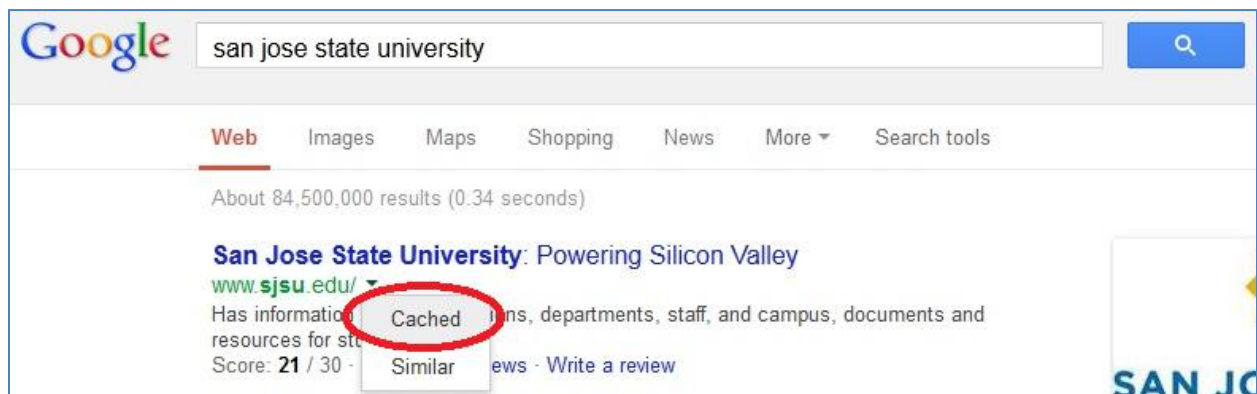


Figure 2: Google's cache link Source: <http://google.com>



Figure 3: Section of a cached page Source: <http://google.com>

## **2.3 The Internet Archive and WayBack Machine**

### **2.3.1 The Internet Archive**

The Internet Archive is an internet library that maintains a vast collection of historical web content. Its web page archive currently has 333 billion pages (Internet Archive). It was established in 1996 by Brewster Kahle and Bruce Gilliat in San Francisco, California (Rackley, 2010). The Internet Archive uses Heritrix, an Open Source, Java-based web crawler, for downloading content from the web (Rackley, 2010). For storing web crawls, the internet archive uses the WARC file format. WARC files save the downloaded web resources along with the meta-data associated with them such as encoding (WARC, Web ARChive File Format).

The history feature of this project allows users to access all versions of a cached page. This feature is similar to the Internet Archive as the entire history of cache pages is accessible to the user. But the history feature allows users to do full-text searches on the cached pages which is not supported by the Internet Archive.

### **2.3.2 The WayBack Machine**

Internet Archive's WayBack machine allows users to access the archived web pages. The Wayback Machine allows users to enter the URL of the webpage for which they wish to see the archived pages. On entering the URL, the Wayback Machine redirects to a page that contains snapshots of the webpage grouped by years and months. Users can select year and month to view a particular snapshot of the web page. There can be more than one snapshots per day for a particular web page. The motivation behind the first feature of this project was to provide a feature similar to the WayBack Machine in a search engine so that users are able to access the archive of cached pages along with the latest version. The history feature in Yioop displays linked to all cached versions of a web page, grouped by year and month.

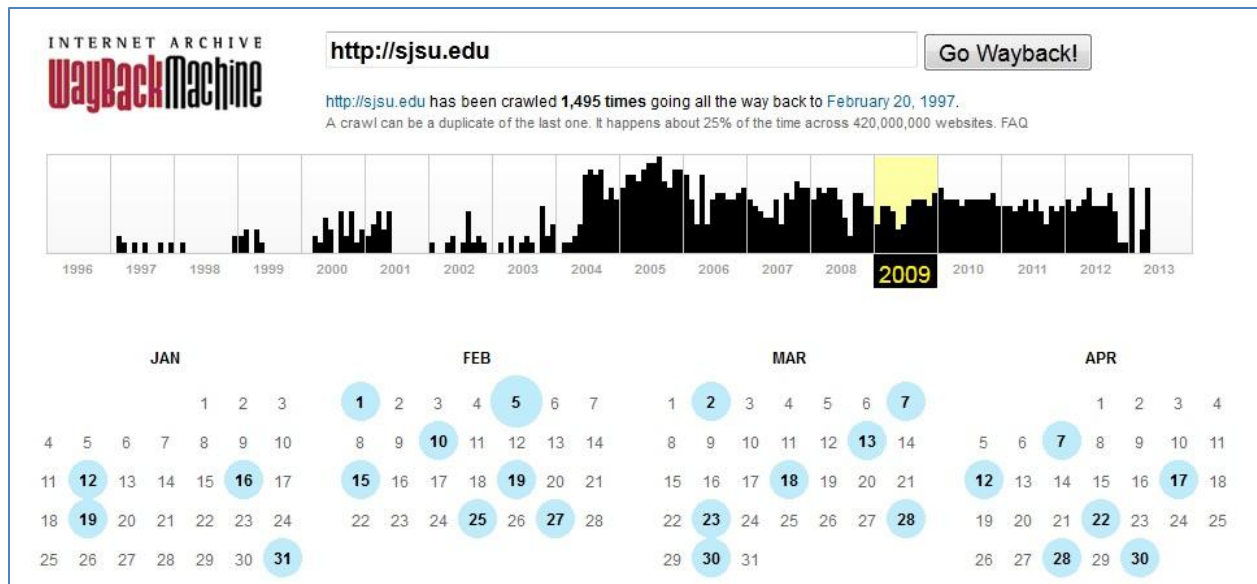


Figure 4: WayBack Machine interface Source: <http://archive.org>

### 2.3.2.1 URL Modification In The WayBack Machine

The Internet Archive modifies the URLs of web pages before caching them. The URLs are modified so that they redirect to a cached page instead of the live page. The link is modified by appending the time at which the page was cached, and the URL to a web prefix.

For example, consider the following modified link:

`web.archive.org/web/20050214202400/http://www.google.com`

Web page URL: `http://www.google.com`

Web Prefix: `web.archive.org/web/`

The number **20050214202400** Corresponds to the time at which the web page was cached.

Year: **2005**

Day and Month: **0214 (February 14)**

Time: **202400 (20:24:00)**

The URL is modified using JavaScript. When a user clicks on a particular date in the WayBack Machine's history interface, the JavaScript calculates the date and time from the click event, appends it to the Web prefix. The original URL is then appended after the date and time to form the modified link. The user is then redirected to the archive page with the modified link.

The history feature in this project also modifies links to cached pages. The link modification is done server-side unlike the WayBack machine's client-side link modification.

## 2.4 Yioop Search Engine Software

Yioop is an Open Source, PHP-based search engine software developed at Seek Quarry by Dr. Chris Pollett (Pollett). Yioop allows users to crawl websites and create search indexes for those websites. The indexing of website is done alongside the website crawl. After stopping a crawl, the resultant search engine can be used for querying Yioop (Pollett). Yioop uses the C-based multi-curl library for multi-threading, which it uses to download multiple web pages concurrently. In order to implement cache page validation in Yioop, Yioop's crawl process had to be modified to support extraction and use of cache page validators like Entity Tags and Expires headers.

### 2.4.1 Yioop's Model

Yioop requires a running web server in order to perform crawls and building search indexes. The major components are listed below. Yioop can work on a single machine and also in a distributed setting. In a distributed setting, each node consists of the components listed below (Pollett).

1. **Name Server:** The Name Server is responsible for coordinating the crawl process. In a distributed setting. The name Server is present in the node that is responsible for coordinating the crawl.
2. **Queue Server:** The Queue Server program is responsible for maintaining a priority queue for URLs. The Queue Server schedules URLs of websites to be downloaded and also adds the downloaded pages to the search index.
3. **Fetcher:** The Fetcher program is responsible for downloading web pages.

### 2.4.1.1 How Yioop Works

The Queue Servers create batches of URLs to be downloaded and save them in their respective working directories. The Fetchers periodically look for URLs to be fetched. When found, the Fetchers read in the batch of URLs to be fetched and download the web pages for those URLs. After downloading the web pages, the Fetchers extract information from the web page headers like and create summaries of downloaded web pages. The summaries are then written back to the working directory. The URLs are hashed on to the id of Queue Servers to make sure that the summary data goes to the correct Queue Server. The Queue Servers read the summary data and incrementally add the data to the search index. The URLs extracted from web page summaries are then added to the priority queue.

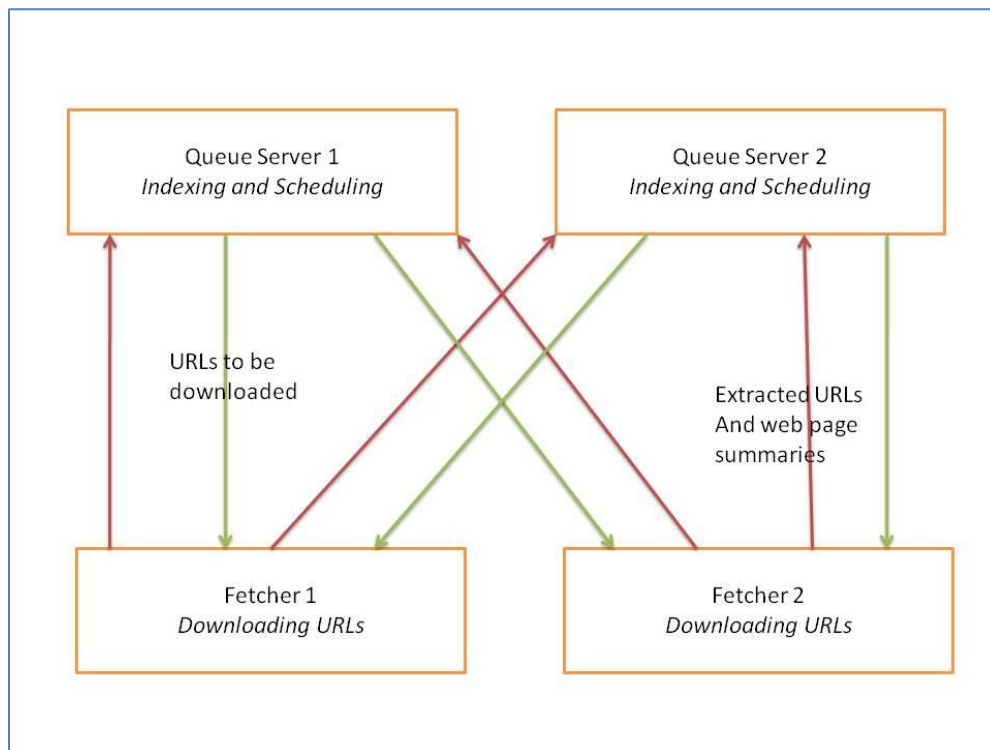


Figure 5: Yioop crawl in a multiple Queue Server setting

Figure 5 depicts Yioop in a multiple Queue Server setting. The second feature of this project augments the Fetcher to separately send the cache page validation data to the Queue Server. The Queue Server is modified to store the cache page validation data received from Fetchers, and use the saved cache page validation data when creating URL fetch batches for the Fetcher.

### 3. Implementing The History Feature

#### 3.1 Modifying Yioop's Cache Request And Output Code

As a first step towards building the history feature, I modified Yioop's cache request and output code. When Yioop displays the results of a search query, it also displays a link to the cached version along with each search result. The following is an example of a link to a cached page.

```
http://www.yioop.com/?YIOOP_TOKEN=7B-qsBHUu34|1354640487&
c=search&a=cache&q=sjsu&arg=http%3A%2F%2Fwww.sjsu.edu%2F&its=1336804999
```

The parameter "a = cache" means that the search is taking place within the cache and the parameter "its = 1336804999" is the timestamp at which the page was cached. When a cache page is requested, Yioop searches for the specified timestamp. If a cached version of a page exists for the specified timestamp, it is displayed. Otherwise, the a "no cache" is returned.

I modified Yioop's code so that if the specified timestamp is not found, Yioop tries to look for a timestamp that is nearest (past or future) to the specified timestamp that has a cached page associated with it. If such a timestamp is found, the cached page is displayed. Otherwise "no-cache" is returned.

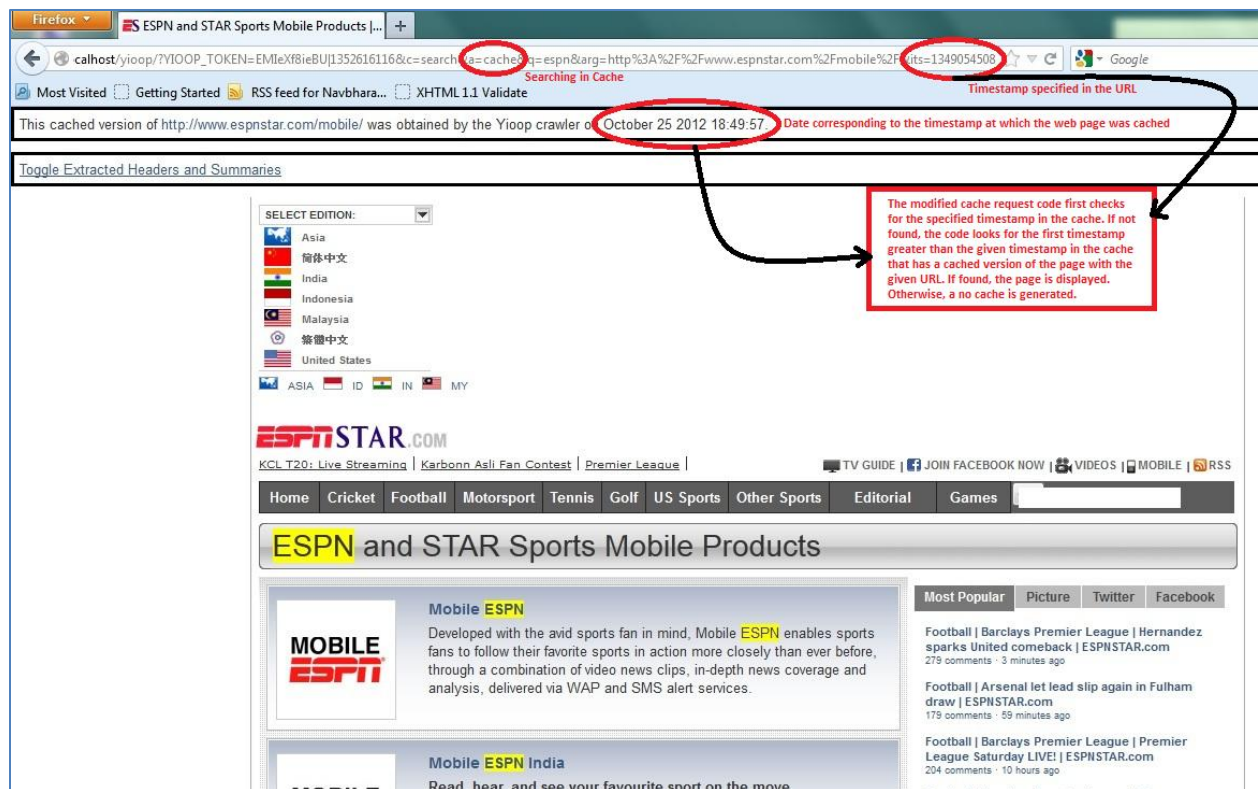


Figure 6: Searching within the cache

### 3.2 Modifying Links Within Cache Pages

In order to make sure that the links within a cached page also redirect to a cached page, I augmented Yioop's URL canonicalization code so that the links within the cached pages follow the cache search process in 3.1. The only difference is that the if no timestamp with a cached page is found for a link within a cached page, the link redirects to the live page. The link modification is similar to the modification done by the WayBack Machine. The modification however, is done at server-side.

In Figure 7, the link at the bottom of the page includes the flag "from\_cache = true" to indicate that the link is from a cached page. The timestamp is also appended to the link so that it redirects within the cache before going to the live site. The modified link redirects to a cached page as shown in Figure 8.



Figure 7: Modified links within cached pages

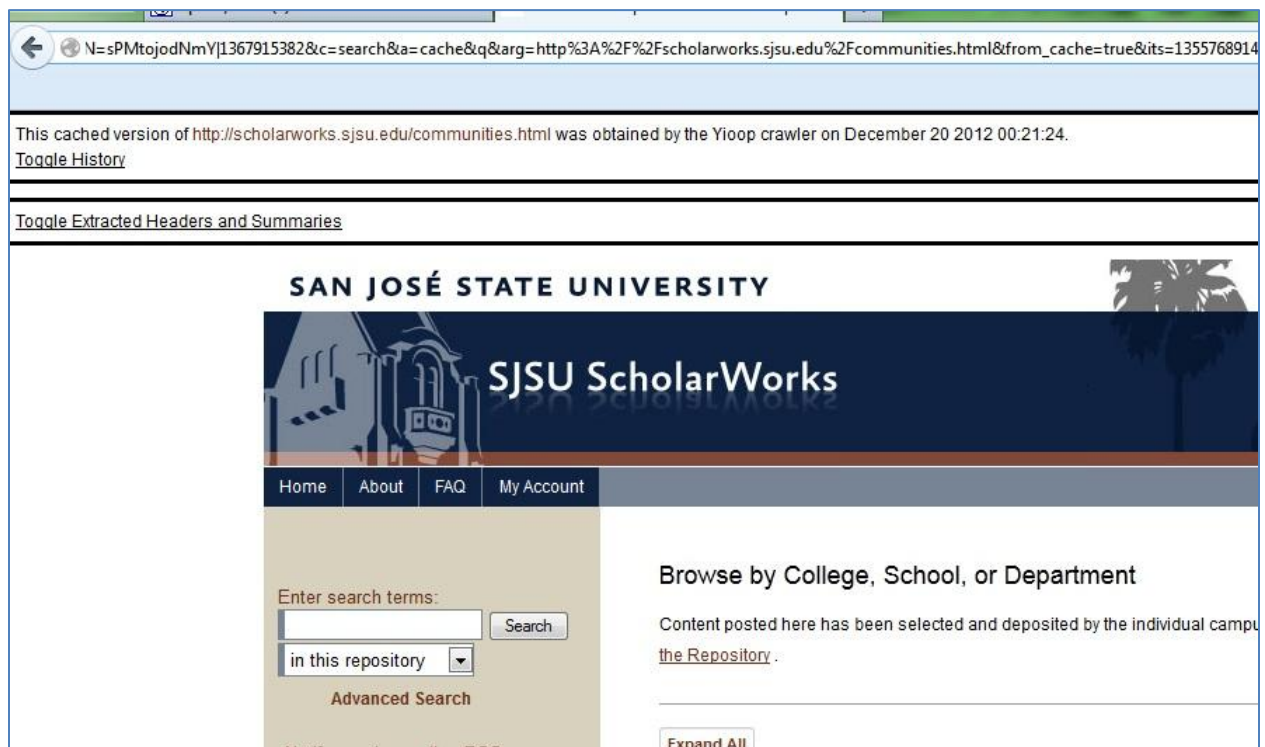


Figure 8: Redirection within the cache

### 3.3 History UI

The history feature combines the work done in steps 3.2 and 3.3. Step 3.1 enabled users to navigate from one cache page to another by changing the timestamp parameter in the link. In this step I created a User Interface for navigating through cached pages. The interface works by first finding links to all cached pages for a given URL. When a cached version of a web page is displayed, links to all cached versions are also displayed. For navigation, the User Interface allows users to select the year and month so that they can choose the time for the cached page. The links display the day and time for each year and month combination. The history feature is controlled by a Toggle link above the extracted headers and summaries link. I developed the history feature using PHP DOM and JavaScript.

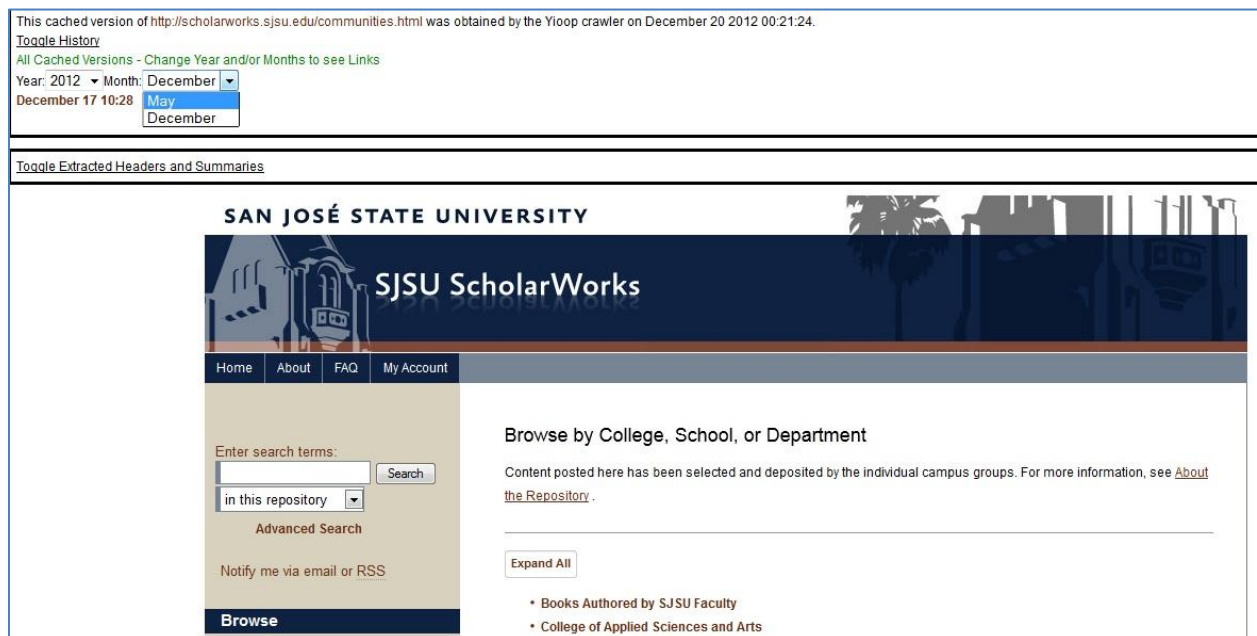


Figure 9: Yioop's History feature

## 4. Entity Tags And Expires Headers

### 4.1 Entity Tags

An Entity Tag or ETag is a unique identifier associated with web resource. It is a part of the HTTP header (HTTP ETag). Whenever a resource with an ETag is modified, a new ETag is generated for the modified resource. This can be used to compare different versions of the same web resource. The difference in ETag values can tell that the versions of the web resource are different. For all resources belonging to the same server, the ETags must be unique. When making a request to a server, the following HTTP headers may be sent to check the version of a particular resource. There are two types of ETags:

1. Strong ETag: Strong ETags are used to check if two resources are the same at every byte (HTTP ETag). For example, ETag: "1cabfJk4c9"
2. Weak ETag: Weak ETags are used to check if two resources are similar. The resources might not be identical. For example, ETag: W/"1cabfJk4c9"

For checking if a resource has changed or not, the following HTTP headers may be sent when making a request for a resource.

1. If-Match: ETag: If ETag matches the ETag value of the resource being requested, the entire resource is returned. Otherwise, a status 402 (precondition failed) is returned.
2. If-None-Match: ETag: If ETag matches the ETag value of the resource being requested, that means the resource has not been modified and a status 302 (not modified) is returned. Otherwise, the entire resource is returned.

### 4.1.1 Experiment With Entity Tags, PHP, and cURL

As mentioned earlier, Yioop uses cURL, a library written in C language, for downloading multiple pages concurrently. The second feature of this project uses ETags for validating cached web pages. As the first towards developing the second feature of this project, I experimented to learn how web pages can be downloaded using cURL and PHP, and also to experiment with different ETag headers.

I wrote a program that downloads a web page and checks if the page has an ETag associated with it. If the ETag is found, the program extracts it. The program then again tries to download the same web page, but with different ETag headers. This experiment helped me understand how various ETag headers work. The cache page validation feature of this project modifies Yioop's Fetcher to use ETags when downloading web pages.

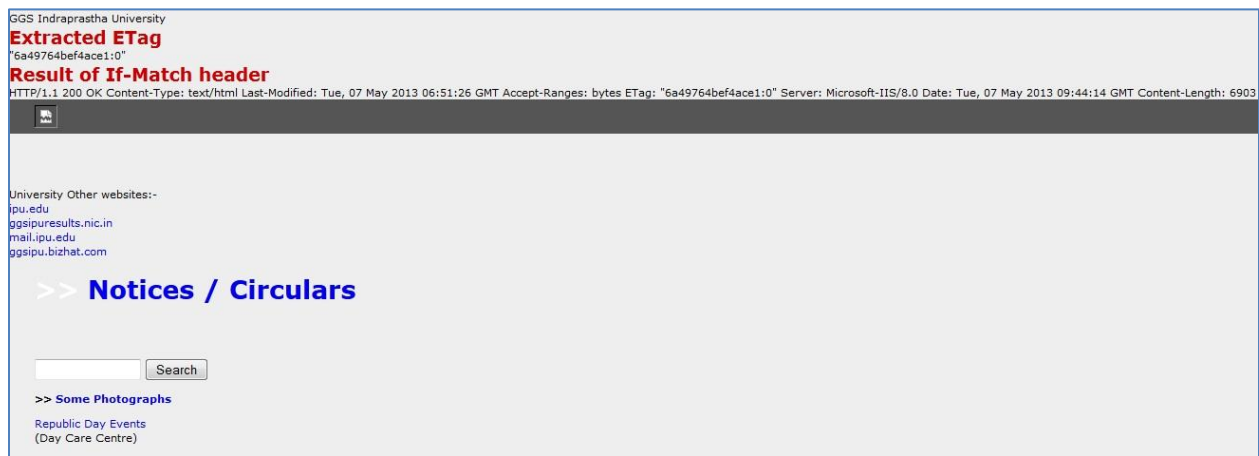


Figure 10: ETag extraction using PHP cURL

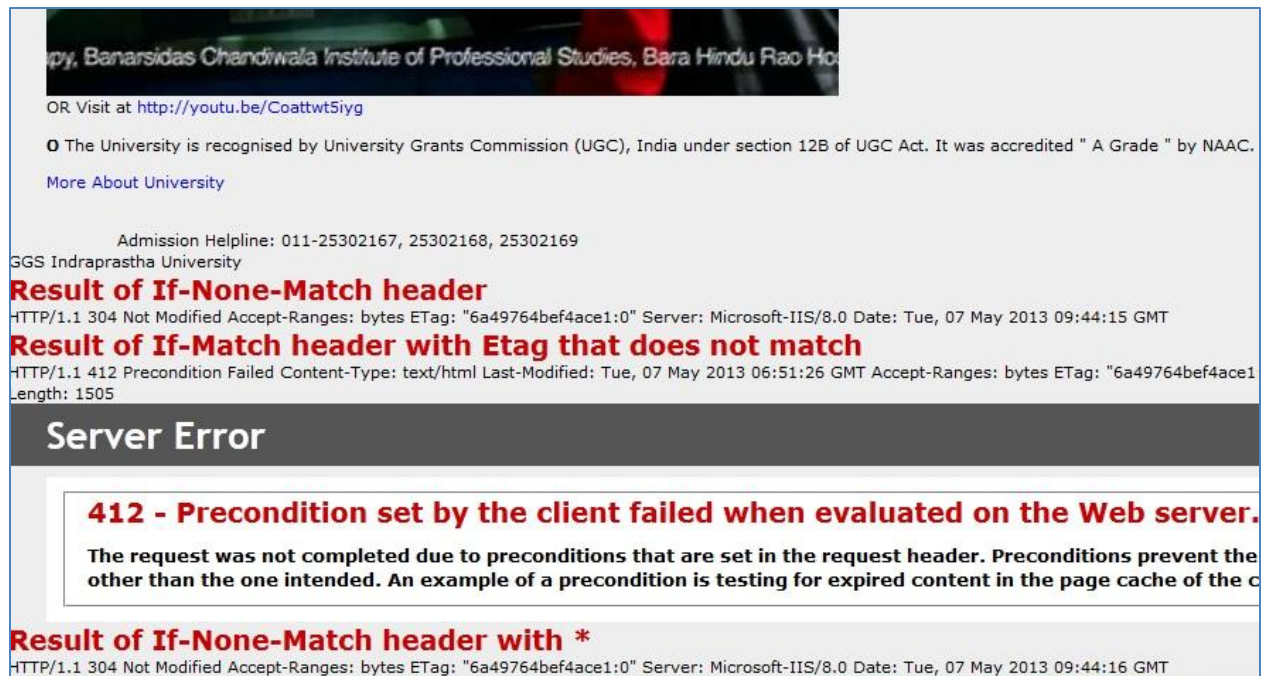


Figure 11: ETag headers in PHP cURL

Figure 11 shows the results of using different ETag headers with PHP and cURL.

## 4.2 Expires Header

The Expires header is also part of HTTP. For a resource, the Expires header has the date after which the resource will expire. The format of the Expires header is as follows (Header Definitions)

Expires: HTTP-Date

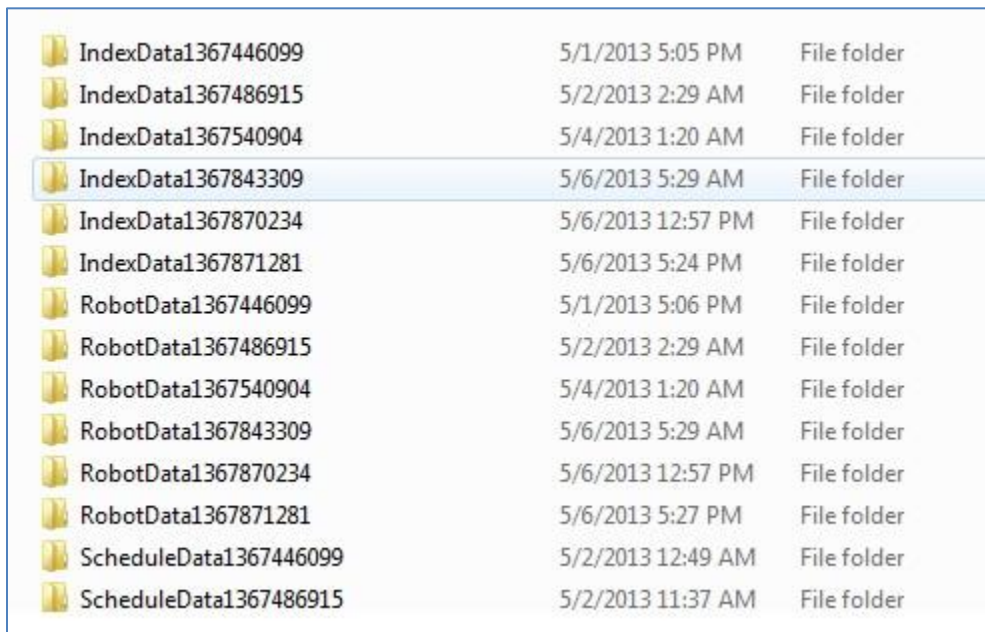
For example, Expires: Sun, 05 May 2013 15:00:00 GMT

The second feature if this project modifies Yioop's Queue Server to check if a web page is still fresh. If yes, then the web page is not scheduled for download.

## 5. Modifying Yioop's Fetcher

After learning about Entity Tags and Expires headers, the next step was to modify Yioop so that it extracts ETags and Expires headers from web pages. As mentioned before, when the Fetcher downloads a web page, it extracts information from the web page such as Content-Type, Last-Modified etc. As the first step, I added the code for extracting ETags and Expires header for a website to the code that downloads the URL. The ETags and Expires headers were then saved with the website summaries, to be processed by the Fetcher.

Based on the size of the downloaded pages, Yioop's Fetcher periodically writes web page summaries, including extracted URLs to the Name Server so that it can be processed by the Queue Server. This data is written to files that are processed by the Queue Server.

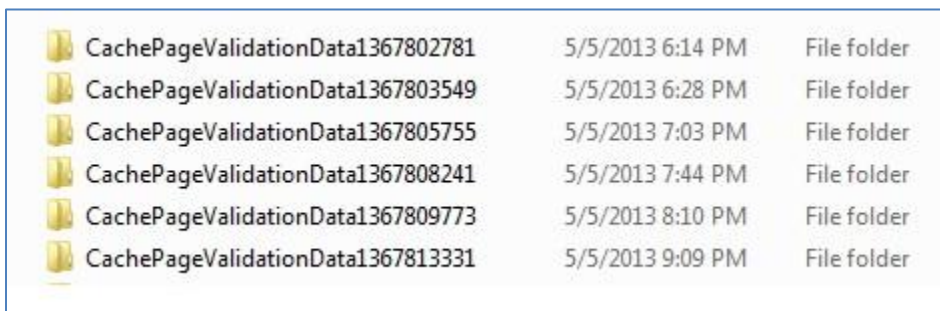


IndexData1367446099	5/1/2013 5:05 PM	File folder
IndexData1367486915	5/2/2013 2:29 AM	File folder
IndexData1367540904	5/4/2013 1:20 AM	File folder
IndexData1367843309	5/6/2013 5:29 AM	File folder
IndexData1367870234	5/6/2013 12:57 PM	File folder
IndexData1367871281	5/6/2013 5:24 PM	File folder
RobotData1367446099	5/1/2013 5:06 PM	File folder
RobotData1367486915	5/2/2013 2:29 AM	File folder
RobotData1367540904	5/4/2013 1:20 AM	File folder
RobotData1367843309	5/6/2013 5:29 AM	File folder
RobotData1367870234	5/6/2013 12:57 PM	File folder
RobotData1367871281	5/6/2013 5:27 PM	File folder
ScheduleData1367446099	5/2/2013 12:49 AM	File folder
ScheduleData1367486915	5/2/2013 11:37 AM	File folder

Figure 12: Yioop Work Directory

Figure 12 shows the contents of the schedules directory within a Yioop instance's working directory. The figure shows Index and Robot data that is written by the Fetcher. Data for different crawl times is written separately.

As shown in Figure 12, the Fetcher sends Robot Data and Index Data back to the Queue Server. Robot Data is used to filter out what paths of the website Yioop is allowed to crawl. The Index data is processed by the Queue Server for building the search index. The Robot data has a lifetime of one day. After one day, the Robot data is deleted and fresh Robot data is downloaded. I modified Yioop's Fetcher so that along with Index and Robot data, Cache page validation data is also sent to the Queue Server.



CachePageValidationData1367802781	5/5/2013 6:14 PM	File folder
CachePageValidationData1367803549	5/5/2013 6:28 PM	File folder
CachePageValidationData1367805755	5/5/2013 7:03 PM	File folder
CachePageValidationData1367808241	5/5/2013 7:44 PM	File folder
CachePageValidationData1367809773	5/5/2013 8:10 PM	File folder
CachePageValidationData1367813331	5/5/2013 9:09 PM	File folder

**Figure 13: Cache page validation data written by Yioop's Fetcher**

When information is extracted from the header of a web page, the modified code checks if there is an ETag and an Expires header. If either is found, the URL, along with the ETag and/or the Expires header is saved so that it can be written back to the Queue Server as cache page validation data. If no ETag and no Expires header is found, the URL is not saved in cache page validation data. The cache page validation data is then appended to Robot data and Index data, and sent to the Queue Server.

## 6. Disk Access Experiment

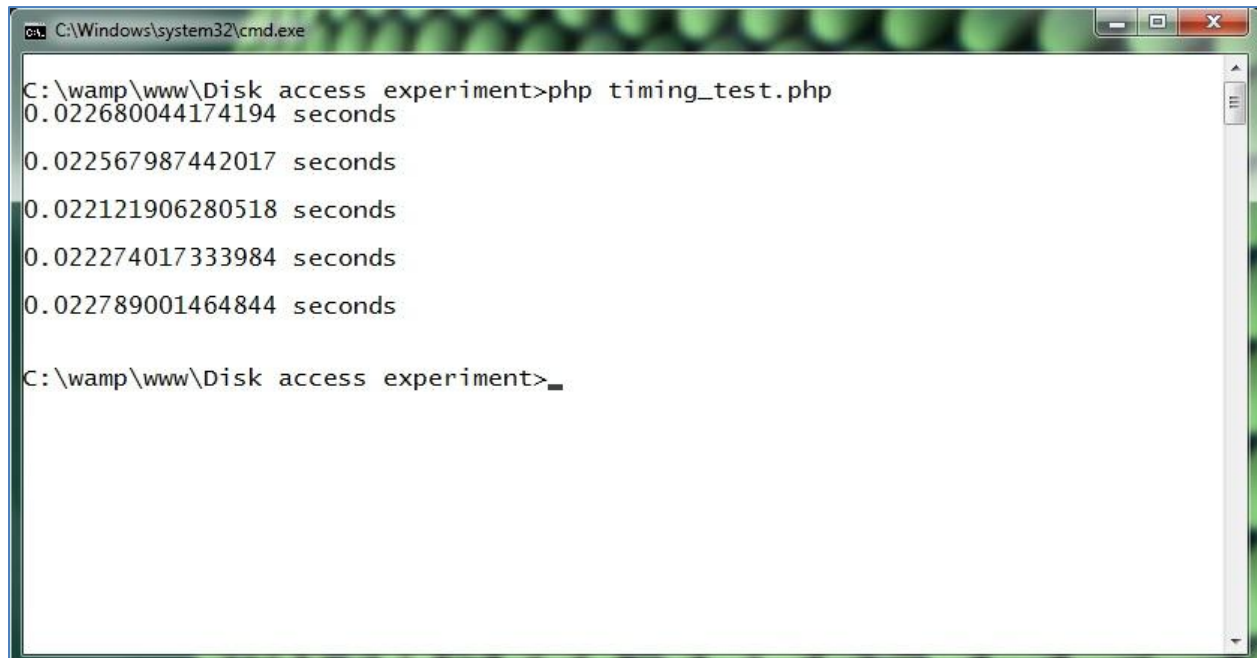
According to (WorlWideWebSize.com), the size of the indexed web is at least 14.37 billion pages. In order to validate a similar or number of cached web pages using by Entity Tags and /or Expires tags, it is important that the solution be efficient in both storing the cache page validators (ETags or Expires) headers, and also fast in lookup so that the crawl speed is not affected.

Let us assume that the average length of an ETag and a URL is 32 bytes. Then, in order to store 1 billion URL-ETag pairs, total space required would be  $(32 + 32) * 1,000,000,000$  bytes that is 64,000,000,000 bytes or 64 GB. There are limits on the maximum file size allowed and this limit varies from one file system to another. For example, the ext2 and ext3(second and third extended file systems) allow the maximum file size to be between 16 GB and 2 TB. Sun's ZFS on the other hand, allows maximum file size to be up to 16 Exabytes. The cache page validation feature, apart from being able to work on modern file systems, should also be able to work on file systems where the maximum allowable size is less than the values mentioned above. According to (Comparison of file systems), the lowest maximum allowable file size is 2 GB for file systems like HFS and FAT16. Therefore I chose 2 GB as the size for my disk access experiment.

Using 2 GB as the file size, I conducted the following experiment to estimate a lower bound on the time it takes to look-up values from a file. I populated a file with values written with 4 byte intervals. The values were randomly generated, and were between 1 and 1 million. The first value was written at byte 0, the next one at byte 0 + length of first entry + 4 bytes, and so on. The file was populated to 2 GB.

When the Queue Server produces a fetch batch, the fetch batch consists of 5000 URLs. Thus, the second feature of this project would require 5000 lookups during scheduling. The

lookup should be fast enough so that the crawl speed is not affected. I generated 5000 random numbers along with their offsets within the file that was populated earlier. I did a lookup for those 5000 values and recorded the total time taken for the lookup.



```
C:\Windows\system32\cmd.exe
C:\wamp\www\Disk access experiment>php timing_test.php
0.022680044174194 seconds
0.022567987442017 seconds
0.022121906280518 seconds
0.022274017333984 seconds
0.022789001464844 seconds
C:\wamp\www\Disk access experiment>
```

**Figure 14:Disk access experiment results**

Figure 14 shows the results of the disk access experiment. The average time to do 5000 lookups for a file containing values at 4 byte offsets is 0.022 seconds. In order to do 5000 lookups on multiple files would require more time. Thus, the experiment helped me estimate the least amount of time required for doing 5000 lookups.

## 7. Data Structure for Cache Page Validators

After estimating the lower bound for doing 5000 lookups, the next step was to select an appropriate data structure for storing cache page validation data. As discussed earlier, 5000 lookups on a huge volume of data would require lookups in multiple files. Therefore, an efficient disk-based data structure would be required to save such large volumes of data and also provide fast lookups.

### 7.1 B-Trees

A B-Tree is a tree data structure that has a high branching factor. The branching factor can be as high as 1000. The branching factor for a B-Tree is decided by a number called the minimum degree of a B-Tree (Thomas H. Cormen, 2001). If the minimum degree of a B-Tree is  $\text{min\_deg}$ , then

1. Minimum number of keys for nodes other than the root node =  $\text{min\_deg} - 1$
2. Minimum number of child nodes for nodes other than the root node =  $\text{min\_deg}$
3. Maximum number of keys for any node =  $2 * \text{min\_deg} - 1$
4. Maximum number of links for any node =  $2 * \text{min\_deg}$

Consider a B-Tree with order 501. Then,

Maximum number of keys in root =  $2 * 501 - 1 = 1001$

Maximum number of children for root node = 1,002

Therefore, at root level, 1001 keys can be saved

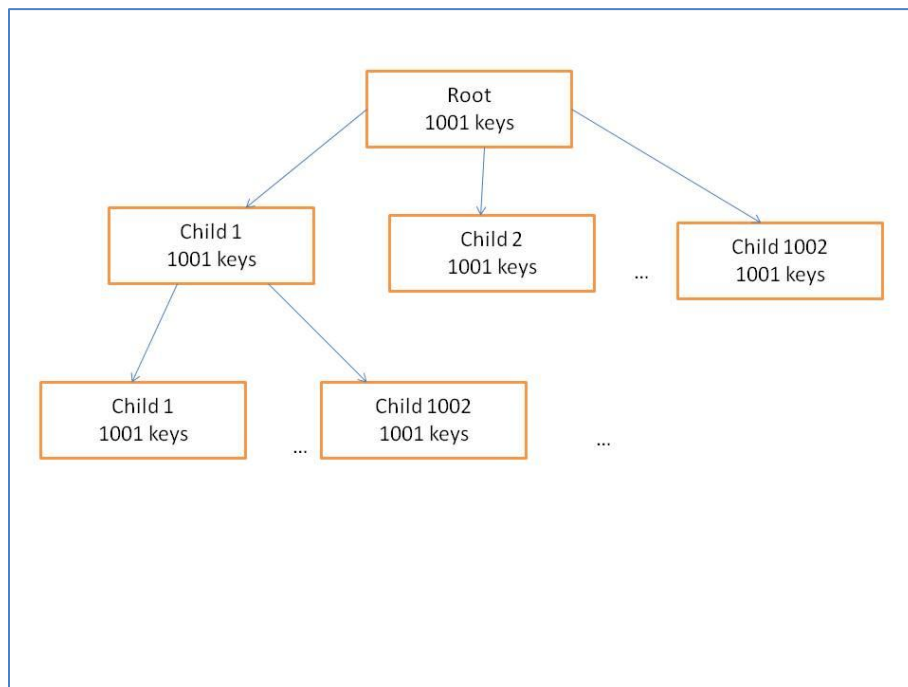
Similarly level 1, each one of 1002 nodes will have 1001 keys. Therefore, number of keys after completely filling up the first two levels =  $1002 * 1001 = 1,003,002$  keys

For level 2, number of keys =  $1003003 * 1001 = 1,004,006,003$  keys

Level 3, number of keys =  $1004006004 * 1001 = 1,005,010,010,004$  keys

Thus, a B-Tree with minimum degree 1001 and height 3 can easily accommodate 100 billion keys. If the root node is kept in memory and each node is a file, then in order to do one lookup, we would access only 3 files. Going back to our assumption about the length of URLs and ETags, the size of a node would be 64 KB.

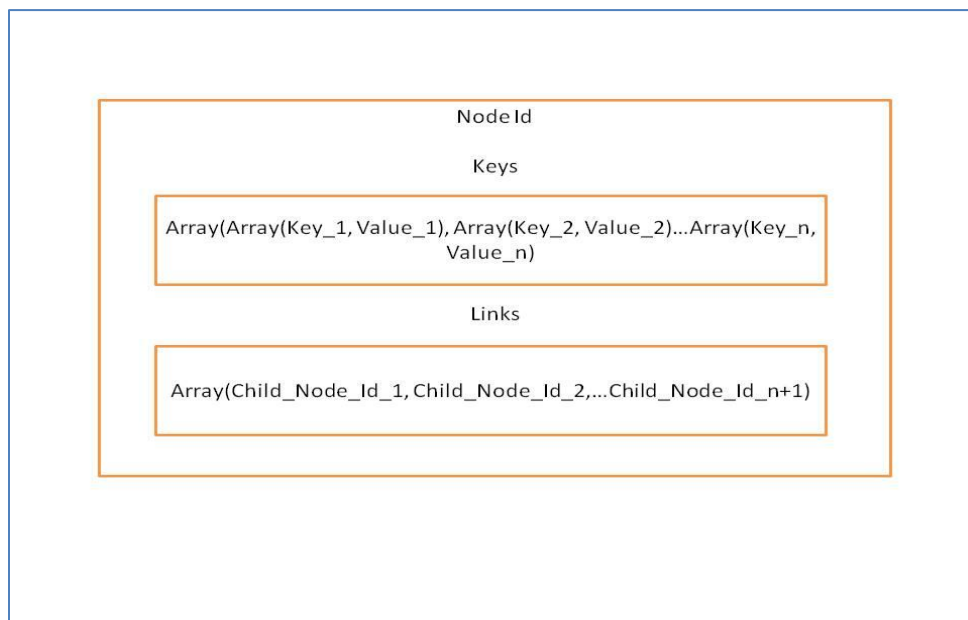
Due to the above mentioned reasons, I chose the B-Tree data structure for implementing the cache page validation feature. The B-Tree definition used in this project is defined in (Thomas H. Cormen, 2001). Each node, whether internal, or leaf, stores the complete information which is keys and values. Internal nodes also store links to child nodes.



**Figure 15: A B-Tree with minimum degree 501**

## 7.2 B-Tree Implementation in PHP

For the cache page validation feature of this project, I implemented a B-Tree for storage and lookup of cache page validation data. As explained above, I chose the minimum degree to be 501. I also made use of Yioop's hash function for generating keys for the B-Tree. Each node of the B-Tree consists of an array of key-value pair. The key-value pair consists of the hash of the URL, calculated using Yioop's hash function, and the ETag/ Expires header as the value. Each internal node of the B-Tree also stores the links to child nodes. Each node has an integer id. Also, each internal node saves links to child nodes. The links are maintained using the node ids.



**Figure 16: B-Tree node for cache page validation feature**

### 7.3 Augmenting Yioop's Queue Server

As the next step, I modified Yioop's Queue Server to include the B-Tree implemented by me in the previous step. During a web crawl, the Queue Server periodically checks if the Fetcher has written Robot and Index data to the Name Server. If found, the Queue Server incrementally adds the Index Data to the search index for the current crawl timestamp. In section 5, I modified Yioop's Fetcher to separately send Cache Page Validation Data along with Index and Robot Data. In this step, I modified the Queue Server to also check for Cache Page Validation Data when it checks for Robot and Index Data. If found, the Queue Server reads the data, and extracts the URL, ETag, and Expires header. The Queue Server then calculates the hash of the URL using its own hash function. The hash of the URL serves as the key for each key-value pair and, the ETag and Expires headers form the value. The hash of the URL, along with the ETag and the Expires header are inserted into the B-Tree. A separate B-Tree is used for each index timestamp and is saved in the work directory.







	CachePageValidators1367802781	5/5/2013 6:15 PM	File folder
	CachePageValidators1367803549	5/5/2013 6:30 PM	File folder
	CachePageValidators1367805755	5/5/2013 7:32 PM	File folder
	CachePageValidators1367808241	5/5/2013 7:57 PM	File folder
	CachePageValidators1367809773	5/5/2013 8:40 PM	File folder
	CachePageValidators1367813331	5/6/2013 2:45 AM	File folder

Figure 17: Cache page validators written by Queue Server









	28	5/6/2013 4:58 AM	TXT File	72 KB
	29	5/6/2013 4:58 AM	TXT File	73 KB
	30	5/6/2013 4:58 AM	TXT File	72 KB
	31	5/6/2013 4:58 AM	TXT File	69 KB
	32	5/6/2013 4:58 AM	TXT File	68 KB
	33	5/6/2013 4:58 AM	TXT File	66 KB
	count	5/6/2013 2:45 AM	TXT File	1 KB
	root	5/6/2013 5:02 AM	TXT File	4 KB

Figure 18: B-Tree nodes for cache page validators

## 7.4 Implementing the Cache Page Validation Feature

After modifying the Queue Server, the next step was to integrate the modified Queue Server with the modified fetcher program. I modified Queue Server to include lookup of ETags and Expires headers when the fetch batch is made. If an ETag is found, it is sent to the Fetcher along with the associated URL when sending the fetch batch. The Fetcher, on getting an ETag, appends the HTTP header with the If-None-Match ETag header before downloading the web page. On the other hand, if the Queue Server finds Expires Tags, it checks if the page is still fresh. If yes, the URL is not scheduled. Otherwise, the URL is scheduled.

### 7.4.1 Queue Server pseudo-code for storing Cache Page Validators

```

1. Lookup cachePageValidationData in Work Directory
2. if cachePageValidationData is found
3.     data = read(cachePageValidationData)
4.     array(url, array(etag, expires)) = data
5.     url_hash = hash(url)
6.     btree->insert(array(url hash, array(etag, expires)))

```

Following is a screen-shot showing the contents of a B-Tree node with Cache Page Validation Data.

```
0:4:"Node":5:{s:2:"id";s:4:"root";s:7:"is_leaf";b:0;s:5:"count";i:31;s:4:"keys";a:31:{i:0;i:1;s:4:"etag";s:27:"ca3f7c-1447-4dbd37129fd87";s:7:"expires";i:-1;}}i:1;a:2:{i:0;s:11:"2vt{i:0;s:11:"5FMwxRbXbls";i:1;a:2:{s:4:"etag";s:18:"a3lcffe2e0cc10";s:7:"expires";i:-1;}}{s:4:"etag";i:-1;s:7:"expires";i:1367815886;}}i:4;a:2:{i:0;s:11:"9cg062mct9o";i:1;a:2:{s:4:i:0;s:11:"C0feylRgowk";i:1;a:2:{s:4:"etag";i:-1;s:7:"expires";i:1367816057;}}i:6;a:2:{i:0{s:4:"etag";s:34:"cfcd208495d565ef66e7dff9f98764da";s:7:"expires";i:1370241664;}}i:7;a:2i:8;a:2:{i:0;s:11:"IMnMdF0URoY";i:1;a:2:{s:4:"etag";s:19:"67991fbd76a6c910";s:7:"expires{s:4:"etag";i:-1;s:7:"expires";i:1367818300;}}i:10;a:2:{i:0;s:11:"Mdws7Zcis-w";i:1;a:2:{s:i:11;a:2:{i:0;s:11:"0aNRgSRuKc8";i:1;a:2:{s:4:"etag";s:19:"537656982728cal0";s:7:"expiri{s:4:"etag";i:-1;s:7:"expires";i:1367816677;}}i:13;a:2:{i:0;s:11:"ShM7MmxK-W4";i:1;a:2:{s:i:10;s:11:"UmcWJiTW5oY";i:1;a:2:{s:4:"etag";i:-1;s:7:"expires";i:1367816238;}}i:15;a:2:{i:i:16;a:2:{i:0;s:11:"Yl1XoxWlHF4";i:1;a:2:{s:4:"etag";s:18:"d3b5b982728cal0";s:7:"expires{s:4:"etag";i:-1;s:7:"expires";i:1367822500;}}i:18;a:2:{i:0;s:11:"bWaeZ1affFw";i:1;a:2:{s:i:10;s:11:"dUrZIZa64uQ";i:1;a:2:{s:4:"etag";i:-1;s:7:"expires";i:1367815804;}}i:20;a:2:{i:
```

#### 7.4.2 Queue Server pseudo-code for Lookup of Cache Page Validation Data when producing fetch batches

```
1. While creating fetch batch of URLs to be downloaded
2.   for each URL selected from priority queue
3.     url_hash = hash(url)
4.     array(url_hash, array(etag, expires)) = btree->lookup(url_hash)
5.     if etag found and expires found
6.       if current timestamp < expires
7.         continue
8.       else
9.         append etag to url
11.    else if only etag found
12.      append etag to url
14.    else if only expires found
15.      if current timestamp < expires
16.        continue
17.    add url to fetch batch
```

### 7.4.3 Fetcher pseudo-code for Fetcher for handling Cache Page Validation

#### Data

Fetcher pseudo-code for downloading a web page using cURL

```
function download(url, header)
1. Initialize cURL with url
2. add headers to cURL HTTP header
3. downloaded_page = execute cURL request
4. return downloaded_page
```

Fetcher pseudo-code downloading web pages, extracting cache page validation data, and sending the data to the Queue Server

```
1. Lookup scheduleddata
2. if scheduleddata is found
3.     urls = read(scheduleddata)
4.     for each url in urls
5.         if url has etag appended to it
6.             url_header = concatenate("If-None-Match:", etag)
7.             downloaded_page = download(url, url_header)
8.             if downloaded_page has etag and expires
9.                 add array(url, array(etag, expires)) to list of found sites
10. if sending Index and Robot Data
11.     Check if found sites have etag and expires
12.     CachePageValidationData = array(array(url1, array(etag1, expires1)),
        array(url2, array(etag2, expires2), ...))
13.     send IndexData, RobotData, and CachePageValidationData to Queue Server
```

## 8. Experiments

The aim of the second feature of this project was to experiment with a different feature for cache page validation that uses Entity Tags and Expires headers. This section describes the experiments done with the implemented cache page validation feature. The first experiment determines the impact of the cache page validation feature on the crawl speed. The second experiment measures savings in bandwidth due to ETags and Expires headers. The experiments were conducted in a multiple Queue Server setting using two instances of Yioop. The tests were conducted on a single machine.

### 8.1 Testing the effect on fetch batch creation time

As mentioned earlier, the Queue Server maintains a priority queue of URLs to be crawled. The Queue Server produces a fetch batch by selecting URLs from the priority queue. For the first experiment, I recorded the time taken to create fetch batches. First, I crawled 100,000 pages using Yioop, but without the cache page validation feature. Then, I crawled 100,000 pages again using the cache page validation feature. The page re-crawl frequency was set to 3 hours.

The following table gives a comparison of the time taken to write a fetch schedule, measured in seconds for both the Queue Servers.

**Figure 19: Table comparing fetch schedule write times**

	Yioop without cache page validation	Yioop with cache page validation
<b>Queue Server 1</b>		
1.	0.0051290000000001	0.001525
2.	0.040342	0.004855
3.	0.001158	0.075212
4.	0.026641	0.045726

5.	0.66937	0.033845
6.	0.680713	0.962385
7.	0.638508	1.230332
8.	0.629087	0.957314
9.	0.639234	15.306216
10.	0.669248	15.788489
11.	0.762741	21.100715
12.	0.74138	24.837989
13.	0.677873	18.193379
14.	2.677664	20.571448
15.	1.67749	32.711314
16.	2.784278	23.794299
17.	2.443083	17.042238
18.	2.640033	26.596688
19.	0.015935	23.030196
20.	0.008443	20.791269
21.	0.04249	23.682045
<b>Queue Server 2</b>		
1.	0.023429	0.010415
2.	0.681112	0.05673
3.	0.230002	0.001767
4.	0.670079	0.942093
5.	0.682405	0.605549
6.	0.677915	0.955226
7.	0.693039	20.635931

8.	0.663932	20.970049
9.	0.68279	20.368085
10.	0.678535	20.551323
11.	0.72232	16.540064
12.	0.666994	18.379397
13.	0.687878	27.736767
14.	0.760716	24.651495
15.	0.730754	18.515776
16.	0.729345	26.390242
17.	1.465243	19.291583
18.	0.962642	16.473845
19.	0.861857	16.899316
20.	1.715415	15.541988
21.	0.013842	16.077356

From the table, it can be seen that the least time taken for a B-Tree lookup was **15.3 seconds**, and the largest time taken was **32.7 seconds**. The average times for Queue Server 1 and 2 are calculated below.

#### **For Queue Server 1**

Average time taken for writing fetch batch schedule without cache page validation: **0.87596381**

Average time taken for writing fetch batch schedule with cache page validation: **13.65511805**

#### **For Queue Server 2**

Average time taken for writing fetch batch schedule without cache page validation: **0.714297333**

Average time taken for writing fetch batch schedule with cache page validation: **14.36166652**

Total Average time taken without cache page validation: **0.7951305715**

Total Average time taken with cache page validation: **14.008391625**

**Conclusion:** For a 100,000 page crawl, the cache page validation feature was on average slower by 14 seconds due to B-Tree lookup.

## **8.2 Determining effect of cache page validation on bandwidth**

The aim of the second experiment was to determine the savings in bandwidth. For this experiment, I recorded the URLs that were not scheduled by the Queue Server because of the expires timestamp being greater than the current timestamp, and the URLs that return HTTP status 304 Not Modified on download due no change in ETag. Then, using cURL, I downloaded the web pages to determine their size. 100,000 web pages were crawled using the same settings used in 8.1.

### **Findings:**

1. Total time taken to crawl 100,000 pages = **8 hours**
2. Total number of entries of cache page validation data stored in B-Tree = **54,939**
3. Total number of URLs that were not re-crawled due to ETags and Expires headers = **412**
4. Savings in MB for the URLs that were not re-crawled = **15.64 MB**
5. Most URLs that resulted were not crawled due to ETag/Expires were image files.

**Conclusion:** For the 100,000 web page crawl with a re-crawl frequency of 3 hours, the total time taken was 8 hours. A re-crawl frequency of 3 hours means that after every 3 hours, if a URL that has been already downloaded is encountered by the Queue Server, it will be scheduled again. From the figures mentioned above, it can be seen that during the 8 hour crawl,

only 412 URLs were re-crawled and resulted in a total of 15.64 MB. The number of URLs re-crawled URLs is a small fraction of the total URLs with cache page validation data stored in the B-Tree (**0.7%**). This small fraction resulted in total 15.64 MB savings. Therefore, based on the re-crawl policy, as the number of re-crawled URLs increases, the savings will be more. Also, a large number of URLs that were not re-downloaded were images. This is useful for both Yioop and the source of the image as bandwidth is saved for both when images are not re-downloaded. This holds true for other types of documents such as videos and PDF documents.

## 9. Conclusion

Modern search engines often maintain caches of web pages and display the link to the cached version when displaying the search results. But, only the latest cached version is displayed. The Internet Archive maintains an archive of the web by taking periodic snapshots of web pages but does not allow users to do a full-text search on the archived pages. Also, content of the web keeps on changing and it is necessary for web crawlers to re-crawl pages to ensure that their index is fresh. But, re-crawling web content that has not change can lead to wastage of bandwidth. Google uses If-Modified-Since for validating its cached pages.

This project implements two new cache-related features for Yioop search engine. The first feature is a history feature that allows users to see all versions of cached pages maintained by Yioop. This feature is an improvement over modern search engines as the entire cache history is available to users. The feature also allows users to do full-text searches on cached pages which is not available with the Internet Archive. The second feature of this project uses Entity Tags and Expires headers to validate cached copies of web pages downloaded by Yioop's crawler. The feature uses B-Trees for storage and lookup of cache page validation data.

This history feature acts an archive of web pages cached by Yioop's crawler and will provide users with a new functionality capable of performing searches on the entire history of cached pages. The cache page validation slows down the Queue Server but is a promising feature, and a good start towards developing more efficient ways of using ETags and Expires headers for cache page validation. From the experiment, it can be seen that URLs re-crawled using cache page validation feature resulted in savings in bandwidth. The future work includes experimenting with B+ Trees that store all the data in the leaves, experimenting with different crawl policies to determine impact on savings in bandwidth, and testing the cache page validation feature on a multiple Queue Server setting on multiple machines.

## Bibliography

*About /robots.txt.* (n.d.). Retrieved May 6, 2013, from robotstxt.org:  
<http://www.robotstxt.org/robotstxt.html>

*Comparison of file systems.* (n.d.). Retrieved May 6, 2013, from Wikipedia:  
[http://en.wikipedia.org/wiki/Comparison\\_of\\_file\\_systems](http://en.wikipedia.org/wiki/Comparison_of_file_systems)

*Header Definitions.* (n.d.). Retrieved May 6, 2013, from Worl Wide Web Consortium:  
<http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>

*Hierarchical File System.* (n.d.). Retrieved May 6, 2013, from Wikipedia:  
[http://en.wikipedia.org/wiki/Hierarchical\\_File\\_System](http://en.wikipedia.org/wiki/Hierarchical_File_System)

*HTTP ETag.* (n.d.). Retrieved May 6, 2013, from Wikipedia:  
[http://en.wikipedia.org/wiki/HTTP\\_ETag](http://en.wikipedia.org/wiki/HTTP_ETag)

*Internet Archive.* (n.d.). Retrieved May 3, 2013, from Internet Archive: <http://archive.org/>

Pollett, C. (n.d.). *Yioop Documentation Version 0.94.* Retrieved May 3, 2013, from SeekQuarry:  
<http://www.seekquarry.com/?c=main&p=documentation>

Rackley, M. (2010). Internet Archive. *Encyclopedia of Library and Information Sciences, Third Edition* , 1:1, 2966 - 2976.

Stefan Büttcher, C. L. (2010). *Information Retrieval: Implementing and Evaluating Search Engines.* Cambridge, Massachusetts London, England: The MIT Press.

Thomas H. Cormen, C. E. (2001). *Introduction to Algorithms Second Edition.* Cambridge, Massachusetts: The MIT Press.

*WARC, Web ARChive File Format.* (n.d.). Retrieved May 6, 2013, from digitalpreservation.gov:  
<http://www.digitalpreservation.gov/formats/fdd/fdd000236.shtml>

*Web Cache.* (n.d.). Retrieved May 3, 2013, from Wikipedia:  
[http://en.wikipedia.org/wiki/Web\\_cache](http://en.wikipedia.org/wiki/Web_cache)

*WorlWideWebSize.com.* (n.d.). Retrieved May 6, 2013, from WorlWideWebSize.com:  
<http://www.worldwidewebsize.com/>