ENHANCING THE QUEUING PROCESS FOR YIOOP'S SCHEDULER

Project Report

Presented to

Dr. Chris Pollett

Department of Computer Science

San José State University

In Partial Fulfillment

Of the Requirements for the Class

CS 297

By

Gargi Sheguri

Spring 2023

ABSTRACT

Search engines use software programs called crawlers to collate web page information and create indexes. These indexes are used to generate search results, and thus the quality of the results is dependent on the efficiency of the search engine's crawling process.

Yioop is an open-source, PHP-based search engine. Currently, the crawling process in Yioop is made up of a distributed network of QueueServers and Fetchers (crawlers). These programs collect web page data over the internet and use it to build search indexes, which are then used to both generate and rank search results. This ranking operates under the principle that web pages are fetched in decreasing order of importance.

The project goal for this semester is to incorporate a Selective Repeat (Sliding Window) notion to tackle situations wherein a Fetcher retrieval fails or is delayed. Such schedules can induce time lag, causing downloaded web pages to be indexed out-of-order. Thus, some important pages may be presented to the user lower in the SERP (Search Engine Results Page) ranking or even excluded from it. This Selective Repeat addition to the existing URL scheduling mechanism will ensure that web pages are indexed in the same order that they were scheduled in. More ways to improve Yioop's overall search algorithm will be explored and implemented in the next semester.

Keywords: ***Yioop, Search Engine, Scheduling, Selective Repeat, Sliding Window***

TABLE OF CONTENTS

I. INTRODUCTION

The Internet today is a constantly evolving, constantly expanding network of interconnected web pages. Fresh content is continuously being generated and existing content is continuously being updated. Given the exponential growth of the volumes of data available online, there is a growing demand for faster and more accurate search engines. Crawling is an important component of the quality of search results generated by a web search engine. In the absence of an efficient crawling mechanism, search engines may fail to produce updated and accurate results to a user query.

Crawling has now developed into a sophisticated and complex process. It enables search engines to keep their search indexes both comprehensive as well as up-to-date. This project focuses on the crawling and indexing mechanisms used by the open-source search engine, Yioop. Yioop uses a crawling strategy referred to as Host Budgeting, inspired by IRLBot. The issue with the current crawling and indexing implementation is that there is no technique to keep track of the order in which downloaded web page content is added to the search index. This opens up the application to possibilities of web pages being indexed incorrectly despite being crawled in the right order, thus diminishing the accuracy of the search results.

The goal of this project for this semester is to incorporate a similar tracking process to ascertain that web pages are indexed in the same order that they were crawled in. The project is split into four major deliverables, which is the same organization followed by this report.

Deliverable 1 is to become familiar with Yioop's current crawling and indexing implementation. The aim of Deliverable 2 is to fix two outstanding bugs in Yioop to become better acquainted with the development process to be followed, while that of Deliverable 3 is to actually design and integrate the Selective Repeat process that will be used to maintain the web page indexing

sequence. Lastly, Deliverable 4 is to research a Yandex signal that can be used to better Yioop's ranking mechanism, and how it can be incorporated into the current state of the source code. The conclusion section finally wraps up this report, and summarizes the project work we have done for this semester.

II. DELIVERABLE I: UNDERSTANDING YIOOP

The aim of this deliverable is to better understand the components involved in web search engines, the associated crawling, indexing, and ranking processes used by Yioop, and also study the relevant portion of the code base.

To accomplish these tasks, we first read Yioop's ranking factors documentation [1], which was instrumental in helping me piece together the underlying implementation. Yioop's crawling mechanism is made up of: a Name Server, which serves as the crawl coordinator; QueueServers, which keep track of the URLs to be downloaded next in priority queues; and Fetchers, the actual crawlers. We also studied the page processing techniques employed, schedule creation and download, and indexing. URLs to be crawled are ordered in tiered priority queues, which QueueServers (as Schedulers) pick up in a round-robin fashion and create schedule files out of. In the event of an active crawl, Fetchers select a QueueServer and pick up a schedule to download. The contents from the Fetcher are processed and returned to the QueueServer, where this data is written into index partitions (as an Indexer) and further processed to create a cumulative search index.

In the next week allotted to this deliverable, we also read about the crawling process employed by IRLBot [2] and the building blocks of distributed web search engines [3]. Once the groundwork on understanding Yioop's architecture and working was complete, we studied the working source code and analyzed the classes/functions that are important to this project. The core components to look into are QueueServer.php, Fetcher.php, and FetchController.php.

QueueServer.php	QueueServer is an executable whose primary role is to maintain a priority queue of the URLs to be downloaded next. It is also responsible for indexing the data returned by a Fetcher.

	Important Crawling Functions: • start(): Actually starts the QueueServer as a CrawlDaemon • loop(): Keeps checking for start/stop/resume crawl messages from the NameServer, and also carries out all the QueueServer scheduling and indexing functionalities • processCrawlData(): Checks if a fetch schedule file is still waiting to be scheduled to a Fetcher; if not, calls produceFetchBatch() • produceFetchBatch(): Generates a new fetch schedule file of URLs to be downloaded
FetchController.php	FetchController serves as the communication platform between a QueueServer and a Fetcher. It allocates schedules to the Fetcher and handles the downloaded web page data returned to the web app. Important Crawling Functions: • processRequest(): Can call the *schedule* and *update* activities • schedule(): If there is a currently unscheduled fetch schedule file, it is assigned to the requesting Fetcher and deleted • update(): Processes and acknowledges the web page information posted by the Fetcher • handleUploadedData(): Processes the data uploaded by update() and calls addScheduleToScheduleDirectory() on the appropriate information type, i.e. robot info, schedule info, or index info • addScheduleToScheduleDirectory(): Uploads the web page data to the appropriate data subfolder (RobotData, ScheduleData, IndexData)
Fetcher.php	Fetcher is an executable whose primary role is to crawl the given schedule of URLs and download them. Important Crawling Functions: • start(): Actually starts the Fetcher as a CrawlDaemon • loop(): Keeps checking if the crawl has stopped/changed, and also carries out all the Fetcher crawling and posting

	functionalities • checkScheduler(): Requests a QueueServer for new web page URL information by making a request to the appropriate FetchController's schedule() activity • selectCurrentServerAndUpdateIfNeeded(): Sends downloaded web page data back to the QueueServer by making a request to the appropriate FetchController's update() activity

Table 1: List of important classes and functions for Yioop's crawling mechanism

III. DELIVERABLE II: FIX OUTSTANDING BUGS IN YIOOP

This deliverable consisted of fixing two bugs in Yioop: the first was fixing all the PHP deprecation warnings that had arisen after upgrading to PHP v8.2, and the second was modifying Yioop's wiki editor UI to use icons instead of button elements.

1. PHP Deprecation Warnings Fix

On upgrading to PHP version 8.2, multiple warnings had come up due to the deprecation of dynamic property creation. This phase of the deliverable consisted of locating where these dynamic properties had been created and declaring the needful variables as member variables.

We used Yioop's comprehensive test suite to locate dynamic properties. After downloading PHP v8.2, we turned on 'Test Info' under the 'Debug Display' tab in the 'Configure' page. On clicking the 'Test Info' link and running the test suite, various deprecation notices were displayed.

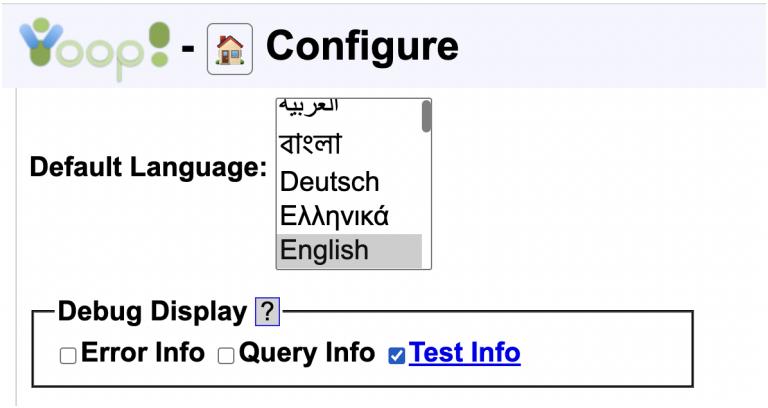

Figure 1: Locating the test suite in Yioop

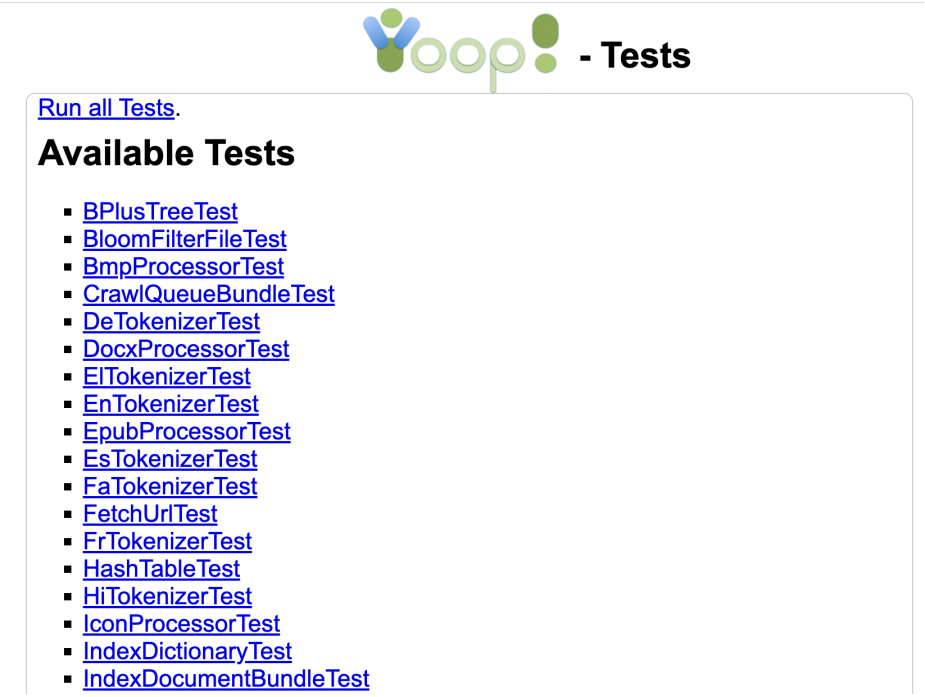

Figure 2: (Partial) list of test cases in Yioop's test suite

To fix the observed warnings, the relevant variables had to be declared in the appropriate classes before being used. Useful PHPDocs were also included in these declarations to explain the function of each of these variables.

2. Yioop's Wiki Editor UI Modification

The latter phase of Deliverable 2 was enhancing Yioop's wiki editor UI to avoid the use of button elements in its HTML. To carry out this enhancement, we proceeded to carry out a series of A/B tests to determine which set of UI changes was the most appealing.

For the A/B testing, the target screen was divided into the Editor UI and the File Upload Area UI. We asked five users to compare the existing wiki editor UI with a few modified versions based on appearance and understandability. This demographic included varying ages and technical fields to collect a well-rounded set of opinions. To do so, we asked them to perform a

set of basic tasks on each and share their feedback (possible improvements, which had better learnability, etc) to finalize the design.

Figure 3: (a) Editor UI (b) File Upload Area UI

The demographic of participants is as follows:

1. UI/UX Engineer (30 yo)
2. Frontend Engineer, current MSCS student (25 yo)
3. Current BSCS sophomore (20 yo)
4. QA Engineer (24 yo)
5. Undergraduate Professor, Web Technology (39 yo)

The list of tasks they were asked to carry out:

1. Display: Chocolate *Chocolate* ***Chocolate***
2. Add hyperlink to the SJSU homepage (https://www.sjsu.edu/)

3. Create a simple grocery list (ordered/unordered/definition)
4. Create a complex list (combination)
5. Change text alignment
6. Add horizontal line between two lines
7. Insert h2 and h4 headings
8. Insert a table with header, two columns, and three rows as shown in Table 3

nation	bender
air	aang
fire	zuko
earth	toph

Table 2: Example table to create for A/B test

9. Add and use search placeholder for "tropical beach"
10. Upload (any local .pdf/.txt file) to main page

The results of the A/B testing led to the following final modifications to the Wiki Editor UI:

1. Editor buttons (rendered with background images) were changed to icons (rendered in tags wrapped in tags).
2. Icons for the non-formatted text, lists, alignments, slide, and table tools were changed to improve learnability.
3. The spacing between the top margin and wiki editor bar was increased.
4. Boxes around individual tools were removed to lessen distraction.

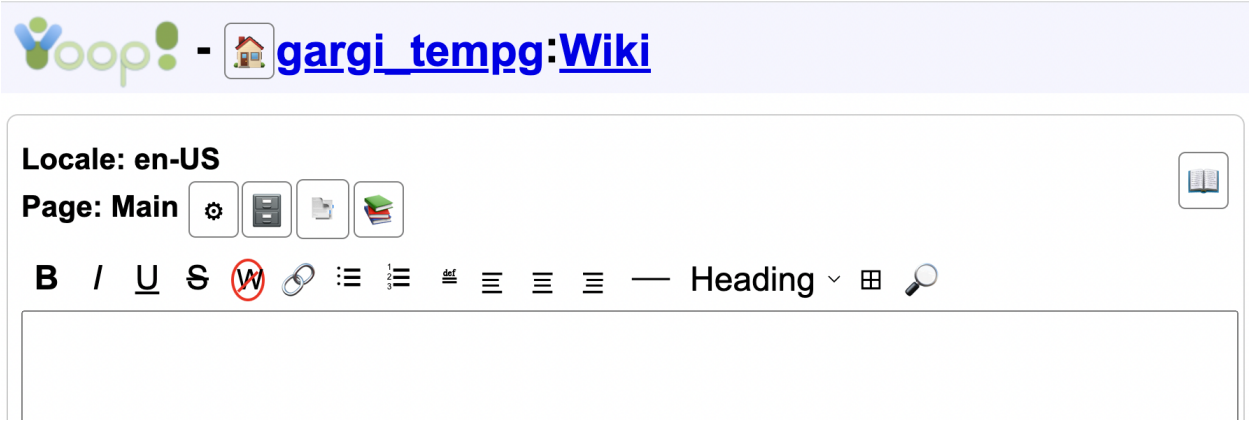

Figure 4: Modified Wiki Editor (for Standard mode)

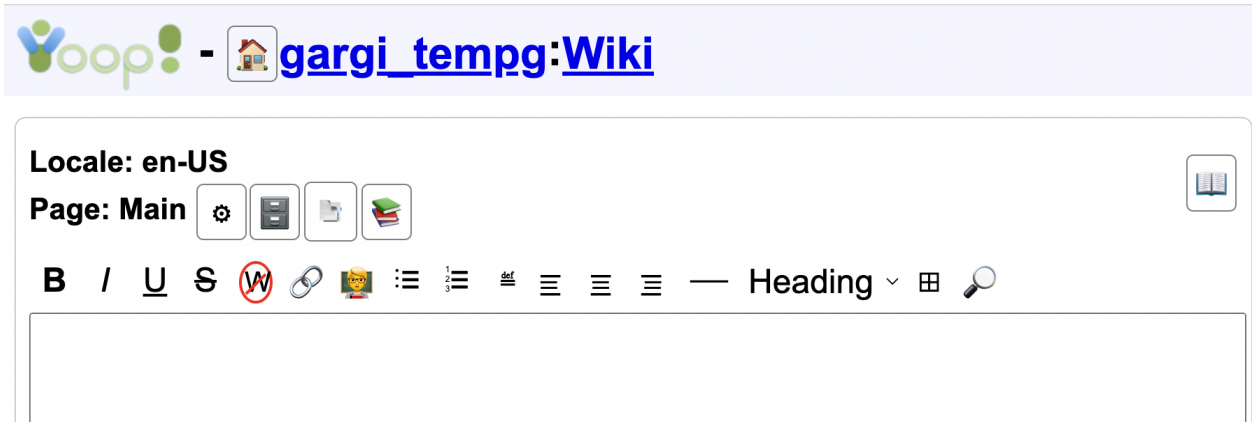

Figure 5: Modified Wiki Editor (for Presentation mode)

As for the File Upload Area, the resultant UI has the following changes:

1. Removed the box and background color from the 'Upload File' area
2. Changed the background color to 'lightgrey'
3. Increased upload-box vertical margin

Figure 6: Modified File Upload Area

IV. DELIVERABLE III: MODIFY YIOOP'S SCHEDULING MECHANISM

The third deliverable encapsulates the main objective of this project, i.e. modifying Yioop's queueing process to retain the crawling order of web pages. Before the inception of this set of modifications, we read about novel approaches towards constructing priority queues in scheduling [4]. This paper outlines two alternative methods of storing the URLs to be parsed next and compares them based on their overall efficiency. Further research into these structures will be done in the next semester.

To retain the crawling order of URLs in Yioop, we inserted a Selective Repeat (Sliding Window) notion between the QueueServer and Fetcher. The sender end of the Selective Repeat protocol creates a window of schedules to be picked up for crawling at a time. Each of these is acknowledged separately by the receiver end. The sent and received schedules are kept track of by means of arrays on either end. In case a schedule is left unacknowledged for a certain period of time (timeout), it is picked up to be rescheduled. Received schedule information is arranged in the receiver array based on its crawling order.

To implement this algorithm, each fetch schedule created by a QueueServer is now assigned a sequence number incrementally, starting with 1. At any given time, QueueServers can only create a window of fetch schedules. FetchController, the coordinating component between a QueueServer and Fetchers, is where the sender and receiver arrays have been set up. These arrays hold objects of type SenderMessage and ReceiverMessage respectively.

SenderMessage	<u>Member variables:</u> • $seq_num: Refers to the sequence number of the schedule

	• $schedule_time: Refers to the scheduling timestamp • $status: Keeps track of the schedule's current status, i.e. 's' for scheduled or 'a' for acknowledged • $reschedule_count: Keeps track of the number of times a schedule has been rescheduled
ReceiverMessage	Member variables: • $seq_num: Refers to the sequence number of the received schedule • $schedule_time: Refers to the scheduling timestamp of the schedule • $diff_origin: Indicates whether the schedule was scheduled by a different QueueServer instance • $filename: Denotes the filename associated with the temporary storage of data from the received schedule

Table 3: Description of member variables in Sender and Receiver Message classes

When a Fetcher posts a 'schedule()' request to a FetchController, the latter first creates a new SenderMessage object with the corresponding sequence number and inserts it into the sender array. The respective fetch schedule file contents are sent back to the Fetcher, but the file itself is kept intact.

Once the Fetcher has finished crawling the requested URLs, it now sends back two additional pieces of information: the sequence number, and a flag indicating whether the schedule's source QueueServer is the same as the current destination for web page information to be posted to. All of this data is sent to the QueueServer collectively through an 'update()' request.

When the FetchController receives the posted web page contents, a new ReceiverMessage object with the received sequence number is created. It uses the aforementioned flag to decide whether the schedule needs to be acknowledged in the sender array. If the flag is unset (indicating the same origin), the corresponding SenderMessage object status is set to 'a', and the generated

ReceiverMessage object is inserted into the receiver array based on its sequence number to arrange it rightly. If the flag is set (indicating a different origin), the acknowledgement notion is skipped and the ReceiverMessage object is inserted into the receiver array based on its time of scheduling. This helps ensure that only those schedules of different origin that fit into the current window are correctly ordered in the receiver (the rest are discarded). Thus, the posted web page content is organized by crawling order as expected.

On a single insertion into the receiver array, the contents of the array are iterated over. If all of the schedules between the currently oldest scheduled sequence number in the sender array and the current sequence number in the receiver array have been received, their web page contents are uploaded to IndexData, their corresponding objects are simultaneously popped from both sender and receiver arrays, and their fetch schedule files are finally deleted to make room for new schedules.

Finally, the last angle added in this implementation is to manage situations wherein the running crawl is paused and needs to be resumed later. This could be in case of an unexpected shutdown of either the running QueueServer or Yioop server, or in case the user gracefully stops the ongoing crawl, or another crawl is resumed/started afresh without shutting down the existing one. To deal with all of these cases collectively, a new function ‘saveWindowData()’ is introduced in the CrawlDaemon class.

void saveWindowData()	<u>Arguments:</u> none <u>Description:</u> This function saves the current ‘window data’ files, i.e. the currently unacknowledged fetch schedules, the next sequence number to be scheduled by the QueueServer (in next_seq_num.txt), and the sliding window sender and receiver information from the FetchController

	(maintained in SlidingWindow.txt) to the cache/WindowData directory. It creates a new folder named WindowData<crawl_timestamp> and shifts the files there, which will be moved back into the schedules directory if the crawl is resumed.

Table 4: Description of saveWindowData() function

To ensure that the Selective Repeat notion works as expected, we validated it for Yioop running as its own web server as well as running under an Apache server on the following setups:

- 1 QueueServer, 1 Fetcher
- 1 QueueServer, 8 Fetchers
- 2 QueueServers, 8 Fetchers

To test the effects of the window size on the overall crawling performance, we also carried out a series of experiment crawls with different window sizes. Each of these crawls had the following common properties:

- Running time: 14 hours
- 5 Seed URLs:
    - https://www.reddit.com/
    - https://www.twitter.com/
    - https://imgur.com/
    - https://www.youtube.com/
    - https://www.ebay.com/
- Crawl setup: 1 QueueServer, 2 Fetchers

The results observed after experimentation were as follows:

Previous implementation (without Selective Repeat)	Result: **Index Description:** test2 **Timestamp:** 1682706591 **Crawl Start Time:** Mon, 28 Apr 2023 11:29:51 -0700 **Downloaded Pages:** 182430 **Seen Urls:** 3080021

Window Size = number of active Fetchers	Result: **Index Description:** test **Timestamp:** 1683213897 **Crawl Start Time:** Thu, 04 May 2023 08:24:57 -0700 **Downloaded Pages:** 137495 **Seen Urls:** 2522217
Window Size = 50	Result: **Index Description:** test2 **Timestamp:** 1683138781 **Crawl Start Time:** Mon, 03 May 2023 11:33:01 -0700 **Downloaded Pages:** 149021 **Seen Urls:** 2877098
Window Size = infinity	Result: **Index Description:** test2 **Timestamp:** 1682976471 **Crawl Start Time:** Mon, 01 May 2023 14:27:51 -0700 **Downloaded Pages:** 149445 **Seen Urls:** 2880009

Table 5: Selective Repeat Window Experimentation Results

Based on the above results, the orderly indexing strategy has definitely slowed crawling down. The explained implementation above (of window size = number of active Fetchers) is approximately 24% slower than the previous implementation.

While an unlimited window size did provide better results than capping it, with a crawl speed of 82% of the original, there were an additional 326 fetch schedule files created ahead of time. This count will only increase for longer crawls, and might lead to memory leaks.

Using a large value for window size, such as 50, showcased similar results. This is because the Fetcher count is a lot smaller than the window size, and the behavior exhibited is thus similar to the infinite size. As the number of Fetchers is much lesser than the window size, there is almost always an unscheduled fetch schedule file to provide a requesting Fetcher. Similar to the previous experiment, however, there were an additional 46 fetch schedule files created by the end of the 14 hours.

V. DELIVERABLE IV: RESEARCH A YANDEX SIGNAL TO INCORPORATE INTO YIOOP

The motive of the last deliverable is to analyze the architecture and ranking factors used by the search engine Yandex, and find a plausible signal that can be incorporated into Yioop's ranking factors. Yandex is the largest search engine in Russia, and portions of its source code were leaked in January 2023. It uses factors called signals to rank search result URLs, which indicate every web page's relevance.

Based on the files leaked, Yandex uses more than 17,000 ranked signals. These signals can be static, dynamic, or specific to user queries. Static signals are those that do not change much for long periods of time, such as the number of links leading to a web page. Dynamic signals tend to vary frequently over time, such as social signals (the number of likes, comments, shares, etc). Some signals can be influenced by the query words used or user preferences, such as geolocation.

At its core, Yandex comprises of an intermediary layer called Metasearch which serves cached results for popular queries. In case results aren't found, the query is forwarded to the underlying Basic Search layer, which is a series of thousands of machines. The machines in the Basic Search layer build posting lists of relevant documents and send them to Yandex's neural network application- MatrixNet- to be re-ranked and built into a SERP.

This posting list computation step of the process is the first place wherein ranking factors come into play. MatrixNet builds complex formulas out of tens of thousands of these signals and increases page relevance significantly. It can also tune signals for query classes and build custom formulas for varying classes.

One particularly interesting signal we studied was FI_NUM_SLASHES. Given a positive weight of +0.05057609417, the possible use for this signal is that the number of slashes is proportional to the depth of the web page (i.e. how far it is from the root/home page). This goes to say that the further a web page is from the home page, the lesser importance it should hold for the query.

To incorporate a NUM_SLASHES ranking factor into Yioop, it can be added as a bonus factor to the Doc Rank score computation. The following changes need to be made to the Yioop code base for this implementation:

1. A new constant NUM_SLASHES will be declared in configs/Config.php.
2. Function findNumSlashes($key) - where $key represents the web page - will be added to library/IndexDocumentBundle.php to calculate the total number of slashes in the page URL.
3. The value of NUM_SLASHES/findNumSlashes($key) will be added to the Doc Rank score in getDocKeyPositionsScoringInfo() in index_bundle_iterators/WordIterator.php.

The implementation of this deliverable, including finding the appropriate value to initialize NUM_SLASHES with through experimentation, will be carried out in the next semester.

VI. CONCLUSION

The purpose of this project is to both understand Yioop's intricacies and improve the queueing mechanism in Yioop's scheduling process. Each of the four aforementioned deliverables contributed to these goals.

Deliverable 1 laid the foundation to comprehend the working and architecture of web search engines, as well as how Yioop has built these components and which parts of the source code make up this implementation. Deliverable 2 improved upon the current version of Yioop. These bug fixes also helped me understand the coding practices and development life cycle used by the application. Deliverable 3, the primary contributor to this project, enhanced the scheduling process by using a sliding window approach to correctly order web page data in the generated search indexes. This set of changes does away with any chances of incorrectly organizing crawled data by attaching sequence numbers to every fetch schedule and using them to order posted data. Deliverable 4 explores an enhancement to Yioop's ranking algorithm by including a Yandex signal (obtained from the Yandex source code leak in January 2023) in DocRank score calculation.

This project and the project next semester are both aimed at improving the crawling performance in Yioop. This preliminary report aims to enhance particularly the URL scheduling mechanism implemented in Yioop. In the next semester, we will explore other ways in which Yioop's overall search algorithm can be improved, by researching new use cases and methods. The researched Yandex signal will also be integrated into Yioop. This addition will further finetune the quality of relevant search results.

REFERENCES

[1] Yioop Search Engine Ranking Mechanisms, https://www.seekquarry.com/p/Ranking.

[2] S. T. Ahmed, C. Sparkman, H. -T. Lee and D. Loguinov, "Around the web in six weeks: Documenting a large-scale crawl," 2015 IEEE Conference on Computer Communications (INFOCOM), Hong Kong, China, 2015, pp. 1598-1606, doi: 10.1109/INFOCOM.2015.7218539.

[3] Distributed web crawler architecture, by S. Severance. (2011, Dec. 15). US20110307467A1 [Online]. Available: https://patents.google.com/patent/US20110307467A1

[4] B. Cambazoglu and R. Baeza-Yates, "Scalability Challenges in Web Search Engines," in Synthesis Lectures on Information Concepts, Retrieval, and Services, vol. 7, 2011, pp. 27-50. doi: 10.1007/978-3-642-20946-8_2.

[5] M. Marin, R. Paredes, and C. Bonacic. "High-performance priority queues for parallel crawlers." In Proceedings of the 10th ACM workshop on Web information and data management, pp. 47-54. 2008.