2017 International Conference on Advances in ICT for Emerging Regions (ICTer): 165 - 171

Detection of Change Frequency in Web Pages to Optimize Server-based Scheduling

Lakmal Meegahapola[#1], Roshan Alwis[#2], Eranga Nimalarathna[#3], Vijini Mallawaarachchi[#4], Dulani Meedeniya[#5], Sampath Jayarathna[*6]

[#]*Department of Computer Science & Engineering, University of Moratuwa, Sri Lanka*

{[1]lakmalbuddikalucky.13, [2]alwisroshan.13, [3]eranga.13, [4]vijini.13, [5]dulanim}@cse.mrt.ac.lk

[*]*Department of Computer Science, California State Polytechnic University, Pomona, CA 91768*

[6]ukjayarathna@cpp.edu

***Abstract*—The Internet at present has become vast and dynamic with the ever increasing number of web pages. These web pages change when more content is added to them. With the availability of change detection and notification systems, keeping track of the changes occurring in web pages has become more simple and straightforward. However, most of these change detection and notification systems work based on predefined crawling schedules with static time intervals. This can become inefficient if there are no relevant changes being made to the web pages, resulting in the wastage of both temporal and computational resources. If the web pages are not crawled frequently, some of the important changes may be missed and there may be delays in notifying the subscribed users. This paper proposes a methodology to detect the frequency of change in web pages to optimize server-side scheduling of change detection and notification systems. The proposed method is based on a dynamic detection process, where the crawling schedule will be adjusted accordingly in order to result in a more efficient server-based scheduler to detect changes in web pages.**

***Keywords*—Internet; web page; change detection and notification systems; crawling; change frequency; server-side scheduling;**

I. INTRODUCTION

The World Wide Web (WWW) is expanding at a rapid rate with the availability of various tools and services to create and update web content. Keeping track of the changes occurring in web content has become a challenge, as there is a vast number of web pages and these web pages keep on changing frequently. If people are interested in certain web pages and have the need to know when changes occur, they have to manually refresh pages periodically to see whether any changes have occurred. This is a time consuming and tedious process. As a result, users may miss certain changes if they were unable to check the web pages frequently as required.

With the introduction of change detection and notification systems, the process of keeping track of changes to web pages has become less time consuming. Some of the most popular and widely used change detection and notification services include Google Alerts [1], ChangeDetection [2] and Follow That Page [3]. These automated change detection systems can identify changes by providing the user the option to provide his or her email address and the list of Uniform Resource Locators (URLs) of the web pages, which they want to keeping track of. These systems send email notifications to users when changes occur to the web pages in which the users have an interest. The monitoring process of these applications varies from service to service and ranges from monitoring a single web page at a time to collections. The content monitored can include text, links, images, documents and layouts.

There are many issues related to current tracking systems due to the increasing number of web pages being registered to track. For example, performance issues related to change detection and tracking system architectures have arisen [4], [5]. Some change detection systems keep track of websites based on static time intervals and therefore, certain critical changes to web pages may not be notified to users as soon as they occur. On the other hand, if change detection systems check web pages frequently, this might cause for wastage of computational and temporal resources of the system.

One of the important research problems at present is the scheduling process of the visits to un-visited pages within the hierarchy of links. Different web pages updated at different temporal schedules and the change frequency may differ from web page to web page. Some web pages (e.g., news sites and blogs) may update frequently, whereas some web pages may update rarely (e.g., wiki pages). Hence, there is a need for a dynamic mechanism to detect the frequency at which the changes occur in web pages and create checking schedules, so that the web pages are crawled efficiently. Users will be able to get updates immediately after the changes occur and this will enable them to receive notifications as soon as they occur. This will ensure the optimum usage of computational resources and time without any wastage.

This paper presents a mechanism to detect the change frequency of web pages and to optimize the server-side scheduling of change detection and notification services. Section II outlines the related work carried out in the area of interest. Section III describes the proposed design and the architecture of the solution. Section IV explains the methodology which we carried out and Section V presents the obtained experiment results. Finally, Section VI concludes the paper with the inferences obtained from the results, emphasizes on the importance of the research being conducted and discusses some of the future research directions.

II. BACKGROUND

A wide range of related literature is available on web crawler scheduling [6], [8], [9]. A comparative study of scheduling strategies for web crawling is described in [6]. Here, the authors have crawled 3.5 million pages from over 50,000 web sites. Using the data collected, they have created a web graph and run a simulator using different scheduling policies. A heap priority queue has been used to represent

sites as nodes. For each site-node there is another heap representing the web pages in each site. The scheduling process has been divided in to two parts: ordering the queue of web sites (long-term scheduling) and ordering the queues of web pages (short-term scheduling). The authors have considered five long term scheduling strategies, namely o*ptimal, depth, length, batch* and *partial.* In *optimum* strategy, the crawler visits the web pages according to the order of their PageRank [7] value. In *depth* strategy, the crawler visits web pages according to their breath-first ordering. *Length* strategy sorts the web pages according to their depth and priority is given according to the number of pages. Where as in *batch* strategy, the crawler downloads web pages as batches and the PageRank algorithm is run on these batches to sort the pages according to the PageRank value. Moreover, *partial* strategy is somewhat similar to the *batch* strategy but new pages are given a temporary PageRank value after the algorithm is run for each batch. Results have shown that simple crawling strategies are sufficient to retrieve web pages with better bandwidth utilization. However, these strategies lack the ability to determine how often web pages change.

An interesting work related to the comparison of scheduling algorithms for domain specific web crawling is described in [8]. The authors have concentrated on scheduling algorithms which determines the order of crawling URLs collected by the crawler. Four different algorithms, *depth first search*, *breadth first search*, *best first search* and *best n-first search* are compared using several metrics. The algorithms have been evaluated based on how much relevant links were found compared to the iteration number of algorithm in the early phase of crawling. The algorithm is considered as better, if it returns more relevant links. However, algorithms may affect by issues such as efficiency as they keep on crawling even though any changes have not occurred in the web pages being crawled.

An approach for distributed content aggregation and change detection for web content using Bloom Filters is described in [9]. In this work, consumers get the templates and start URL to crawl from a queue, which are updated by a scheduler on a periodic basis and run the crawler on the working machine. Once crawled, the web crawlers will load the content to another queue and this process is repeated. This method distributes the work among consumers running crawlers in parallel and finally the results are aggregated by improving the efficiency of the crawling mechanism. This also claims to reduce the coupling of spiders to machines allowing them to operate in distributed networks. Hash key for each visited URL will be loaded to the bloom filter and when a new URL is found, it is checked with the bloom filter entry. This ensures that the crawling mechanism will not repeat in a loop. However, certain limitations can be caused by the use of bloom filters. Users may not know size of the available storage space and lookup times can be inconvenient when many web pages are crawled. Furthermore, the crawlers keep on crawling without an extract track of the change frequency of web pages, which can impact on the performance of the system.

In several related studies [10], [11], the Poisson model has been identified as an important element for the estimation of web page changes in a given time interval. The work done in [11], states that the number of updates that occurs within a certain time satisfies a Poisson distribution. *Simhash* technique has been used to identify changes occurred in a web page, where the distance between *simhash* values are considered to measure similar content. A model has been proposed to compute the rate of change of a given web site using a simple estimator (dividing the total number of changes observed by the total time observed). However, the question of whether all the changes that have occurred are useful to the users is still unanswered. If changes have not occurred or the occurred changes are not relevant, the crawling effort will be useless. Hence, more vigorous methods are necessary to detect the frequency of relevant change occurrences in web pages.

Most of the currently available change detection and notification systems have pre-defined intervals to check web pages, which is not optimal as the user has no clear indicator on the change frequency of web pages. Follow That Page [3] is one of the widely used online web page monitoring services. It provides 20 daily checks and one hour check per each user. The premium account provides users with 1000 daily checks, 20 hourly checks, five 10-minute checks and 100 weekly checks. The free account is limited to 1 hourly check per each user and this is a disadvantage, if the user has to track more than one frequently changing web page. Moreover, ChangeDetect [12] is another popular online web page monitoring service where users can monitor relevant changes. Saved web pages are automatically checked once a week and additionally users can request on demand web page change detection with a limited number of requests (one request per day). Change detection for registered users can be scheduled to check every 24 hours or every 12 hours and on demand detection is available without any limitation. The major drawback of these systems is that users have no idea on how often web pages change. Thus, they randomly specify the checking interval out of the pre-defined set of checking intervals, which can result in crawls, even though no changes have occurred. Furthermore, these systems have no mechanism to evaluate how often web pages change and they keep on crawling according to the pre-defines static schedule. This results in inefficient use of temporal and computational resources of the change detection systems.

As discussed above, researches have been carried out regarding how to increase the overall efficiency of the change detection process of web pages using efficient algorithms for server-side change detection. Many tools are available to provide change detection and notification services as well. An improved scheduling mechanism can provide better results in improving the change detection of web pages in server-side architectures. In order to address the above discussed issues, this paper proposes a mechanism to detect the change frequency of web pages in order to make server-side change detection process more efficient and robust.

III. SYSTEM ARCHITECTURE

A. Overview

This section describes the architecture of the change frequency detection algorithm for server-side change detection systems to optimize the scheduling of web pages. The proposed architecture is based on the content and layout changes of web sites as they are the main types that change frequently in web pages that are relevant to the users. The solution is designed in a scalable manner to cater many web pages.

B. *Frequency Detection*

This system contains a web application with a web service, where users can enter the websites that they want to keep track of and the multi-threaded server, which crawls web pages to detect changes.

Consider the following example. When a new user uses this web application, he or she includes a web site named *x* to the change detection process. If *x* is a web page which is already tracked by the servers, priority of the web site would increase and the new user will be associated with *x,* so that the user would get notifications when *x* is changed. However, in case if *x* is not in the system, the change frequency detection process starts which takes up to 1 day. Initially servers poll the web site within 4 hours and depending on the occurrence of number of changes within the time period, server increases the time period to do the next poll by multiplying the time by a factor of 1.5. Hence the times from the initial poll would be 4 hours, 6 hours, 9 hours, 14 hours and 21 hours.

The reason behind using 1.5 as the multiplication factor is the necessity to have changes detected in uneven time intervals. Using change values for uneven time intervals is useful in obtaining accurate results from the machine learning model. Although we have used 4 hours and 48 hours as the minimum and maximum time limits, respectively and 1.5 as the multiplication factor in the experimental setup, these values could be changed according to the necessities when integrating the proposed methodology with a currently available change detection system. These necessities could be factors such as the required minimum change detection time, processing power of the server and the change frequency prediction accuracy of the proposed system.

By increasing the time interval, the server polls the web page several times in the first day to identify whether it is a frequently changing page or not. This classification is performed by a machine learning based model, which assigns a time between 4 hours and 2 days for the new web page based on the changes occurred in each of the considered time-intervals. This would mean that for web pages which change very frequently, we would do server polls every 4 hours and for web pages which do not change very often, servers poll every 2 days [13]. For other web pages in which changes occur somewhere in the middle the poll time could be between 4 hours and 2 days (48 hours). These are the minimum and maximum time limits of the algorithm. Assignment of this time to a new web page would be done within the first day and after that it would be added to the scheduler. Fig. 1 explains how the above change frequency detection process helps in the process of scheduling web sites to detect changes.

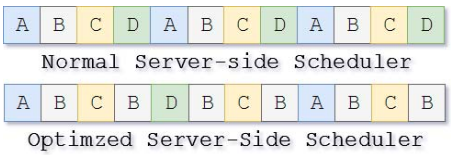

Fig. 1. Optimization of scheduling based on change frequency

Given below is the terminology used for Fig. 1

- 'A' – A web page in which significant changes occur only in long time intervals.
- 'B' – A web page which changes frequently.
- 'C'– A web page which is changing in a moderate manner. The frequency of changes occurring is between web page 'A' and web page 'B'.
- 'D' – A web page in which changes occur very slowly. Changes occur in a manner slower than 'A'.
- Normal Server-side scheduler – Case 1
- Optimized server-side scheduler – Case 2

In the first case of the above scenario, the four web pages would run cyclically, where each web page is polled once in every cycle. However, the four web pages are not of the same type and the amount of changes occurring in each of them is different. Hence it does not make sense to crawl those pages like in case 1. With prior change frequency detection as in the proposed algorithm, the schedule can be optimized as shown in case 2. For example, the web page D, which does not change often, crawled just once during the considered entire time period. The web page B crawled often, thus frequent changes can be identified in a timely manner. Hence, the proposed system facilitates the change frequency detection process in a resource constrained environment.

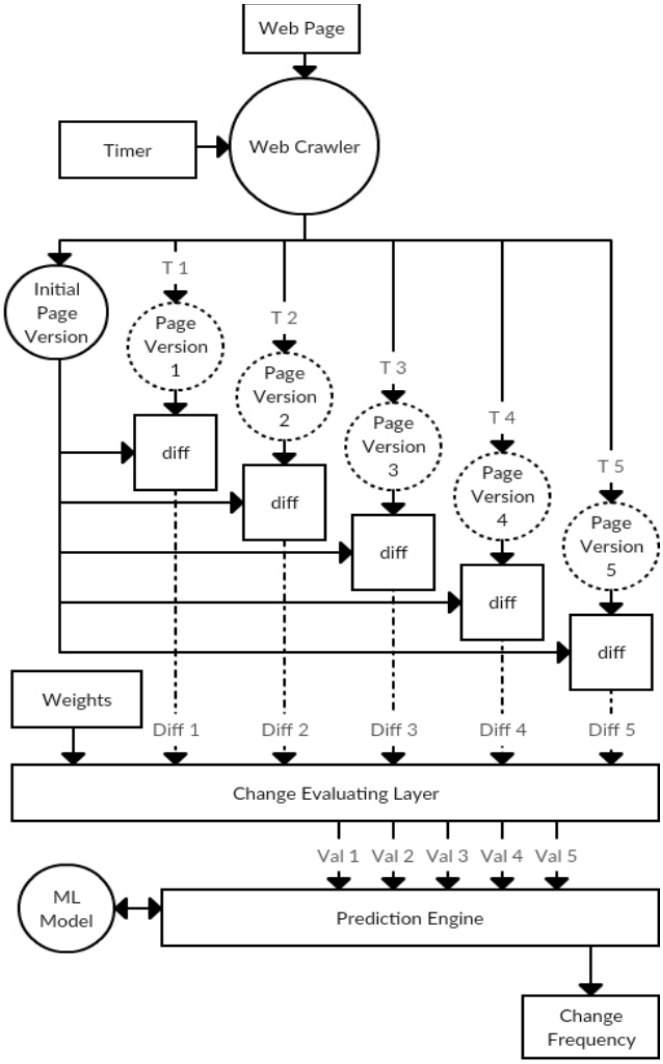

Fig. 2. Flow diagram for change frequency detection

Further, by applying the proposed algorithm to the schedule, the time taken to query and the total time taken to poll web pages do not change; Only the order in which the web pages are polled changes. This ensures that the proposed algorithm does not affect the efficiency of the scheduler.

Fig. 2, shows the simplified scenario of the change frequency detection. As shown here, when the input to the algorithm is a new web page, the output would be the change frequency (the loop-time basket to which the particular web page would belong).

C. Pseudocode for frequency detection

Algorithm 1 describes the pseudocode of the frequency detection algorithm. When user input a new web page to track changes, this algorithm will determine the initial crawl interval, which could be used later by a change detection system.

Algorithm 1 Frequency Detection Algorithm

Require: URL to find the frequency
Ensure: Find a new perfect frequency to poll the web page

```
1.  input: url
2.  w3>w2>w1
3.  threshold
4.  changeList = {}
5.  max_cycle_length = 24hr
6.  current_cycle_length, min_cycle_length = 4hr
7.  initial_version = crawl(url)
8.  while (current_cycle_length < max_cycle_length)
9.      delay(current_cycle_length)
10.     current_version = crawl(url)
11.     new_elements, changed_elements, missing_elements
    = diff (initial_version, current_version)
12.     change_value = w1*new_elements + w2*
    changed_elements + w3 * missing_elements
13.     changeList.add(change_value)
14.     current_cycle_length = round(current_cycle_length *
    1.5)
15. mapped_cycle_length = map_time(changeList)
16. output: mapped_cycle_length
```

Given below is the terminology of Algorithm 1.

- Weights (w1, w2, w3) = prioritize the changes by assigning weight to each type of change.
- change_value = numeric representation of the change
- threshold = accepted change
- max_cycle_length = Maximum cycle length (gap) between two consecutive crawls
- min_cycle_length = Minimum cycle length (gap) between two consecutive crawls
- current_cycle_length = Time gap between the two crawls for which the change is determined
- new_elements = Number of new tags appeared in the current version.
- changed_elements = Number of changed tags appeared in the current version.
- missing_elements = Number of absent tags in the current version
- initial_version = Initial version of the web content
- current_version = Most recent version of the content
- crawl(url) = Grab the web content through the internet
- diff (version_1, version_2) = This function will find the difference between two versions of the same web content over time. This will return the number if newly added element count (new_elements), changed element count (changed_elements) and missing element count (missing_elements)
- map_time(changeList) = calls the machine learning model which assigns an inter-poll time for a given web page.

The algorithm will execute until the *current_cycle_length* exceeds its upper limit. In each iteration, the crawler will be delayed by *current_cycle_length* in order to get a newer version of the web content with respect to the initial version. Although the algorithm has given higher priority to the missing content and lesser priority to the newly added content, these priorities can be manipulated by changing the weights (w1, w2 and w3) that were defined initially in the algorithm; hence, the algorithm is flexible to change.

After exiting the while loop, the loop-time of the web page would be mapped to the schedulers time scale using the machine learning based mapping function called *map_time* and the answer would be returned as the output from the algorithm. This *map_time* function puts the web site into a basket out of 5, where each basket has a loop-time assigned to it. The loop-time values for each basket in hours are 4, 12, 24, 36 and 48 hours. Hence after the change values are sent to the machine learning based *map_time* function, frequently changing web pages would get a small loop-time compared to non-frequently changing web pages which would get a higher loop-time.

The *mapped_cycle_length* could be used as the input to a change detection system in scheduling the list of web pages to crawl to detect changes. At the runtime of the scheduler, if change detection system detects a mismatch between the input given from Frequency Detection Algorithm and the actual behaviour of the web page, it will notify the frequency detection system to recalculate the *mapped_cycle_length.* This kind of situations could arise when abnormal behaviours are shown by web pages during the time when frequency detection system crawls the web page.

IV. Methodology

We implemented two java programs to simulate the *optimized server-side scheduler* (implementation of the scheduler for the change detection systems using the proposed methodology) and the *normal server-side scheduler* (an implementation similar to existing change detection methodologies).

The *normal server-side scheduler* is simulated by implementing a scheduler, which has a collection of websites to detect changes. The scheduler would take one by one of its web pages and crawl for changes. Each website in the collection would get exactly one chance in every cycle to detect any changes. The *optimized server-side scheduler* is implemented with the use of *mapped_cycle_length* values for each of the web pages in the list of web pages to crawl in the change detection system. Within a lower bound (*min_cycle_length*) and an upper bound (*max_cycle_length*) of interval, each web page is given a value for its change detection frequency. For each cycle of the server, it would detect each website according to the frequency value given to that particular web page.

For simulation purposes of the optimized server-side scheduler, we set *min_cycle_length* and *max_cycle_length* values in Algorithm 1, to 4 hours and 24 hours, respectively. Then we set the threshold for change detection to a low value, so that web pages we selected would have more probability to

change within that time interval. Note that when the simulation is running, we ensure that there is at least one website from each, which would take less than *min_cycle_length* to change, more than *max_cycle_length* to change and which would change within those two values.

The experimental study of this research has used 240 web pages to simulate the normal scheduler and the optimized scheduler with frequency related data. The data set consists of all types of web pages from frequently changing to static websites that change rarely. The simulation environment was a virtual server hosted in Microsoft Azure private cloud. The virtual server was running 64-bit IntelTM Intel Xeon E7-4809 v4 (Sandy Bridge) which operates at 2.70GHz. It had two CPU sockets, with 8 cores. The sizes of L1(d/i) and L2 caches were 32KB and 4096KB respectively. It had 16GB RAM with one 64GB hard disk. Virtual server was installed with Linux Ubuntu (kernel 4.4.0-66-generic). The two java programs were run with Oracle JDK (jdk1.8.0_121). At the initiation of each simulation, we ran Java Flight Recorder [14] for 7 days to measure performance of the JVM which simulates running on virtual server.

For the training and testing of the machine learning based classifier we used H2O.ai machine learning API [15] with Random Forest Classifier [16] and used 1179 web pages [17]. When training the machine learning model, 943 (i.e. 1179 x 80%) different web sites were manually collected over a period of 2 months. These websites were carefully chosen to represent web sites ranging from frequently changing to ones which do not change frequently. These 943 websites were categorised into 5 groups according to the frequency in which they change. In order to determine the frequency group, we collected data regarding changes happening in each of those websites for 24 hours (in intervals of 4, 6, 9, 14, 21 hours from the initial crawl) and categorized them into those 5 groups giving them loop-times of 4, 12, 24, 36 and 48 hours. For this categorization, we used human perception regarding websites in addition to the change values.

Then the accuracy of this machine learning classifier is measured using another 236 (i.e. 1179 x 20%) new websites, by putting them in one of the 5 baskets based on the change occurrences in each of the time intervals. Hence, each of the 236 web pages were analysed using the proposed algorithm and sent to the machine learning based classifier. All these 236 web pages were manually categorized into the 5 categories considering the change values for 5 time intervals and the type of web page, without looking at the categorization.The obtained results from the classifier are described in Section V.

V. Results Analysis

This this study, the network usage of each simulation is measured for more than 7 days and the results are shown in Fig. 3 – Fig. 6. Here, a spike in the graph indicates the server polling a web page to get content and detect any changes.

Fig. 3, shows the network usage for detecting changes of a particular web page in *normal server-side scheduler*, similar to existing approaches. Here, the scheduler cycles through its web page collection of 240 web pages every 24 hours. Hence, each web page would be crawled exactly once in every 24 hours in the normal scheduler.

In Fig. 4, it clearly shows how a frequently changing web page is crawled in *optimized server-side scheduling* with the loop-time of 12 hours for the particular web page. After the first crawl, the server crawls again that site in 4 hours (*min_cycle_length*) and checks whether the threshold is exceeded. Then it multiplies that value (4 hours) by the factor of 1.5 and crawls again in 6 hours from the beginning of time. Similarly, it keeps multiplying the time by the factor 1.5 and waits until the threshold gets exceeded. It sends all the change values of the particular web page for each of the time intervals to the machine learning based mapping function and get a suitable time basket for the particular web page. If any site changes more frequently than 4 hours (*min_cycle_length*), then it will crawl once in every 4 hours.

Fig. 5, shows how *optimized server-side scheduler* crawls a web page which does not change often. Here the website takes more than 24 hours of time (*max_cycle_length*) to make any changes, which exceed the threshold. Beginning from 4 hours' interval, server multiplies the interval of crawling for the web page in the factor of 1.5 and detects changes. When the interval exceeds 24 hours, server stops multiplying the interval further and goes through the process of getting a suitable time using machine learning model. Since the web site is not changing frequently, its loop-time is 48 hours according to the model.

Fig. 6, represents the total network usage of *optimized server-side scheduler* when detecting changes of a collection of web pages. The network usage graph for *normal server-side detection,* which is similar to existing methodologies, would almost be the same. Each scheduler continues to crawl for the next website in line as soon as the current website is finished checking for changes. Hence, both schedulers have the same graph for total network usage of their own website collection.

Although the total network usage of both systems looks alike, frequently changing web pages are crawled in a higher frequency in the optimized server-side scheduler than the existing normal server-side architecture. Rarely changing web pages are not checked for changes often, in the optimized server-side scheduler compared to the normal server-side schedulers. Hence, change detection is optimized. It is clear that with the proposed new frequency detection mechanism for changes in web pages, with the same resource utilization and network usage, servers perform the change detection process in a more meaningful way by scheduling in an optimum manner. Further, this mechanism has been experimented and prepared in a manner such that the entire process is flexible and scalable.

The crucial process of this methodology is the machine learning based classifier, which is used to classify web pages into one of the 5 baskets. This classifier is tested to analyse its accuracy in determining a basket and the results are shown in Fig.7.

In this analysis, for each web page in the test set, we consider the term b1 as the loop-time basket determined by the machine learning algorithm and the term b2 as the loop-time basket determined by the manual process. For each of the web pages the difference between b1 and b2, which is the *error* (E) for each web page, was calculated. As shown in Fig. 7, the x-axis gives the error E and the y-axis gives the frequency (number) of web pages with that E.

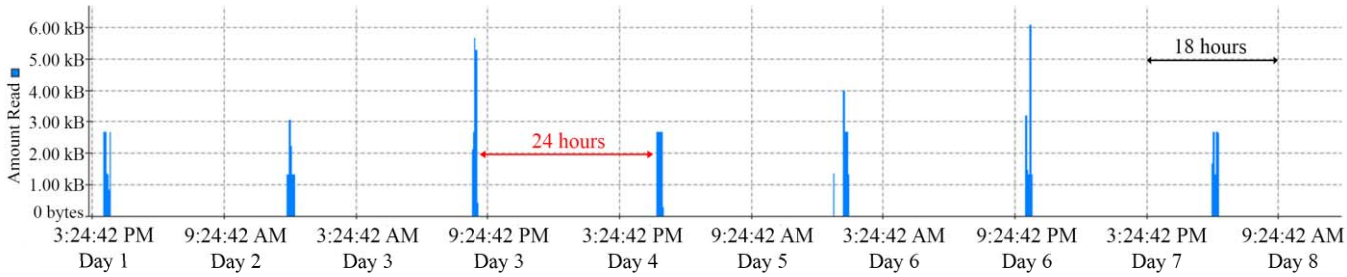

Fig. 3. Change detection of any web page in *normal server-side scheduler* (Period for every web page would be the same)

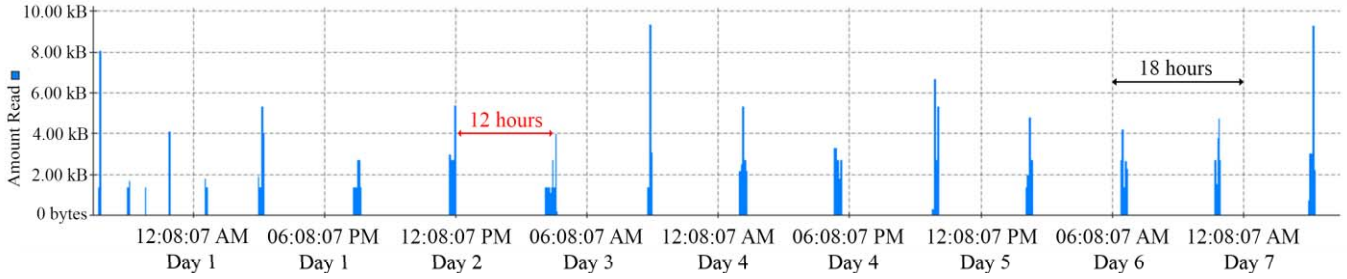

Fig. 4. Change detection of a frequently changing web page in *optimized server-side scheduler*

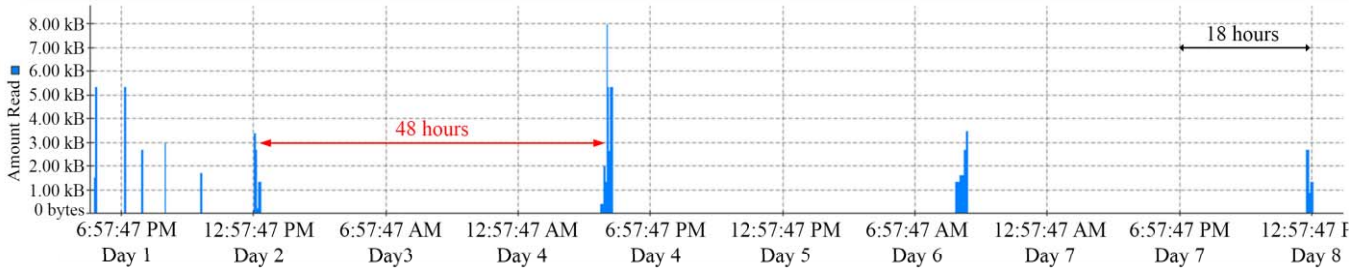

Fig. 5. Change detection of a non-frequently changing web page in *optimized server-side scheduler*

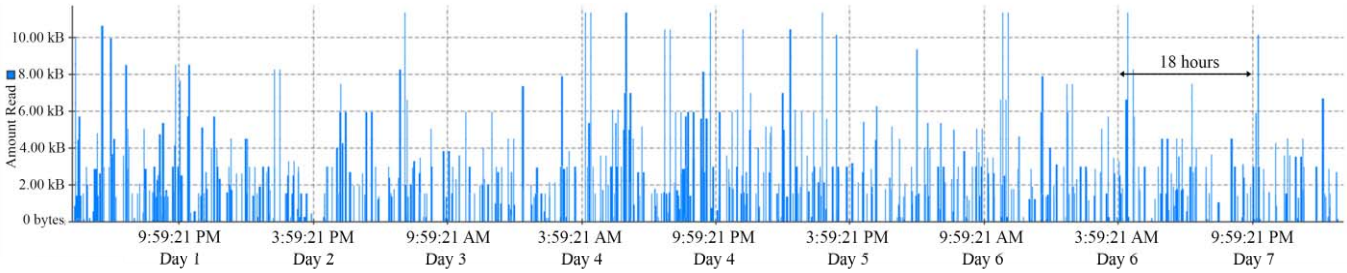

Fig. 6. Change detection of a web page collection in *normal / optimized server-side scheduler*

With this error values, we have defined two terms, *Information gain* (IG) and *Processing gain* (PG). Information Gain is a scenario where, for a web page which needs to be checked for changes in a period of T1 time, the machine learning algorithm would suggest a time basket T2 for that web page such that T2<T1. Hence in this scenario the web page would be polled more frequently than it should normally be polled; hence a positive IG. However, it would require more processing. In this scenario, PG (which would be explained in the next paragraph) would be negative as it requires more processing in the server to have the information gain.

Processing gain in a scenario where for a web page, which needs to be checked for changes in a period of time T1, the machine learning algorithm would suggest a time basket of time T2 for that web page such that T2>T1. In this scenario, the web page would not be polled sufficiently to detect all the changes. Here, the processing power of the server would be saved; hence can be considered a positive PG. On the other hand, this is a negative IG because it will lose the crucial information about frequent changes.

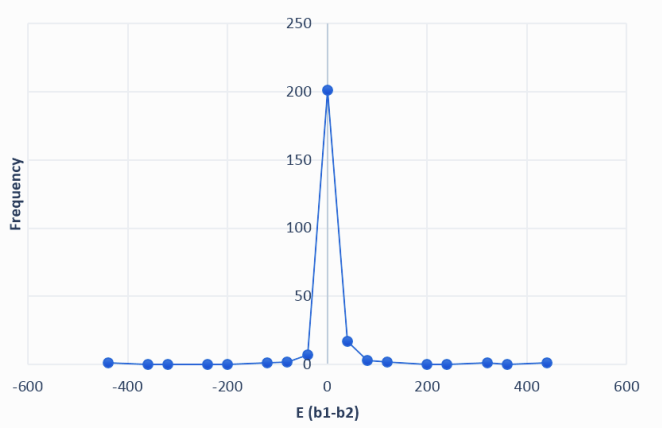

Fig. 7. Frequency against E for the test data set of machine learning algorithm.

According to the results shown in Fig. 7, more than 85.1% of the web pages (201 out of 236) have reported an E value of 0, which means that there is no deviation from the machine learning based algorithm. In these scenarios, no IGs or PGs can be observed as both the values are 0. In 17 cases, we observed an E value of 40, which means b1, is greater than b2. All these cases have positive IGs because more information is collected than normally needed.

However, negative PGs are visible, because they consume more processing power from the server. 7 cases showed an E of 40, which means b1, is less than b2. These cases have positive PGs because they are not checked as frequently as they should be, but they might lead to information losses or negative IGs. Apart from 3 cases, which have 1 case each with E values of -440, 320 and +440, rest of the cases yield E values close to 0.

According to the above results, it can be seen that the proposed algorithm works accurately when IG and PG both are zero; i.e. the point in which error E becomes 0. Hence, the machine learning model is 85.1% accurate and could be improved further with the increase of the dataset used in the model.

VI. Conclusion

Change Detection and Notification (CDN) systems play a significant role on the modern area of information retrieval by automating the process of change detection in web pages. These systems have made the process of keeping track of web pages much easier and less tedious to users. However, still there are opportunities for improvement on the current implementations of CDN systems, when considering performance and speed of detection. This paper is proposed a mechanism to detect the change frequency of web pages,which as a result will optimize the server-side scheduling of change detection and notification services.

An experimental study that has been carried out to compare the performance of the proposed optimized server-side scheduler and the existing normal server-side scheduler is presented in this paper. The experimental results indicate that there is a significant improvement in the performance of change detection using the proposed server-side scheduler with the optimization for the change detection frequency. The change detection process can be performed expressively by identifying the approximate frequency at which changes occur in web pages. Here, the crawlers execute efficiently with optimised resources; thus save computational resources, cost and time. Further, this study has depicted the importance of efficient server-side scheduling in the process of change detection in order to detect changes in a timelier manner, with minimum wastage of computational resources.

This research can be extended for distributed servers as well, where frequency detection is done in multiple servers. It would be beneficial if there is a flexible methodology that can be applied to various scenarios considering the number of servers used in the distributed network. Further research work can be done on the machine learning model, by increasing the dataset and using more parameters for the classification.

References

[1] Google Alerts. (2017) Monitor the Web for interesting new content. [Online]. Available: http://www.google.com/alerts

[2] ChangeDetection. (2017) Know when any web page changes. [Online]. Available: https://www.changedetection.com/

[3] Follow That Page. (2017) Web monitor: we send you an email when your favourite page has changed. [Online]. Available: https://www.followthatpage.com.

[4] S. Jayarathna and F. Poursardar, "Change Detection and Classification of Digital Collections," in *Proceedings of 2016 IEEE International Conference on Big Data*, Washington D.C., USA, 2016, pp. 1750-1759.

[5] D. Yadav and A. K. Sharma, "Change Detection in Web Pages," in *Proceedings of 10th International Conference on Information Technology*, India, 2007, pp. 265-270.

[6] C. Castillo, M. Marin, A. Rodriguez and R. Baeza-Yates, "Scheduling Algorithms for Web Crawling," in *Proceedings of Latin American Web Conference (WebMedia/LA-WEB),* 2004, Brazil, IEEE CS Press, pp. 10-17.

[7] L. Page, S. Brin, R. Motwani and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford University, Stanford, California, USA, Technical Report, 1998.

[8] K. Filipowski, "Comparison of Scheduling Algorithms for Domain Specific Web Crawler," in *Proceedings of European Network Intelligence Conference*, Poland, 2014, IEEE Explorer, pp. 69-74.

[9] S. Nadaraj, "Distributed Content Aggregation & Content Change Detection using Bloom Filters," *International Journal of Computer Science and Information Technologies*, vol. 7, no. 2, pp. 745-748, 2016.

[10] D. Fetterly, M. Manasse, M. Najork and J. Wiener, "A large-scale study of the evolution of web pages*," Software: Practice and Experience*, vol. 34, no. 2, p. 213–237, 2004,

[11] C. Grimes and S. O'Brien, "Microscale Evolution of Web Pages," in *Proceedings of 17th International World Wide Web Conference*, China, 2008, pp. 1149-1150.

[12] ChangeDetect. (2017) Web page monitoring. [Online]. Available: https://www.changedetect.com/

[13] V. M. Prieto, M. A. lvarez, V. Carneiro and F. Cacheda, "Distributed and Collaborative Web Change Detection System," *Computer Science and Information Systems*, vol. 12, no. 1, pp. 91-114, 2015.

[14] ORACLE. (2017) Java Flight Recorder Runtime Guide. [Online]. Available: https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/run.htm#JFRUH164

[15] D. Cook, *Practical Machine Learning with H2O*, 1st ed., O'Reilly Media Incorporated., 2016.

[16] H2O.ai. (2017) Distributed Random Forest (DRF) - H2O 3.12.0.1 documentation. [Online]. Available: http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/drf.html

[17] Dropbox. (2017) List of websites. [Online]. Available: https://www.dropbox.com/s/1izg9qlsjk90pxl/List%20of%20websites.xlsx?dl=0