ENHANCING THE QUEUING PROCESS FOR YIOOP'S SCHEDULER

A Project

Presented to

The Faculty of the Department of Computer Science

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

By

Gargi Sheguri

December 2023

The Designated Project Committee Approves of the Project Titled

Enhancing the Queueing Process for Yioop's Scheduler

by

Gargi Sheguri

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

December 2023

Dr. Chris Pollett	Department of Computer Science
Dr. Robert Chun	Department of Computer Science
Dr. Ben Reed	Department of Computer Engineering

ABSTRACT

Enhancing the Queueing Process for Yioop's Scheduler

by Gargi Sheguri

Indexing in search engines is the process of storing information related to crawled pages to facilitate searches. A crucial determinant of the success of a search engine is the efficiency of the indexing process utilized, which greatly affects both the speed and relevancy of search results. Yioop is an open-source web search engine that employs an inverted index strategy, wherein each term is mapped to a list of the documents it appeared in while crawling.

The primary aim of this project is to better the indexing system used by Yioop, and thus improve the quality of the Search Engine Results Page (SERP) generated for user queries. To achieve this, various methods aimed at bringing down the processing time and boosting Yioop's page ranking mechanism have been employed. These modifications have been implemented in both the indexing process as well as in the lookup process. To bolster more relevant pages in the final results order, bonus factors for scoring certain types of documents higher are incorporated into the indexing process. The lookup system has been revised to fetch the most recently-crawled version of a document in an effort to improve freshness. Furthermore, Yioop now uses disjoint queries to maximize the number of results produced for a search phrase. In order to cut down on the response time, MaxScore calculation has been put into effect, which approximates an upper bound on the contribution a search term can have to the overall output ranking.

These enhancements have each been efficiently designed and evaluated to make sure that they further the quality of Yioop's search functionality. This project report provides a comprehensive outline of the details and impact of these improvements.

Keywords: ***Yioop, Search Engine, Search Engine Results Page (SERP), Indexing***

ACKNOWLEDGMENTS

I would like to express my heartfelt appreciation to my project advisor, Dr. Chris Pollett for being a beacon of inspiration during my journey as a graduate student. I am deeply grateful for his expertise, guidance, and genuine interest in both this Masters project as well as my in personal and academic growth. I would also like to take this opportunity to thank the distinguished faculty at San José State University, particularly my committee members, Dr. Robert Chun and Dr. Ben Reed, for their support and insight. I am also incredibly thankful to all of my cherished friends and family, for consistently fueling me with determination and being untiring pillars of support over the past two years.

TABLE OF CONTENTS

CHAPTER

CHAPTER 1

INTRODUCTION

The Internet today is an unending, dynamic trove of information. It is thus vital to employ effective means to catalog this information for quick and reliable retrieval via search queries. In the context of search engines, indexing refers to the activity of efficiently storing and organizing the contents of web pages obtained from crawling the Internet. The primary motive behind creating an index is ensuring fast and fruitful data fetches as responses to search queries. Indexes pre-process and store extracted information including important keywords, hyperlinks, and descriptive metadata from pages in a structured manner to facilitate easy lookup.

The lookup (or search) functionality in a search engine is the process of fetching web page data most accurately answering a user's input query requirements. On submitting a query successfully, the search engine internally uses complex algorithms to scan its index and retrieve information about the stored web pages that best match the terms in the search query. This collection of results is scored and ranked based on a variety of criteria to find the most relevant pages, which are then supplied to the user in decreasing order of importance.

Yioop is PHP-based, open-source web search engine. It uses an inverted index to efficiently store and arrange web page information. The purpose of this project is to investigate and incorporate methods to improve the quality of search results generated by Yioop for user queries. These developments span across both the indexing as well as the searching processes.

This project is split into three major deliverables, each of which is aimed at ultimately bettering the search feature in Yioop. We first study the different kinds of factors that influence the visibility of documents in the SERPs of modern search engines such as Yandex. These special factors include the number and quality of hyperlinks leading to a page (backlinks), how a page URL is formed, whether a page originated from a secure connection or not, etc [1]. We use this

knowledge to then introduce two new bonus factors into Yioop: WIKI_BONUS, a flag that identifies pages originating from Wikipedia and boosts them in the search results, and NUM_SLASHES_BONUS, which pushes URLs closer to the domain homepage above those nested deeper. These bonus factors are inspired by Yandex's FI_IS_WIKI and FI_NUM_SLASHES respectively [2].

The successive phases of this project focus on making the lookup operation more efficient. To achieve this, we buckled down on two major types of enhancements: uplifting the SERP freshness [3, 4] and increasing the number of results presented to the user.

To carry out the former target, we modified the search flow in Yioop to always include only the most recently-crawled copy of a web page in the final collection of results. When the crawling operation is performed, the primary assumption is that the most important pages are crawled first. This means that relevant documents appearing earlier in the index are deemed as "better" than those appearing later on, and will thus make it to the top few results. However, this can also bring forth a challenge. To keep up with websites whose contents are updated often, Yioop needs to recrawl some pages intermittently. Hence, it is likely that in the event of a long crawl, an indexed page appearing in the top results may be an outdated version of the website, while the newer version is presented as a separate result lower down in the SERP or even skipped altogether. It is also possible that the latest version of the web page doesn't contain the search terms at all. This inclusion of stale results can impact the quality of search negatively. With the modifications from this deliverable, the served results are always recent versions of the URLs while retaining the original rank and avoiding duplications.

The final deliverable of this project is aimed at increasing the total number of responses to a user query. Yioop previously used conjunctive queries to find relevant web pages, which means that only pages containing all of the terms of the search query were included in the final results. We introduced a round of changes that allows Yioop to use disjunctive query structures [5] by

default, which can be changed back to conjunctive logic by toggling a flag in the configuration. The purpose of using disjunctions is to bolster inclusivity and impose lesser restrictions in search, and therefore achieve a larger pool of results. We also incorporated a heap data structure [6] to sort the top ranked documents in intervals rather than ultimately ordering the entire set of results to cut down on the sorting time complexity. To further ameliorate the complexity of lookup, this deliverable includes the implementation of the MaxScore algorithm [7] to weigh the maximum contribution of each search term to the overall SERP and avoid searching the index for terms that cannot independently assist in boosting a document to the top results.

The organization of this report follows this pattern. The upcoming chapter provides an in-depth coverage of the project background, comprising of the search engine Yioop, and how the indexing and search functionalities work in Yioop. It also covers additional background about the algorithms and concepts related to this project. Next, the Preliminary Work chapter contains prerequisite information about the results achieved in the initial half of the project, and we go summarize the initial study of Yandex and other search engines performed. The following chapter on Implementation discusses the aforementioned deliverables in detail, including their designs, specifics about their implementation, and observations. Finally, the report culminates with the Conclusion section.

CHAPTER 2

BACKGROUND

This chapter covers the background information required to understand the project. We first go over some of the related work in the sector of information retrieval, and then talk about Yioop as a search engine and its search system. Finally, we will discuss some of the concepts used to make the lookup process more efficient, comprising of disjunctive queries, heap data structures, and the MaxScore algorithm.

2.1 Yioop Web Search Engine

Yioop is an open-source, multilingual web search engine portal and has been published under the GNU General Public License. Yioop is developed in PHP and is capable of storing and retrieving documents, websites, and images efficiently. There are two prominent functionalities it supports, which are crawling and indexing. Crawling is the process of downloading and parsing web pages roughly in their decreasing order of importance, whereas indexing is the process of processing the information extracted from these crawled pages and storing them effectively for fast retrieval.

To handle and coordinate between these operations, Yioop's codebase is divided into multiple high-level components [8]. These consist of Fetchers, Schedulers, Indexers, and Query Processors. As their names suggest, Fetchers are responsible for the actual web page crawling in Yioop. They are handed batches of URLs to download and "fetch" information by downloading and parsing these websites. Schedulers maintain priority queues of URLs that are to be crawled next, and create batches for Fetchers to pick up. Once a Fetcher sends the information extracted from crawled web pages back to the server, the Indexer oversees the pre-processing and subsequent storage of this data into the inverted index. The pre-processing step includes detecting the page properties (such as language, page type, domain), extracting links and

important terms from the page contents, and generating a summary for the web page. Each term found is stemmed and converted into a standard format, and finally a mapping between the term and document is inserted into the inverted index. The Query Processor manages the manner in which user queries are processed and lookup is performed for that search phrase.

2.2 Inverted Indexing

A popular strategy followed by most modern search engines, including Yioop, is storing large collections of web page information in structures called inverted indexes [9]. In this mechanism, the search engine supports a mapping between terms and the documents that these terms appeared in. In this context, a term could either be a pre-processed word or phrase. Inverted indexes use terms as keys to search for relevant documents, which is done by storing a list of pointers to the actual locations of the documents' information in the index in the value field. Yioop uses a directory of files to store its inverted index, as it can become too large to fit into main memory. Each of these files is called a partition. Once the in-memory index is large enough, it is written into a new partition and subsequently wiped from memory. Indexing then continues as if from the beginning.

The benefit of using this inverted index format of storage is fast and efficient retrieval of documents, especially when the size of the index is very large. This is particularly better suited for lookup than traditional methods of indexing (such as forward indexing) for large indexes. Once a document has been retrieved, its page rank can be calculated optimally by using additional data like document and term frequencies. The following table captures the portions of Yioop's codebase that are most important to the creation and maintenance of its inverted index:

IndexDocumentBundle.php	The IndexDocumentBundle is responsible for creating an inverted index out of the crawled web pages. Important indexing functions:

	• updateDictionary: Adds posting list information to the dictionary B+ tree for the index in memory • buildInvertedIndexPartition: Builds an inverted index shard for the given partition • invertOneSite: Creates an inverted index for a single document and adds it to the current partition • computeDocId: Generates a unique *doc_id* for the given crawled website information
PartitionDocumentBundle.php	A PartitionDocumentBundle holds the collection of partitions that make up an inverted index. Each partition is assigned a serial integer value to uniquely identify it. <u>Important indexing functions:</u> • put: Adds new entries to the current partition • get: Returns the values for the given fields for the given key value from the given partition
PackedTableTools.php	This class defines methods that describe how records are to be encoded and decoded while creating a partition.

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

Indexes in Yioop are made up of the following components:

- documents:
    - The documents directory is a type of PartitionDocumentBundle and contains independent partitions. Each partition is numbered serially represented by a pair of files: *partition_(some_integer).ix* and *partition_(some_integer).txt.gz*.
    - While the *.txt.gz* file holds sequential entries of gzip-compressed document summaries and objects, the *.ix* file contains records referencing these entries via records of (*doc_id, offset in the .txt.gz file to the document summary, offset in the .txt.gz file to the document object, length of the document object*).
    - documents also keeps information related to the utilized record and compression formats in Yioop, an upper bound on the size of a partition (both in bytes and

count of records), and the number of URLs visited.

- positions_doc_map:
    - This directory is made up of a nested series of directories (marked numerically), wherein each corresponds to a partition in documents. Each of these is made up of three additional files, namely doc_map, positions, and postings. The last folder also has a last_entries file.
    - doc_map:
        - This file contains a list of tuple pairs of *doc_ids* mapped to a list of (*position, score*), signifying an offset into the corresponding *.txt.gz* file for that document, along with its general score (from the crawl).
        - To facilitate deduplication, URLs with the same computed hash values (in the *.ix* file of the current partition) are grouped together and represented by a single *doc_id*. The general score for this set of document entries is a lumpsum of the scores of the constituent documents.
        - Each list of *(position, score)* signifies the score that should be assigned to term locations in the document's text contents sequentially.
        - The final pairs represent scores for the document for each classifier the crawl is currently using.
    - positions:
        - This is a binary-format file and keeps track of the encountered terms in a partition. For each *term_id,* its location in every document it appeared in is stored through a gamma-code for the first occurrence and Rice-code difference format for successive occurrences.
        - *term_ids* are made up of first extracting the initial seven characters of the term (padding with _ for shorter terms) and then appending an 8 byte hash value of the term.

    - postings:
        - The postings file keeps track of the actual inverted index mapping between *term_ids* and their posting list. This is done by maintaining a list of tuple pairs, where each posting list comprises of (*document index in doc_map, number of occurrences of term in document, offset into positions file of term_id's position list, total length of entry for term_id in positions file*) for each document the term shows up in.
        - For older partitions (that are already full), the *term_ids* are stored in the dictionary folder instead.
        - The *document index in doc_map* and *offset into positions file of term_id's position list* values are stored in a delta difference format.
    - last_entries:
        - This file holds triplets of *(term_id, last_doc_index, last_offset, number of occurrences),* wherein the *last_doc_index* and *last_offset* values are used to locate the components based on the last non-delta'd values for the posting to be added next.
- dictionary:
    - This directory is responsible for holding information about the B+ tree used for retrieval. This is done by mapping *term_ids* to their posting lists, i.e., where the posting list for the term can be found in positions_doc_map.
    - Nodes are represented in nested folders with references to the next node in the tree. Each *term_id* points to a sorted list of records representing partition information in the form of (*partition_number, number of documents in that partition that the term_id occurred in, total occurrences of the term_id in that partition, offset into postings file for that term_id, and the total length of the posting information*).

- next_partition.txt:
    - This file holds an integer value denoting the next serial number to be given to a new partition.
- archive_info.txt:
    - The archive_info file has information about the archive's creating time, crawl parameters used, and the archive's version format.

2.3 Query Processing in Yioop

Yioop uses document-at-a-time processing. In this strategy, the search engine calculates scores for whole documents instead of individual search terms. These scores are based on a number of factors, such as the frequency of the search term appearing in the document and its posting in the index. Once a list of the matching documents has been created, they are sorted based on their scores, and the top few pages are returned to the user as the search results. The advantage of using document-at-a-time processing over other strategies (such as term-at-a-time query processing) is that the overall relevance of an entire document to the user's query can be taken into account. This usually leads to a better quality of results, as the context of the search phrase is considered while finding the resultant pages.

Yioop uses the Divergence-from-Randomness (DFR) algorithm [10] to determine document scores. This algorithm operates with the assumption that terms are distributed randomly across a document. Document scores are thus calculated under the consideration that the odds of the term distribution actually found is per chance. The DFR for a document (for a particular term) is given by [11]

$$DFR_t = (1 - P_2) * (-\log P_1)$$

Here, P_1 signifies the probability that the term t occurs exactly $f_{t,d}$ times in any document d (chosen randomly). The number of bits of information associated with this knowledge (called

self information) is approximately $-log\ P_1.\ P_2$ is used to make up for the rapid gap created between P_1 and $-logP_1$, and is referred to as a the *eliteness* representing whether *d* is "about" the topic corresponding to *t*. The *eliteness* of a document tells us about the probability that *t* occurs in it at least once.

2.4 Disjunctive Queries

Disjunctive logic in search phrases is implemented by separating each term with an OR operator. Basically, this means that the search engine will search the index for documents that contain any of the search terms, rather than all of them. This is the alternative to the conjunctive logic previously utilized by Yioop, wherein search phrases where treated to be separated by AND operators, and thus the SERP would be made up only of documents containing all of search terms.

Disjunctions are helpful in broadening the search scope and are hence useful in fetching a more diverse set of resultant web pages to suffice user requirements. This becomes particularly useful when users wish to search on synonyms or alternative words. By allowing greater flexibility in the conditions that will be used to match results, the odds of documents meeting the expectations of users looking for comprehensive results also increase.

2.5 Heaps for Query Processing

The heap data structure is a binary tree that is popularly used for sorting. These heaps can be used to sort elements in either ascending or descending order, which we label as min heaps and max heaps respectively. Min heaps retain the minimum element in the tree at the root level, whereas max heaps keep the maximum element at the root level.

Yioop uses document-at-a-time processing, wherein each document in the index is scored based on the search phrase to find *m* matching documents. These are then sorted and the top *k* documents are returned as the search results. Given that the SERP only displays *k* results,

storing and sorting through *m* documents can become an additional overhead.

A better approach towards generating the results page is using a min heap to store the top *k* documents as the index is processed and disregard documents scoring lower than the current k^{th} best score. The total cost of building a min heap is O(*k*), and the cost of sorting after all *m* documents have been returned is O(*k* * *log k*). Sorting is the result of extracting each document one-by-one from the heap, where the k^{th} best scoring result is removed in O(1) and the tree is then reheaped to maintain the min heap property in O(*log k*) on each deletion.

2.6 MaxScore

The MaxScore of a term refers to approximating an upper bound on the maximum score that a document containing it could achieve. This is a consideration of the value that each individual term in the search phrase could contribute to the general score. When a search query is entered, it is first pre-processed and separated into a list of terms for lookup in the index. The Yioop codebase then creates an iterator for each term to find matching documents.

For every individual search term, an initial theoretical maximum score that a document containing it could achieve is calculated. This is called the term's MaxScore. The MaxScore of a term is directly proportional to its individual score contribution, which is determined by the DFR algorithm described previously in Yioop. Amati and van Rijsbergen [10] have further simplified the DFR formula by estimating values of P_1 and P_2 to

$$\mathrm{DFR}_t = \frac{\log\left(1 + \frac{l_t}{N}\right) + f_{t,d} \log\left(1 + \frac{N}{l_t}\right)}{f_{t,d} + 1}$$

where $f_{t,d}$ represents the frequency of term *t* in document *d,* and l_t represents the occurrences of *t* distributed over *N* documents, such that

$$l_t = f_{t,1} + f_{t,2} + \ldots + f_{t,N}$$

Based on the above formula, an upper bound can be determined on the contribution of a term to the results as

$$\lim_{f_{t,d} \to \infty} DFR_t = \log (1 + N/l_t)$$

The overall score of a document is also dependent on the position of the document in the index, since we know that Yioop's crawler assumes that the most important URLs are crawled first. This position is called the DocRank, and is found as

$$DOC\_RANK_d = \log_{10}(\text{number of documents after } d)$$

The above equations thus lead to the calculation of the MaxScore of a term as

$$MaxScore_t = \log (1 + N/l_t) + DOC\_RANK_{max}$$

where N is the total number of documents in the index, and DOC_RANK_{max} refers to the maximum document rank possible.

As matched documents are returned to be sorted into the top k results during the lookup process, the value of each MaxScore is compared to the current k^{th} best score. If a MaxScore is lesser than the current k^{th} best score, documents containing only the corresponding term cannot make it into the top set of results. In this case, the iterator for the term is thus deleted and the lookup operation continues with the remaining iterators.

CHAPTER 3

PRELIMINARY WORK

The overall goal of this project is to improve the performance of Yioop's scheduling and indexing system. In this chapter, we discuss the target and results that were achieved in the first half of this project. Whereas this phase of the project focuses on the betterment of the indexing process, the initial phase focused on enhancing the crawling mechanism in Yioop.

3.1 Crawling Operation in Yioop

The crawl functionality in Yioop [8] is made up of three major components: the NameServer, QueueServers, and Fetchers. The NameServer acts as the crawl coordinator and is responsible for starting, stopping, and managing crawling activities. QueueServers overlook the scheduling mechanism of a crawl by maintaining priority queues of URLs to download in batches. Fetchers are delegated the responsibility of visiting and downloading these URLs, carrying out an initial processing on the obtained page contents, and posting this information back to the QueueServer.

The first half of this project was targeted at improving the crawling mechanism in Yioop. This was done by initially studying the relevant portions of Yioop's source code to identify the various components involved in crawling and understand how they work together.

QueueServer.php	The executable QueueServer can take up the role of either the Scheduler or the Indexer. Its pivotal role is to maintain priority queues of the URLs that are to be downloaded next. Once Fetchers post web page information back to the server, the QueueServer processes and indexes the data. 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	The FetchController basically bridges the communication gap between the QueueServer and Fetchers. It is responsible for handing a requesting Fetcher the next schedule of URLs to be downloaded, and further handles the posted web page data. 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	The executable Fetcher is Yioop's crawler. Fetchers visit and download URLs in the order provided to them, and carry out an initial pre-processing of page contents before handing the information back to the QueueServer. 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 2: List of important classes and functions for Yioop's crawling mechanism

3.2 Selective Repeat Protocol Integration into Yioop

A major issue found in prior versions of Yioop was that in certain cases, the order in which URLs are to be crawled was not maintained. This could happen in cases where a Fetcher retrieval was delayed or failed altogether. As schedules of URLs are handed to Fetchers to be downloaded in their approximate order of importance, unexpected problems with Fetcher retrievals could cause pages to be returned out-of-order (or cause some scheduled pages to be skipped completely), and thus affect the ranking of results during search.

As a part of this project, a Selective Repeat (Sliding Window) notion was implemented in the Yioop crawl logic to keep an orderly track of URLs that were sent out by the Scheduler. Each schedule picked up by a Fetcher was assigned a serial number. This notion was added between the QueueServer and Fetcher components. A window of schedules is created on the sender end. When the FetchController dolls out a new schedule of URLs to be downloaded, the Selective Repeat protocol adds a log of the schedule number and sent time to the window. On receiving a bundle of web page information from a Fetcher, the FetchController first pushes the data into a

receiver window and checks its schedule number. This window is used to arrange received website information correctly while indexing. In case a schedule fails or times out, it is re-issued.

3.3 Bonus Factors in Yandex

The Yandex search engine and web portal is a widely-utilized search engine, most popularly used in Russia. In January 2023, close to 45 GB worth of Yandex source code was leaked online by a former employee [2]. This code also revealed more than 1,920 of the search factors Yandex used. Although the actual source code isn't legally available on the Internet anymore, several verified resources documenting key takeaways from the leak are still accessible. Most of these sources have compiled a list of search ranking factors based on personal analyses of the code documentation and their observed values.

D. Goodwin [2] put forth a listing of several important ranking factors, which were thoroughly studied to identify any plausible factors that can be incorporated similarly into Yioop. Two significant signals that were found were FI_IS_WIKI and FI_NUM_SLASHES. Based on the documentation accompanying its declaration, FI_IS_WIKI indicates that web pages originating from Wikipedia get preference while computing document scores. There are a number of reasons for this. Wikipedia is a well-known and trustworthy source of information, owing to its collaborative nature. Moreover, it has exhaustive information on a wide range of popular subjects that tend to be updated frequently. Most Wikipedia articles also include detailed lists of references which can easily be cross-verified to ensure accuracy. Yandex's source code has assigned the FI_NUM_SLASHES signal a positive weight of +0.05057609417. The probable use case for this is that the number of forward slashes (/) in a URL corresponds to its depth from the domain home page. Hence, the further a page is nested within a domain, the lesser important it is with respect to the home page. This project has added two new bonus factors WIKI_BONUS and NUM_SLASHES_BONUS (inspired by Yandex's FI_IS_WIKI and FI_NUM_SLASHES respectively) into Yioop's lookup logic to improve search results.

CHAPTER 4

IMPLEMENTATION

This chapter focuses on the actual design and development of the enhancements made to Yioop's search mechanism in this project.

4.1 Incorporation of Yandex-inspired Signals in Yioop

The Yandex source code leak in early 2023 revealed that the search engine uses more than 17,000 ranking signals while finding search results. These signals can be any of static, dynamic, or user query specific types. While static signals do not change drastically for long intervals of time (such as the number of hyperlinks leading to a website), dynamic signals like the number of shares on a post change often. Other signals can be specific to user queries and preferences, such as the user's location. This project added two ranking factors inspired by Yandex to Yioop's bonus scores.

While adding a document to the inverted index, Yioop generates an ID to represent the document URL uniquely in an effort to standardize the index entry. This is referred to as the *doc_id*. A *doc_id* is made up of the following parts (in order):

- 8 bytes representing the URL hash value
- 8 bytes representing the URL hostname hash value
- 1 byte representing a letter code
- 7 bytes representing the text contents of the document

The hash values used for *doc_id* creation are the result of MD5 hash functions that perform XOR operations on the first and last 8 bytes of the required text (7 byte hashes do not use the last byte of this value).

To accommodate the new ranking factors, the 17th byte in the *doc_id* format (the letter code) was modified to include information needed to calculate the bonus values. Whereas the letter code was previously mapped to a single character representing the type of document (image, text, video, etc.), it now uses the following arrangement:

- The 8th bit represents whether the URL is a company level domain or not
- The 4th, 5th, 6th, and 7th bits represent the document type (mapped between 0-8)
- The 3rd bit represents whether the URL points to a Wikipedia page or not
- The 1st and 2nd bits represent the number of forward slashes in the document URL

This change to the *doc_id* has been carried out in IndexDocumentBundle::computeDocId.

4.1.1 Wikipedia Bonus Factor

To use the WIKI_BONUS score, the 3rd bit in the letter code is now a flag that represents whether or not the document originated from Wikipedia. This flag is set based on whether the hostname of the URL contains the substring *wikipedia* in it. When a lookup operation for a search term is being done, the WordIterator corresponding to it will use the this flag value to adjust the relevance score of a found document. This addition to the document score is included in the WordIterator::getDocKeyPositionsScoringInfo, which uses the *doc_id* of the current document to check whether WIKI_BONUS should be included in the relevance calculation. To perform this check, IndexDocumentBundle::isAWikipediaPage is called on the *doc_id,* which returns the value of the target bit in the letter code byte.

Further experimentation with multiple values of WIKI_BONUS was performed to land on the best suited weight for the bonus. The testing setup was as follows:

Tested Weights
Bonus = 5
Bonus = 1
Bonus = 0.75
Bonus = 0.5
Bonus = 0.25

Table 3: Tested Weights for Deliverable#1: WIKI_BONUS experiments

Tested Crawl Sizes
300024
557808
600110
1868398

Table 4: Tested Crawl Sizes for Deliverable#1: WIKI_BONUS experiments

Tested Searches
google
apple
wikipedia
yahoo no:guess
verizon
weather
ebay lang:en
site:google.com

site:apple.com
site:pinterest.com lang:en

Table 5: Tested Search Phrases for Deliverable#1: WIKI_BONUS experiments

The following results were observed for the conducted experiments:

Weights Tested	Observations
Bonus = {5, 1, 0.75}	• All values above 0.5 boosted Wikipedia too far up. • Eg. On testing with scores 1 and 5, *www.wikipedia.org* appeared as the third and first result respectively on searching for *google*, *verizon*, and *weather*. On testing with 0.75, *www.wikipedia.org* appeared as the second result for keyword *apple*, above all *www.apple.com* retail URLs.
Bonus = 0.5	• WIKI_BONUS = 0.5 gave the most appropriate search results for the tested keywords. • All of the host URLs appeared as the first search result in the SERP, followed by seemingly more important URLs (such as *us.yahoo.com, www.yahoo.com/plus/..., developers.apple.com, cloud.google.com*, etc.), and *www.wikipedia.org* ranked higher than other (further nested) URLs (such as the websites *www.apple.com/am/privacy/control, www.barnesandnoble.com* for a search on *apple*).
Bonus = 0.25	• Although 0.25 did give good results as well, the score did not boost Wikipedia pages higher than other lesser-relevant pages for all the search words used. • For some, the SERP results were the same/very similar to the previous implementation (no WIKI_BONUS). • Eg. For search word *wikimedia*, wikipedia.org appeared lower than quickbooks.intuit.com.

Table 6: Observations from Deliverable#1: WIKI_BONUS experiments

Based on the above results, the value of 0.5 was chosed as the default weight of the bonus factor WIKI_BONUS.

4.1.2 Number of Slashes Bonus Factor

The implementation of the number of slashes bonus factor is similar to the Wikipedia bonus factor implementation explained above. While generating the *doc_id* for the document to be indexed in IndexDocumentBundle::computeDocId, the number of forward slashes appearing in the URL after the company-level domain name (i.e., appearing after the top-level domain such as .com or .org) is counted in IndexDocumentBundle::findNumSlashes. Based on this count, the first two bits of the letter code are given a specific number between 0 and 3. This number is assigned based on which bucket the count of forward slashes falls into.

To find the most suitable weight and range of buckets that should be used for the NUM_SLASHES_BONUS factor, a set of experiments were carried out. The setup for these experiments was as follows:

Tested Weights
Bonus = 2 Buckets = {0-2, 3-4, 5-7, 8+}
Bonus = 1 Buckets = {0-1, 2-4, 3-6, 7+}
Bonus = 1 Buckets = {0, 1-2, 3-4, 5+}
Bonus = 0.5 Buckets = {0, 1, 2, 3+}
Bonus = 0.5 Buckets = {0-1, 2-4, 5-6, 7+}
Bonus = 0.5

Buckets = {0-2, 3-4, 5+}

Table 7: Tested Weights for Deliverable#1: NUM_SLASHES_BONUS experiments

Tested Crawl Sizes
300024
557808
600110
1868398

Table 8: Tested Crawl Sizes for Deliverable#1: NUM_SLASHES_BONUS experiments

Tested Searches
google
apple
wikipedia
yahoo no:guess
verizon
weather
ebay lang:en
site:google.com
site:apple.com
site:pinterest.com lang:en

Table 9: Tested Search Phrases for Deliverable#1: NUM_SLASHES_BONUS experiments

The following results were observed for the conducted experiments:

Weights Tested	Observations
Bonus = {2, 1}	• Any value greater than 0.5 was significantly scoring CLD/root page URLs higher than any nested pages, sometimes overshooting even more relevant nested pages. • Eg. Ideally, on searching for *iphone*, all *www.apple.com/products/...*, *www.apple.com/support/...*, *www.apple.com//iphone-/specs*, etc. URLs should be high on the list. With the NUM_SLASHES implementation, root URLs such as *www.verizon.com, www.ebay.com,* and *www.grammarly.com* were ranking significantly higher than most *www.apple.com/...* URLs.
Bonus = 0.5 Buckets = {0, 1, 2, 3+}	• This first segregation of '/' count was (again) pushing CLD URLs higher than all nested paths, even if the latter were more relevant to the keyword.
Bonus = 0.5 Buckets = {0-1, 2-4, 5-6, 7+}	• The second segregation of '/' count gave better results than any of the other tested divisions. URLs closer to the root page (such as *www.verizon.com/deals, www.verizon.com/home/internet, www.verizon.com/solutions-and-services* for keyword *verizon*) appeared before deeper nested URLs (such as *www.verizon.com/home/accessories/cables-connectors*).
Bonus = 0.5 Buckets = {0-2, 3-4, 5+}	• The final segregation of '/' count also gave close results. However, there were a few times when a deeper-nested URL appeared above its parent URL (lower down in the SERP). Eg. On searching for *tokyo*, the second set of results pushed *www.apple.com/am/business/enterprise/success-stories/transportation* below *www.apple.com/am/business/enterprise/success-stories/transportation/tokyo-metro*.

Table 10: Observations from Deliverable#1: NUM_SLASHES_BONUS experiments

Based on the above results, the value of 0.5 was chosed as the default weight of the bonus factor NUM_SLASHES_BONUS, and the bucket range was picked as {0-1, 2-4, 5-6, 7+} for the number of observed trailing forward slashes in a URL.

4.2 Getting Latest-Crawled Pages in Yioop Results

To ensure that the same URLs are not crawled over and over again, Yioop makes use of a Bloom filter. This is set up in library/BloomFilterBundle.php. The concept of a Bloom filter was introduced by B. H. Bloom [12], and is aimed at creating a memory-efficient data structure to check element membership in sets. Multiple hash functions are used to map member elements to positions in a bit array. To check if an element exists in the set or not, the same functions are carried out on the test element. If the positions corresponding to the hash output are set, the element is a member. The advantage of using Bloom filters is that it does not allow for false negatives. Thus, if a position is not set in the bit array, the element most definitely is not a member.

During scheduling, the Bloom filter bundle in Yioop keeps track of the URLs that have been crawled already by maintaining their hash values in a bit array. When a Fetcher posts a crawled web page's information back to the QueueServer, the top links occurring in its page contents are extracted to be inserted into the URL scheduling queues. Before adding a URL to the list of web pages to be crawled, the Scheduler checks if it has been crawled already by the membership test delineated above. If positive, the URL has been crawled before and will be discarded.

For long crawls spanning over several days or weeks however, retaining the same Bloom filter can lead to inefficiency. This is both in terms of the space needed to store the Bloom filter files as well as the notion that some websites tend to change frequently. Thus, Yioop clears logs of previously-crawled URLs periodically and starts afresh to allow popular web pages to be recrawled.

This led to the issue that the second deliverable of the project aims to tackle: in the event of a website being crawled multiple times, the original version obtained was treated to be more important than the successive versions. This was because Yioop operates under the assumption that the most important URLs are crawled first. Therefore, top search results could comprise of outdated versions of a website while their most recent versions were found lower down in the ranking or even excluded completely. It was even possible that the latest version of the page didn't contain the search term at all, leading to "stale" results in the SERP and overall deteriorating the quality of search results.

To overcome this problem, this project carried out a set of modifications to both the indexing and lookup processes to ensure that result freshness is retained.

4.2.1 Tracking Constituent Terms in Indexed Documents

The indexing logic was updated to include information about the most important words appearing in the contents of the document being indexed. The purpose of implementing this change is to be able to verify that document contains the current search term in its contents. Thus, when the most recent version of a matched crawled web page is looked up in the index, Yioop first checks if the query term being searched on is present in it. In case the check fails, the matched URL is discarded from the results altogether.

This check for inclusion is done by means of a Bloom filter representing the terms appearing in the document while adding it to the inverted index. A *doc_map* entry has been modified to capture both a document's position score list as before as well as this Bloom filter. Before adding an entry to *doc_map*, IndexDocumentBundle::invertOneSite first extracts the top 300 unique terms occurring in the document. This list of terms is passed into a new function, IndexDocumentBundle::storeWords. storeWords() is responsible for generating a Bloom filter made up of the word list passed in.

The Bloom filter uses 3 hash functions to calculate positions for each term, and the filter is made up of a string representing a bit array of size 125 bytes (or 1000 bits). The value of 300 (for the top terms) was chosen as a cap value through experimentation, where approximately 50,000 URLs were crawled and the maximum number of unique terms found in a single document was 282. The Bloom filter string is also prepended with a character *t* for backward compatibility purposes, such that the logic to find the latest version of a page is only carried out for crawls that took place after this change in the doc_map entry format. For each doc_map entry created and zipped (through Utility::pack()), this Bloom filter is prepended to the table entry and stored in the *doc_map* file.

4.2.2 Finding the Freshest Result

The primary search changes to get the latest version of a page are implemented in this function. For each document that matches the search criteria, the corresponding iterator first finds the most recent version of its URL stored in the index. Once that is retrieved, the Bloom filter of terms attached to its *doc_map* entry is checked for the search term's membership. The current result document is replaced with the latest version if this check succeeds, and is discarded from the results if not.

The WordIterator class now accepts a flag *$latest_version* into its constructor, which indicates whether the logic to lookup the latest version of a result should be invoked. This is set to true during the initial lookup on a search term by the calling PhraseModel::getQueryIterator function. While iterating over the list of postings in the current generation (and for the current search *term_id*), the Yioop codebase first checks if this *$latest_version* flag is set. If so, the most recent version of each search result has to be looked for. This lookup is done in WordIterator::getDocKeyPositionsScoringInfo. The first 8 bytes of a fetched *doc_key* entry for each posting are accordingly extracted into *$url_hash,* which represents the hash value of that document's URL. The value obtained in *$url_hash* (prepended with the meta word *info:*), as

well as the current index being searched in, are both then passed into IndexManager::lookupLatestVersionPage. The IndexManager class maintains a cache of the latest 1000 hashes of URLs searched on mapped to their latest result versions to speed up search called *$urls_cache*. If the *$url_hash* does not exist in the cache, it is then passed into ParallelModel::lookupSummaryOffsetGeneration to get a list of all the occurrences of that URL in the index.

The ParallelModel::lookupSummaryOffsetGeneration function has also been tweaked to accept an additional flag argument called *$latest_version_lookup*, which serves the same purpose as the aforementioned *$latest_version*. If this flag is set, the list of postings (by generation) corresponding to the hash of the URL passed to the function is returned. Back in WordIterator::getDocKeyPositionsScoringInfo, the obtained postings list is scanned to get the last entry (*$latest_entry*), which corresponds to the most recent crawled version of the URL. If this is the same as the current posting/generation combination, it is skipped as the current posting itself is the latest version of the page.

From *$latest_entry*, we find the *doc_map* entry associated with that posting and extract the Bloom filter string embedded in it (the 125 bytes after the *doc_key*). We then check if the current *term_id* is present in that string. If not, the posting is skipped, since this means that the term doesn't exist in the latest version of the page and should thus not appear in the SERP. If the document contains the current term, the term must appear in the postings file for the generation associated with *$latest_entry*. So we get the postings entry in that generation for the current term and find the posting with the *doc_map* index value matching that present in *$latest_entry*. This posting entry is used to obtain the positions list and frequency of the current *term_id* in the document, which is stored in *$latest_posting*. Finally, *$posting* is replaced with the *$latest_posting* values for generation, *doc_map* index, positions, and frequency. The logic to fetch the positions file for the current generation and posting has now been extracted into

separate functions WordIterator::getPositionsFile and WordIterator::getPositionsList respectively.

The modifications made to the Yioop codebase to implement this deliverable were tested on the following specifications:

Tested Crawl Sizes
1128038
1793491
1500000

Table 11: Tested Crawl Sizes for Deliverable#2 experiments

Tested Searches
mobile
horse
mountain
goodread book
yahoo
weather.com
site:https://www.google.com/
site:https://www.wikipedia.org/
bestseller no:guess

Table 12: Tested Search Phrases for Deliverable#2 experiments

The following differences in times taken to generate SERPs were observed for the conducted experiments:

Search	Original Time (ms)	Crawl#1 (ms)	Crawl#2 (ms)
mobile	801	846	866
mountain	688	693	693
goodread book	551	688	691
site:https://www.google.com/	1013	1102	1099
horse	440	502	511
yahoo	399	410	502

Table 13: Observations from Deliverable#2 experiments

For most of the tested queries, the response generation time did not drastically increase with this round of changes. For example, *mobile, mountain, yahoo,* and *site:https://www.google.com/* all retained approximately the same response time. For a select few, including *goodread book* and *horse,* the response time increased by around 0.1 seconds for the second index. Overall, the tradeoff between the SERP-generation time and improved result freshness seems to be fair and increases Yioop's search functionality efficiently.

Some examples of updated results captured during testing are:

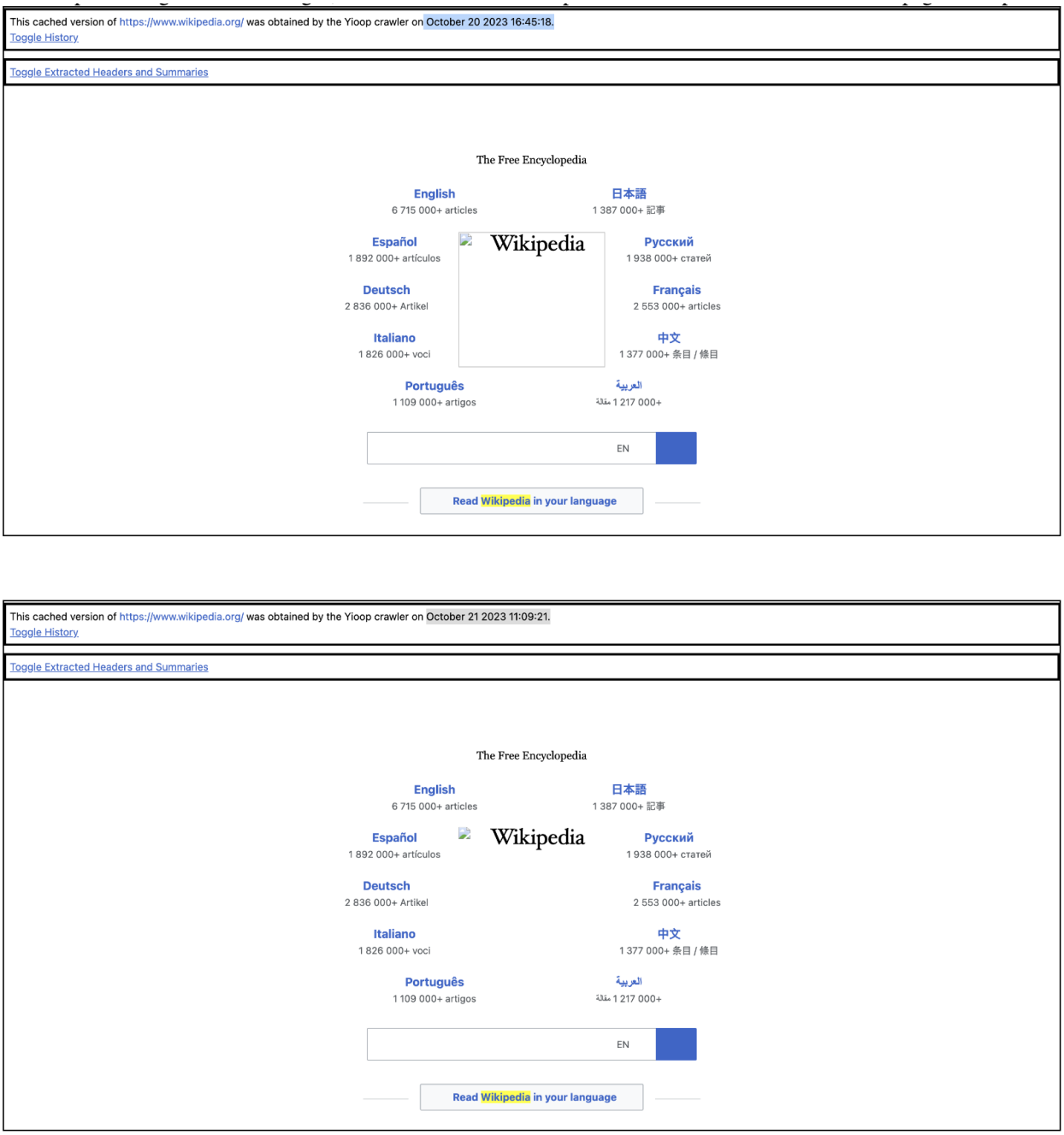
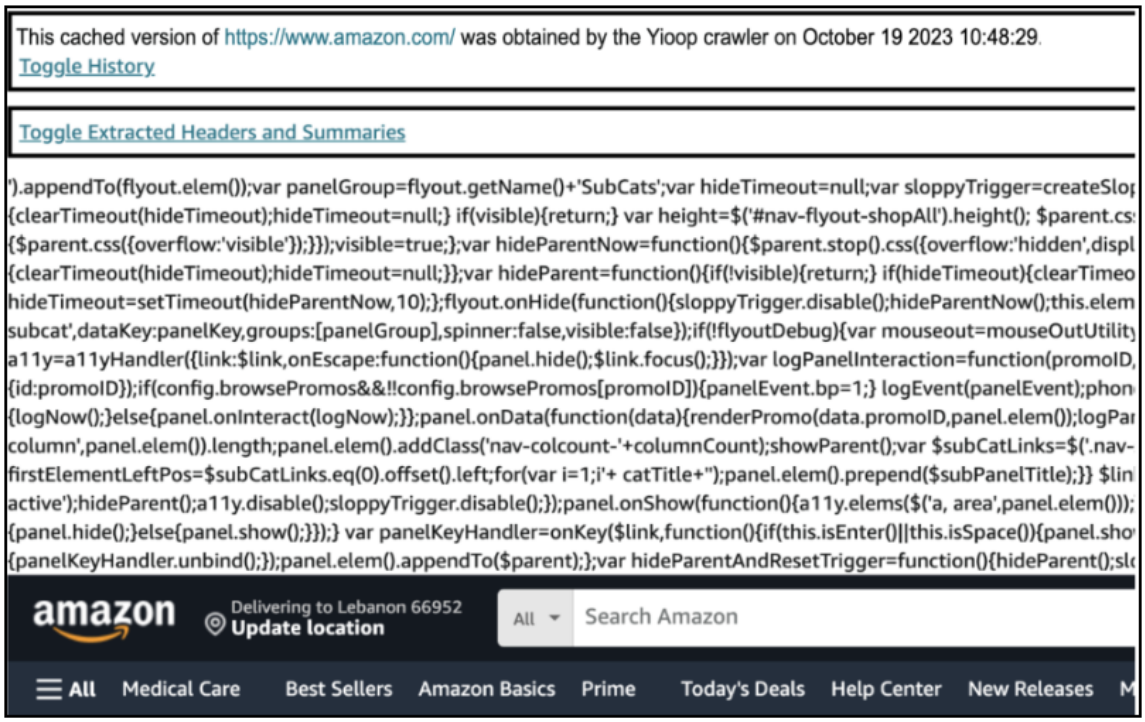


Figure 1: Wikipedia search result crawled date updated



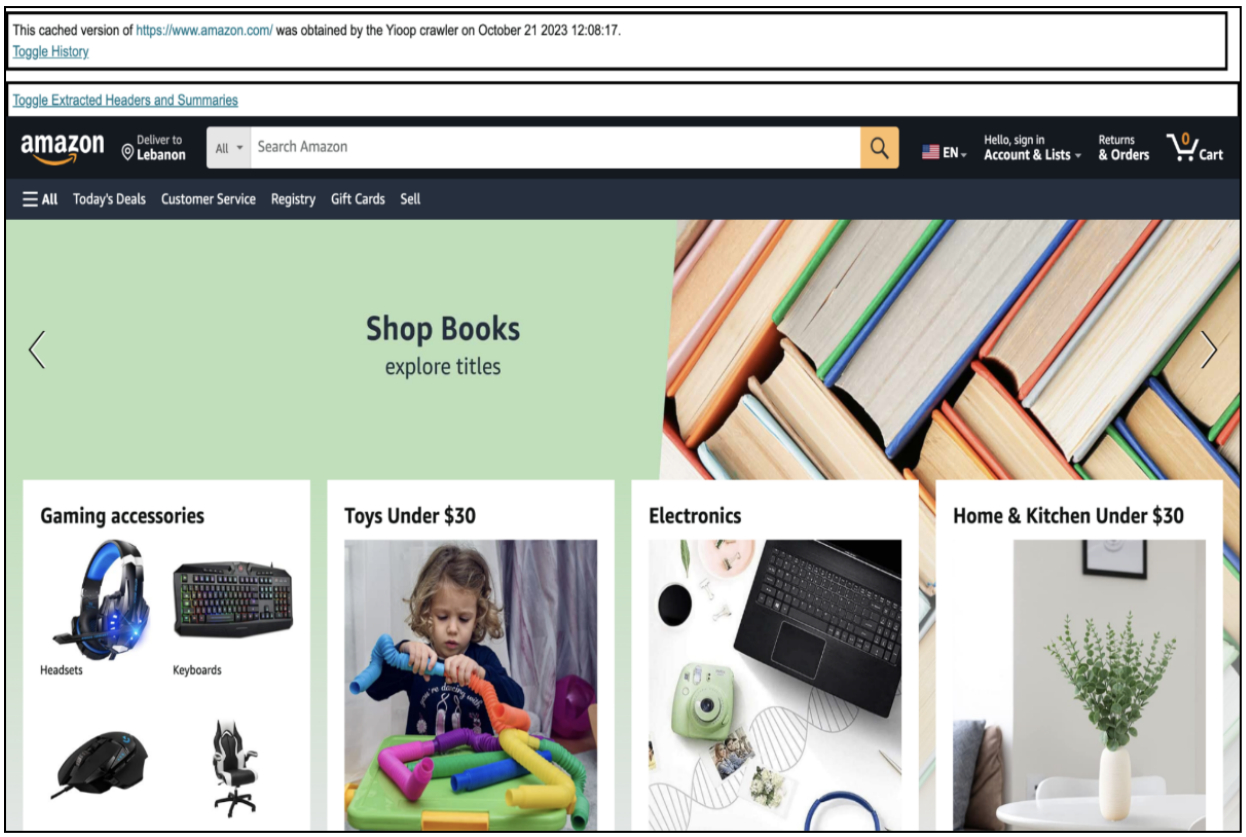


Figure 2: Amazon search result crawled date updated

4.3 Improving Yioop Queries

The final deliverable of this project was to bring forth improvements in Yioop's querying mechanism. This target was accomplished by undertaking three avenues of enhancements:

- Converting the current query processing logic to create disjunctive queries instead of conjunctive ones.
- Using a heap to maintain the top *k* documents seen so far while fetching results for a search query, where *k* represents the count of documents that make up the SERP results.
- Calculating the MaxScore associated with each term in the search query and deleting those that cannot make it to the top *k* results to improve on response time.

4.3.1 Using Disjunctive Queries

The modifications made to the codebase for this change are all in PhraseModel::getPhrasePageResults. The objective of this change is to use disjunct, i.e., OR queries instead of the current conjunct (AND) logic to separate search terms. For example, consider the input search query *halloween party october*. The conjunctive query processing works as follows:

- Yioop first detects meta keywords to additionally filter results. For the given query, it adds *lang:en* and *safe:true*, indicating that the detected language for search is English and safe search should be turned on. Thus, the final query is *halloween party october lang:en safe:true*.
- A WordIterator instance is created for each of the constituent terms, and lookup is performed to find documents containing all of the search terms.

Instead, with disjunctive logic, the query processing for the same search query would work in the following manner:

- Similar to conjunction search, Yioop creates the query *halloween party october lang:en safe:true.*
- The query processor then extracts any detected meta keywords from the search query, and splits the remaining search string based on whitespaces into separate terms. A logical OR ('|') symbol is inserted between them to form *halloween* | *party* | *october,* where each term can be treated as a disjunct term.
- The found meta keywords are then appended to each of these disjunct terms to form *halloween lang:en safe:true* | *party lang:en safe:true* | *october lang:en safe:true.*
- This search string can be thought of to comprise of three independent sub-queries.
- An iterator is created for each of the constituent terms, and lookup is performed on each of them separately.
- Finally, a UnionIterator instance is used to combine the matched documents, score them collectively, rank them by overall score, and return the top *k* documents to make up the SERP.

A flag *USE_CONJUNCTIVE_QUERY* was introduced in Config.php to switch between both sets of logic, and is set to *false* by default. To supplement these changes, the user's input search phrase is first passed through a new function PhraseModel::parseWordStructDisjunctiveQuery, which separates the string into multiple disjunct search strings, which are each processed as individual search queries and collated in UnionIterator. The working of the introduced function to support disjunction logic is as follows:

- PhraseModel::parseWordStructDisjunctiveQuery takes in two parameters, namely *$search_phrase* (the input search query) and *$guess_semantics* (a flag denoting whether the search phrase should be used to guess semantic meta-words).

- The incoming query is processed in this manner:

- *$search_terms* stores all of the disjunct query strings.
- Any text occurring between quotes (*"*) is first extracted from the query into a separate string. Further, any words separated by an ampersand (*&*) - indicating a conjunctive query- are extracted into a separate string.
- The remaining terms in the search query are then checked. If any meta words are encountered, they are appended to each of the disjunct queries found.
- Once all of the disjunct queries have been formed, the original search query is passed through PhraseParser::extractPhrases, and the search terms in the original query are modified accordingly.
- Finally, the search string is separated by whitespaces for terms to be treated as individual query strings. PhraseModel::guessSemantics appends any other detected meta words to the disjunct queries.
- Once all of the disjunct phrases are created, they replace the previous *$disjunct_phrases* variable. Each of them is individually passed into PhraseModel::parseWordStructConjunctiveQuery.
- This overlapping logic (of guessing semantics and extracting phrases) is now removed from the conjunction logic. A Word/IntersectIterator is created for each disjunct query as usual.

4.3.2 Using Heaps

The modifications made to the codebase for this set of changes can be found in UnionIterator::findDocsWithWord. Rather than appending matching documents found in each round of retrievals, *$results_heap* maintains the top *k* documents found until then in a min heap, arranged by their relevance scores. *k* here represents the maximum number of results that

a block can return in one go, which is set by the *$results_per_block* value of the umbrella GroupIterator. Two additional heaps are maintained to keep track of the constituent Intersect and/or Word Iterator instances, called *$terms_heap* and *$low_scoring_terms*. *$terms_heap* is in charge of retaining iterator information including its index on the list of iterators on the current UnionIterator, the MaxScore it could achieve, and the next occurrence of the term in the index (called *NEXT_DOC*). The next occurrence is returned by every iterator's currentGenDocOffsetOfWord function, which returns a tuple of its generation (partition) and *doc_map* offset. *$low_scoring_terms* is a heap that retains the same information about the terms which will be removed by MaxScore comparison.

The functioning of the UnionIterator lookup is now such that query processing is done via the aforementioned heap structures. When a UnionIterator is created, *$results_heap* is initialized to size *results_per_block* and each score is set to 0. *$terms_heap* is initialized by iterating over each of the constituent iterators and storing the NEXT_DOC, iterator index (based on the list of iterators on UnionIterator::*index_bundle_iterators*), and MaxScore for each. The heap is arranged based on the NEXT_DOC value. *$lower_scoring_terms* is initialized to an empty heap.

When UnionIterator::findDocsWithWord is called, the current position of the UnionIterator is compared to the NEXT_DOC positions of each of the elements in *$terms_heap*. This basically indicates how many of the query terms are present in the current document being fetched, and its score is updated accordingly. For the appearing terms, the relevance score of each is added to the total score for the document along with its DOC_RANK score. After the score has been calculated, it is compared to the current minimum score in *$results_heap* (i.e., current k^{th} best score of the search results). If the heap isn't full, the document is automatically added to the heap. If full, it is only included in the heap if it’s score is higher, in which case it will replace the current k^{th} best result document. On each insertion operation, the heap is re-heaped via

UnionIterator::heapifyDown to maintain the min heap property. The min heap property states that the determining score of a parent element must always be lesser than that of its children elements.

Assuming that the document fetched makes it to the top search results, the total count of found documents is updated in the GroupIterator. If there are still more results to be found, the UnionIterator instance is advanced to the position of the next document to be retrieved. This advance() works as follows. UnionIterator::currentGenDocOffsetOfWord is used to find the current position of the UnionIterator in the index. Then, *$terms_heap* is cycled over to advance each of the constituent iterators at the same location, and the heap is reheaped each time to maintain the min heap property. Once an iterator has been advanced, the code checks its MaxScore value to identify those that will not make it into *$results_heap*. The idea behind using this upper bound is avoiding the extra fetching of documents containing only terms that cannot contribute to the top results. If the corresponding MaxScore is lower, the term is removed from *$terms_heap* and added to *$lower_scoring_terms*. The latter tracks terms that have been discarded from the terms heap, such that these terms will only be used to calculate the overall relevance score and not the document retrieval itself.

Once the total capacity of a block is reached (or there are no more results found), UnionIterator::getResultsHeap is invoked to retrieve the top results in descending order of score. This updates the pages in GroupIterator::getPagesToGroup, and the next block of documents is set to be retrieved.

4.3.3 Using MaxScore

The modifications made to the Yioop codebase for this improvement are all in UnionIterator::findDocsWithWord, similar to the prior heap change. The UnionIterator constructor now accepts parameters for the current index name (*$index_name*) and total

number of crawled documents stored in the index (*$total_num_docs*). Both of these parameters are used to calculate term MaxScore values.

To accommodate the MaxScore-relevant modifications, a list of search terms is first put together by iterating over all of the nested iterators in the current UnionIterator instance. Every nested iterator is either an IntersectIterator or a WordIterator, and their corresponding getMaxScore() is called to get this value. This list *$terms_heap* is created in UnionIterator::getQueryTerms. For every term, both the MaxScore it could possibly contribute to the final set of results as well as the index of the iterator it appears on in *$index_bundle_iterators* are stored. The MaxScore calculation takes place in the added function UnionIterator::getMaxScoreForTerm. The idea behind calculating a MaxScore is to find an upper bound on the maximum relevance score any document containing the current search term could conjure. Once *$results_heap* is full, a term's MaxScore can be compared to that of the k^{th} best scoring document encountered yet in UnionIterator::compareByMaxScore. If lesser, it can be safely assumed that any document containing only the current query term can never make it to the top *k*, and thus remove that iterator entirely from the terms heap.

The total number of documents in the current index (to find the value of *N* in the MaxScore formula) is determined by the *$total_num_docs* value of the current UnionIterator instance. The N_t value (number of documents in the index containing the current search term) is fetched from IndexManager::discountedNumDocsTerm. This function has been modified to accept a flag parameter *$discount_terms*, which determines if terms should be discounted based on their generation. *$discount_terms* is set to false in the call from UnionIterator::getMaxScoreForTerm to get the total count of occurrences of the current query term across all documents. Once *$terms_heap* has been found, it is used as the bases for comparison while inserting documents into *$results_heap*. Each time the latter is full, the current k^{th} best score (i.e., the result heap's current minimum score) is contrasted to each of the MaxScores in *$terms_heap*. If the

MaxScore of a term is not greater than the current minimum score in *$results_heap*, the word iterator associated with it can safely be deleted from the current terms heap and added to *$lower_scoring_terms*.

After implementing the changes explained above, they were tested on the following setups:

Tested Crawl Sizes
1128038
1793491
1500000

Table 14: Tested Crawl Sizes for Deliverable#3 experiments

The seed sites used for each of the crawls were slightly different, to obtain a comprehensive set of observations. By toggling Config::*USE_CONJUNCTIVE_QUERY,* the search queries were observed to be transformed as shown below:

	Conjunctive Query	Disjunctive Query
1.	*justin trudeau lang:en safe:true*	*justin lang:en safe:true* | *trudeau lang:en safe:true* | *justin-trudeau lang:en safe:true*
2.	*prime lang:en*	*prime lang:en*
3.	*prime minist safe:true lang:en*	*prime safe:true lang:en* | *minister safe:true lang:en* | *prime-minist safe:true lang:en*
4.	*prime-minist no:guess*	*prime-minist no:guess*
5.	*google verizon pinterest safe:true*	*google safe:true* | *verizon safe:true* | *pinterest safe:true*

6.	*“google verizon” lang:en safe:false*	*“google verizon” lang:en safe:false*
7.	*apple & mac lang:en safe:true*	*apple & mac lang:en safe:true*
8.	*weather site:weather.com lang:en safe:true*	*weather site:weather.com lang:en safe:true*
9.	*lang:en media:news w:1 -i:100 #1#*	*lang:en media:news w:1 -i:100 #1#*
10.	*lang:en media:news w:1 -i:100 safe:true*	*lang:en media:news w:1 -i:100 safe:true*
11.	*sand beach california safe:true*	*sand safe:true | beach safe:true | california safe:true*
12.	*chatgpt gpt4 openai safe:true lang:en*	*chatgpt safe:true lang:en | gpt4 safe:true lang:en | openai safe:true lang:en*
13.	*“chatgpt openai” & gpt4 safe:true lang:en*	*“chatgpt openai” & gpt4 safe:true lang:en*
14.	*wildfir usa site:www.usatoday.com*	*wildfir site:www.usatoday.com | usa site:www.usatoday.com*

Table 15: Tested Search Queries in Conjunctive and Disjunctive Forms for Deliverable#3 experiments

For each of the above queries, the differences in SERP result sizes observed are captured as below:

	Crawl#1 Conjunctive	Crawl#1 Disjunctive	Crawl#2 Conjunctive	Crawl#2 Disjunctive	Crawl#3 Conjunctive	Crawl#3 Disjunctive
1.	2	49	0	32	11	81

2.	389	389	501	501	442	442
3.	12	513	9	634	1	501
4.	2	2	2	2	0	0
5.	0	412	0	200	1	290
6.	1	1	0	0	0	0
7.	14	14	21	21	11	11
8.	3	3	1	1	3	3
9.	542	542	564	564	1110	1110
10.	502	502	543	543	1092	1092
11.	0	307	2	399	0	299
12.	0	72	0	26	2	59
13.	0	0	0	0	0	0
14.	1	4	2	6	2	2

Table 16: Observations from Deliverable#3 experiments

From the above results, it is evident that disjunction logic provides a wider range of results as compared to conjunctive logic, which is more selective. This project used human judgement to

gauge the quality of search results. Human judgement factors included comparing search results based on the following criteria:

- Relevance of top 10 search results for the query
- Verified sources and content quality of the top 10 search results for the query
- Overall recency of pages

An indicator of the success of this enhancement was that for all search queries tested wherein the disjunctive and conjunctive structures of the query varied, the top 3 results coming up for the conjunction logic were present in the top 10 search results for disjunction logic. Based on the human judgement factors listed above, the overall quality of the SERP results did not deteriorate by pushing less relevant or unreliable sources of information into the top few matches. Rather, when using disjunctive logic, the overall number of responses generated increased, allowing for more convenient searches for synonyms such as *chatgpt gpt4 openai*.

To quantify the outcome of these changes better, the top few results obtained from both the prior conjunction and new disjunction logic in Yioop, as well as those fetched from Google and Yandex for the same queries was compared. This comparison was done by considering human relevance judgement, where I categorized results as positives or negatives based on my opinion of their quality. The setup was as follows:

Tested Searches
prime minister
apple mac
goodread book
election potus america
california earthquake

Table 17: Search queries for Deliverable#3 comparison experiments

The classification of results was inspired by D. Hawking et al. [13]. This is operating under the assumption that most popular search engines follow, which is that the top 10 results are the most important to a user [14]. For each query, the top search results were classified as follows:

- True Positive: Relevant websites making it to the top 10 results
- False Positive: Irrelevant websites making it to the top 10 results
- True Negative: Irrelevant websites in the top 20 results that did not make it into the top 10 results
- False Negative: Relevant websites in the top 20 results that did not make it into the top 10 results

The term “irrelevant” is misleading in this context: this experiment considers results that did not crack the top 10 (or usually the first page of results) as less relevant to the search query despite being appropriate responses primarily because the odds of them being clicked on are considerably low [14]. The below results are a sum of the corresponding results observed for the 5 test search queries:

	Actual Relevant	Actual Irrelevant
Predicted Relevant	34	16
Predicted Irrelevant	41	9

Table 18: Confusion Matrix for Yioop (Conjunction) Results

	Actual Relevant	Actual Irrelevant
Predicted Relevant	30	20
Predicted Irrelevant	7	43

Table 19: Confusion Matrix for Yioop (Disjunction) Results

	Actual Relevant	Actual Irrelevant
Predicted Relevant	45	5
Predicted Irrelevant	15	35

Table 20: Confusion Matrix for Google Results

	Actual Relevant	Actual Irrelevant
Predicted Relevant	40	10
Predicted Irrelevant	25	25

Table 21: Confusion Matrix for Yandex Results

	Precision	Recall	F1 Score
Yioop (Conjunctive)	0.68	0.453	0.544
Yioop (Disjunctive)	0.60	0.81	0.689
Google	0.90	0.75	0.818
Yandex	0.80	0.615	0.695

Table 22: Observations from Deliverable#3 search engine results comparison

Some deductions that came of this test are the following:

- The precision of Yioop's conjunctive logic surpasses that of the disjunction logic, although the 3 highest-scored results in each SERP generated also appeared in the top 10 results of the corresponding disjunctive logic SERPs for most queries.
- The second page of search results in Yioop's disjunctive logic almost always comprises of documents that are definitely less relevant to the search query (with a low False Positives score). This differs from conjunctive logic, whereas there were several instances of web pages that were possibly relevant enough to make the first page of results being pushed to the second page.

- In case of search terms that (disjunctively) are seemingly meaningful to each other, the outcome of disjunctive logic provided better overall results than conjunctive logic did in the top 20 results put together. Examples of such queries are *apple mac, election potus america,* and *goodread book*.
- For queries such as *prime minister*, the overall top 20 conjunctive logic results were more meaningful than the disjunctive ones, as the latter also had websites related to the term *prime* singularly which covers a wide range of unrelated topics.
- By comparison, Google and Yandex's SERP results for each of the queries more in tune with the expected results. However, as this experimentation of Yioop was done on a limited index (of approximately 1500000 documents), it is unfair to compare the quality of search results.
- The count of overlapping results between the four entities (top 20 pages) contrasted in this experiment are as follows:

Search Query	Google Overlapping Results (Yioop Conjunctive)	Yandex Overlapping Results (Yioop Conjunctive)	Google Overlapping Results (Yioop Disjunctive)	Yandex Overlapping Results (Yioop Disjunctive)
prime minister	10	7	8	13
apple mac	13	15	9	11
goodread book	11	14	10	12
election potus america	6	4	15	13
california earthquake	8	11	6	6

Table 23: Observations from Deliverable#3 search engine overlapping results

CHAPTER 5

CONCLUSION

To conclude with, this project has successfully enhanced the quality of the search functionality in Yioop. The work done over the course of the deliverables explained in the previous chapter has collectively improved the ranking of pages in the final Seach Engine Results Page (SERP) for a user query with the introduction of new bonus factors, ensured that these results are updated and fresh by presenting the most recently crawled versions of websites from the inverted index, increased the total size of the collection of results served for a query with disjunctions, and brought down the overall response time by using the efficient information retrieval concepts of heaps and term MaxScore. The last deliverable has further provided the option of switching between conjunctive and disjunctive query logic to cater to different use cases. Finally, this report has additionally explained crucial aspects of the indexing and lookup processes in Yioop both theoretically as well as by documenting the affected portions of the source code.

REFERENCES

[1] R. Lundbohm. "To Click or Not to Click: A study of tourists social media click behavior on search engine result pages." *Journal of Tourism Quarterly* 4, 2022, no. 1-2: 1-10.

[2] M. King, "Yandex scrapes Google and other SEO learnings from the source code leak," SearchEngineLand, 2023. https://searchengineland.com/yandex-leak-learnings-392393.

[3] M. Siedlaczek, A. Mallia, and T. Suel. "Using conjunctions for faster disjunctive top-k queries." In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*, pp. 917-927. 2022.

[4] C. Shepard, “10 Illustrations of How Fresh Content May Influence Google Rankings”, Moz blog, 2016. https://moz.com/blog/google-fresh-factor-new.

[5] A. Singhal, “Giving you fresher, more recent search results, Google blog”, 2011. https://googleblog.blogspot.rs/2011/11/giving-you-fresher-more-recent-search.html.

[6] M. La Rocca, *Advanced Algorithms and Data Structures*. Simon and Schuster, 2021.

[7] H. Turtle and James Flood. *Query evaluation: strategies and optimizations*. Information Processing & Management 31, 1995, 831–850.

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

[9] Page, L., Brin, S., *The anatomy of a large-scale hypertextual web search engine*. Proceedings of the 7th Intl. WWW Conf., 1998, pp. 107–117

[10] G. Amati and C. Rijsbergen, “Probabilistic Models of Information Retrieval based on Measuring the Divergence from Randomness”, ACM Transactions of Information Systems (TOIS), 2022.

[11] C. Pollett, "Lecture Slides for CS267, 2022", San Jose State University, Accessed: October 23, 2023. [Online]. Available: https://www.cs.sjsu.edu/faculty/pollett/267.1.22s.

[12] B. Bloom. "Space/time trade-offs in hash coding with allowable errors." *Communications of the ACM* 13, no. 7, 1970. pp 422-426.

[13] D. Hawking, N. Craswell, P. Bailey, and K. Griffihs, "Measuring search engine quality." *Information retrieval* 4, 2001. pp 33-59.

[14] Chitika Insights, "The value of Google result positioning." Chitika Inc., 2013. http://info.chitika.com/uploads/4/9/2/1/49215843/chitikainsights-valueofgoogleresults positioning.pdf