Spring 5-8-2020

Improved User News Feed Customization for an Open Source Search Engine

Timothy Chow

Improved User News Feed Customization for an Open Source Search Engine

A Project Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Timothy Chow

May 2020

The Designated Project Committee Approves the Master's Project Titled

IMPROVED USER NEWS FEED CUSTOMIZATION FOR AN OPEN SOURCE SEARCH ENGINE

by

Timothy Chow

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2020

ABSTRACT

Yioop is an open source search engine project hosted on the site of the same name.It offers several features outside of searching, with one such feature being a news feed. The current news feed system aggregates articles from a curated list of news sites determined by the owner. However in its current state, the feed list is limited in size, constrained by the hardware that the aggregator is run on. The goal of my project was to overcome this limit by improving the current storage method used. The solution was derived by making use of IndexArchiveBundles and IndexShards, both of which are abstract data structures designed to handle large indexes. An additional aspect needed to accomodate for news feed was the ability to traverse said data structures in decreasing order of recently added. New methods were added to the preexisting WordIterator to handle this need. The result is a system with two new advantages, the capacity to store more feed items than before and the functionality of moving through indexes from the end back to the start. Our findings also indicate that the new process is much faster, with insertions taking one-tenth of the time at its fastest. Additionally, whereas the old system only stored around 37500 items at most, the new system allows for potentially unlimited news items to be stored. The methodology detailed in this project can also be applied to any information retrieval system to construct an index and read from it.

ACKNOWLEDGEMENTS

I want to first thank my project advisor, Chris Pollett, for his guidance throughout this project. Without his work and dedication on Yioop, there would be no search engine or project for me to work on.

TABLE OF CONTENTS

TABLE OF FIGURES

Chapter 1

Introduction

Within the past few decades, the Internet has grown by leaps and bounds into this huge repository for information. One particular area that is of big interest to many people is news. According to a study done by the Pew Internet and AmericanLife Project, 61% of American get their news online from the Internet on a typical day. This research was done all the way back in 2010, so taking into account the growth of technology in the present day, it would be safe to assume that even more people do so now. While some people may only go to one particular site for news consumption, a greater amount tends to visit several as a way of collecting news from a broader scope. However, it is inconvenient to visit many sites in a day just for this purpose. As a way of overcoming this, content aggregation was brought into existence. For this project, we focus on one particular content aggregator, called Yioop. In this report, we will discuss the particulars of Yioop structure, how it performs this feature currently, and how the existing implementation is improved using a better means of storage.

Content aggregation is at its core a simple idea: instead of making a human do the work of going to different sites, we could have a machine or a system to automate this process instead. The collected results could then be neatly presented to the user in one place. One of the first, most well known examples of a content aggregation was Yahoo! News, which was created by Yahoo! as far back as 1996, and at the time, it changed how people perceived news consumption via non-traditional means, such as newspapers. This also brought about a change of how content on sites was stored as it was desirable to make the aggregation process as simple as possible. This is known as web syndication.

The particular technologies that go into web syndication will be further detailed in Section 2.1, but the gist of it is that there are different formats for sites to display news items. These formats are not so much intended for human eyes, as it is for programs to read and pull from them. Once pulled, these items are also stored on the web aggregation system, as opposed to being discarded immediately after the next update. To see why, imagine being a casual user that is using one such service yourself. After quickly browsing the headlines during the morning, you see a particular title that interests you, although you have no time to read it now. In the evening, you finally find the time, but now the list of articles has already refreshed leaving you unable to remember which one it was. While a favorite or “saved” tagging system could help solve this, it would only work best in conjunction with having all previously seen items stored. A large part of this report will focus on how exactly are these items stored for future retrieval, and what kind of optimizations have been done in order to allow for better performance and scalability.

The rest of the report will be organized as follows. Chapter 2 will explore some of the other large news feed aggregators that exist. Chapter 3 and 4 will detail the two main aspects of Yioop, indexing and storage, respectively. This should lead into the implementation of the changes I have made to Yioop in Chapter 5. In Chapter 6, we will sample the steps that an end to end process would take. Chapter 7 will describe the testing methodology used and finally round things off with the conclusion in Chapter 8.

Chapter 2

Preliminaries

Before going further into discussing Yioop, we will explore the methodology of online feed aggregation, as well as some of the other feed aggregation systems that currently exist, as well as what innovations they bring to this field.

2.1 Content Aggregation

Traditionally, website content is stored using HTML (HyperText Markup Language). In HTML, we use tags and attributes to define the layout and structure of a web page. While it provided a simple and easy way of creating these pages via tag usage, its focus was primarily on how that data should be displayed to a user using a web browser. Data that was stored in a format more suited for carrying and sharing data came to be known as a web feed. Examples of web feed formats include XML (Extensible Markup Language), YAML (YAML Ain't Markup Language) and JSON (JavaScript Object Notation). While each makes use of different syntax, each one attempts to store data in a structured format. The very commonly used RSS feed is also extended from XML. A generated RSS document will usually contain a summary of the text and metadata pertaining to each item in the feed. Any given feed generally works using the pull strategy.

Before the start of aggregation, a list of feeds is provided to an aggregator program. This aggregator periodically checks each feed for new content, which automatically gets stored on the system itself. Then a user can pull from the aggregator, usually from a web application, for an overview of the collected items. Some aggregators will have their own private list of feeds, such as Google News, while others let the user customize which specific feeds to subscribe to.

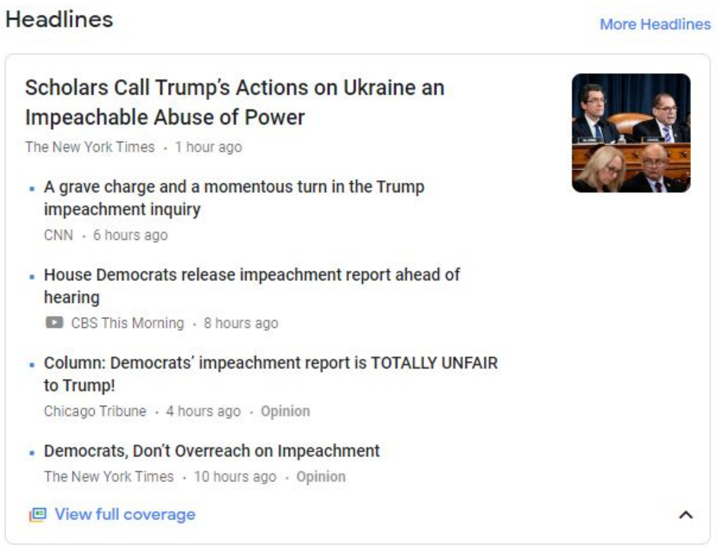
Today there are many sites that support both aggregating feeds as well as storing their own content as a feed for other aggregators to easily take in. Some systems have even moved beyond aggregation, opting to produce their own content on top of collecting feed items from other sites. For instance Yahoo! News, which used to lead the pack when it came to feed aggregation, now has original stories in which they hire their own staff to produce news. Regardless of this, when we discuss news feeds and feed aggregation, we will focus exclusively on the topic of collecting from other sources as opposed to creating our own.

2.2 News Ranking

Once these items have been pulled and stored, there still needs to be some way to sort out which items are presented in what order. Generally for a search engine, we might have some kind of scoring system that is based on relevancy to rank the items. Often this is calculated by taking the terms of a search query and comparing its frequency among all documents. Some algorithms, like Google's PageRank, will have other factors to determine the importance of a given site. A news feed system does not necessarily work off of searching however., as it is unnecessary to search to get the top stories of the day. One of the major contributing factors to ranking in feeds is the age of a story, also referred to as its freshness. As one might expect, stories that are more recent are more important as opposed to something that might have happened a year ago.

Other major factors that could come into play for ranking include its clustered weight score, which is where we see how often news about this subject has been reported on recently. We assume that if more outlets are covering this subject, then it must be important. Source authority is another prominent feature where we consider if one source is more reputable than others. There are even some more intricate systems which will try to determine the temporal freshness of a given story, and this can include some in depth features such as whether a story covers enough new content compared to previously seen stories on this subject, whether dates mentioned inside a story are relatively recent, or how many times a query that would return this story has been made within a period of time. All of this does not even take into account how the feed should be personalized for each individual user.

2.3 Google News


Headlines

More Headlines

Scholars Call Trump's Actions on Ukraine an Impeachable Abuse of Power
The New York Times · 1 hour ago

- A grave charge and a momentous turn in the Trump impeachment inquiry
  CNN · 6 hours ago
- House Democrats release impeachment report ahead of hearing
  CBS This Morning · 8 hours ago
- Column: Democrats' impeachment report is TOTALLY UNFAIR to Trump!
  Chicago Tribune · 4 hours ago · Opinion
- Democrats, Don't Overreach on Impeachment
  The New York Times · 10 hours ago · Opinion

View full coverage

Figure 1. Clustering of Stories in Google News

Most people are likely familiar with Google's widely popular and famous search engine. The traditional Google system is famous for pioneering the PageRank algorithm as a form of relevancy ranking for search results. Google also has a news aggregation feature built in, but this actually ditches PageRank in favor of ranking based on other features. These include user clicks, an estimated authority of a publication for a given topic, freshness, and more. Each item, which is referred to as a story, is ranked in order of a perceived interest. For instance, what makes a story interesting, what topics are people looking into, what do editors feel is a top story, etc, are among the questions that Google asks themselves when deciding how to rank stories. Stories themselves are clustered into subjects or topics, and there is an attempt to provide a broader view of topics by using stories from different sources as different angles into an event. Google will also take into account a user’s previous search queries when deciding which stories will get returned. This entire process is also completely automated, meaning there are no editors that are sitting around ranking stories. The entirety of their system is built to run purely off the chosen ranking features.

2.4 Facebook News Feed

Facebook has a news feed aggregator which uses their own ranking algorithm as well. In in, Facebook defines four major steps of ranking and retrieval: inventory, signals, predictions, and relevancy scores. In the first step, inventory simply collects stories that a user has not seen yet. After inventory, Facebook produces signals about the current state. Signals represent every bit of information that they deem necessary to choose which stories are relevant to the user at this moment. They include data ranging from the age of a story to little things such as how fast the user’s internet connection is. Using this, Facebook makes a prediction on how the user will react to the story. For instance, the user might be likely to share this story or they might just ignore it entirely. From all this data a relevancy score for each story is produced which represents how interested this user will be in this one story that was found in his/her inventory. A large focus is put on generating a news feed that Facebook thinks the user will enjoy following.

2.5 RSS feed aggregators

In contrast to the very complex systems behind Google News and Facebook News Feed, we have regular RSS feed aggregators. Whereas the previous two are engineered to produce specialized results for each user, the primary goal of normal RSS feed aggregators is to simply mix in results from multiple RSS feeds. Customization is largely left up to the user's discretion, as they will have to manually choose which feeds to subscribe to and what kind of items might be filtered. These programs can also be either a web based application or it could be standalone, with a focus on interfaces that are clean and easy to use. Yioop is most similar to this style, in that it does not try to do anything too fancy with the gathered feed items, but it does support aggregation across multiple web feed formats plus a searching mechanism that does do some form of document ranking, albeit not geared towards any particular user.

2.6 Trending words

While Yioop does not support the clustering system which Google News employs, there is some functionality that could be used to work towards this feature. While visiting Yioop, one can see the top trending words within a current time range. These words are simply selected by detecting the most common words that are seen during a single news feed update. The words and their counts are saved and some light statistics are calculated and displayed. Users can even click on each word, which does a search for all news feed items that contain this word. Trending gives us a straightforward way of grouping up feed items by terms, and the terms themselves are ranked in order of occurrence, i.e. how trending they are. The purpose behind such a feature is mostly for search engine optimization, or SEO, in which we try to understand what exactly is relevant content, what stuff are people likely to search for, and what terms they might be using to search for it. As it is now, it can serve as a very rudimentary clustering system, but there would need to be more work done to calculate relative similarity between any given article with the same trending term. Additionally, this only works as a general non-specific system, whereas Google News is curated to each individual user.

Chapter 3

Yioop and Indexing

3.1 Yioop

Being the focus and the basis of the entire project, we will briefly introduce the system that is Yioop. Yioop is an open source search engine that is designed for the core task of crawling the online web, archiving it, and using the crawled archive to allow users to search the web. In doing so, Yioop creates an index for each site that it visits. The process of indexing the site includes downloading the web page, generating a posting list, and extracting a generalized summary. We will talk more in depth on the specifics of how Yioop does indexing in a later section. Like other search engines such as Google or Bing, users can interact with Yioop through querying the generated index, but unlike either product the entirety of Yioop can be downloaded by the user and its parameters configured so that one can have a personalized crawl of the web. These options include limiting the scope of the crawl by providing a specific list of sites to be crawled as well as how much depth should Yioop go into when hopping from one site to another. This allows for greater flexibility and control over the exact results that are returned. In comparison, Google does allow a user to restrict search results to a specified domain, but only offers searches over their whole index. On one hand this provides users with probably the most complete search index out of all search engines, likely encompassing the near entirety of the surface web. Google is able to achieve this, thanks to their much larger scale of operation and hardware, whereas Yioop has nowhere near the same manpower or finances backing it.

3.2 Yioop Indexing Process

Since the primary goal of this project is to enhance the pre-existing Yioop newsfeed functionality, it would be prudent to get a better understanding of how things work under the covers. As mentioned above, one of the primary functions of Yioop is to crawl web pages, store them, and create an index on which users can run queries. In this section, we will break down the exact steps and components that make this possible, from start to beginning.

Yioop is built on a distributed computing framework consisting of name servers and queue servers. The reason behind this structure for Yioop is that it allows for easy deployment and scaling even for individuals or small businesses. Name servers act as a node in a Yioop system which helps coordinate crawls. Each node can have several queue server processes which serve two main purposes, one which is scheduling jobs and the other which is indexing jobs. In addition to queue servers, there are also fetcher processes that help with downloading and processing pages from the crawl. Crawling and indexing newsfeed items are considered a separate job from normal crawls, thus they do not use the exact same fetcher and queue server processes. However, the methodology that is used is inherently similar and so we will discuss the process as if the two processes were used. Any differences between the two will be highlighted later on.

3.2.1 Crawling

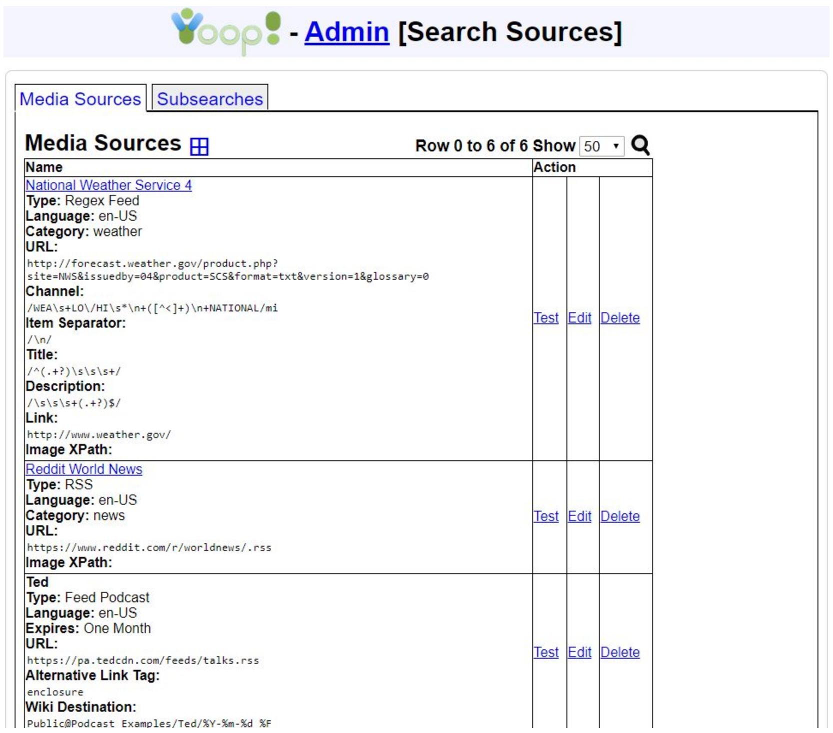
During a crawl, we initially have to set up a list of sites that will be crawled, which can be done through Yioop's web app interface. From this list, the fetcher processes will create a schedule that holds the data that will need to be processed later on, as well as the type of processing required. It does this by periodically pinging the queue server for the list of web pages, downloading them, and then creating a summary on the page. A summary can be seen as a shortened description of the page that holds various data that is used for later steps in the indexing process. In order to differentiate pages on a machine level, a unique hash id is created for each page. Additionally it is here that the inverted index construction is started.

3.2.2 Indexing

When reading a book, one might notice a section called the index at the end that stores a list of words as well as the page numbers that the word appears in. In a similar way, the inverted index that Yioop makes is generating a term to document mapping, where each document is stored as an id representing a web page and terms are words or groups of words on that page. On top of storing the document id that the term appears in, the inverted index also stores the exact position of the term on that document. This document id and position combo is referred to as a posting, and each term will have its own posting list. There is a two fold benefit in constructing such a structure: the first is that the index will certainly be smaller in file size than the document itself, and secondly it allows for faster lookup when compared to certain scores for page ranking purposes are also calculated during this step, and this is all saved in a mini inverted index which is POSTed to queue server. As the queue server receives the data, it attempts to merge these mini inverted indexes into a larger index structure along with the summary data and downloaded page data. After a single round of this, the crawl can be stopped and we can immediately perform queries using the produced index.

3.2.3 News feed Indexing

In the case of the news feeds, it is handled by a process called MediaUpdater, which in turn deals with media jobs. The specific media job that we are interested in is the FeedsUpdateJob. Whereas the queue server and fetcher setup continuously crawls the web non stop, MediaUpdater only runs during a set interval, which can be user configured. Additionally, the queue server is designed to crawl with depth in mind, where links found within crawled pages will also be crawled later. In comparison, media jobs rely on a list of source sites in the database.


- Admin [Search Sources]

Media Sources | Subsearches

Media Sources

Row 0 to 6 of 6 Show 50

Name	Action		

National Weather Service 4
Type: Regex Feed
Language: en-US
Category: weather
URL:
`http://forecast.weather.gov/product.php?site=NWS&issuedby=04&product=SCS&format=txt&version=1&glossary=0`
Channel:
`/WEA\s+LO\/HI\s*\n+([^<]+)\n+NATIONAL/mi`
Item Separator:
`/\n/`
Title:
`/^(.+?)\s\s\s+/`
Description:
`/\s\s\s+(.+?)$/`
Link:
`http://www.weather.gov/`
Image XPath:
	Test	Edit	Delete

Reddit World News **Type:** RSS **Language:** en-US **Category:** news **URL:** `https://www.reddit.com/r/worldnews/.rss` **Image XPath:**	Test	Edit	Delete

Ted
Type: Feed Podcast
Language: en-US
Expires: One Month
URL:
`https://pa.tedcdn.com/feeds/talks.rss`
Alternative Link Tag:
`enclosure`
Wiki Destination:
`Public@Podcast Examples/Ted/%Y-%m-%d %F`
	Test	Edit	Delete

Figure 2. Interface to configure media sources from which feeds are retrieved.

Each source can be one of five types: an RSS feed, JSON feed, HTML feed, Regex feed, or podcast. In order to function correctly, all parameters must be set up correctly when adding a new source to the list. Each source itself is intended to be a feed itself, where new items are added to it over time.

```
    [title] => American flight bound for Miami diverted after woman fakes medical condition, police say
    [description] =>
        An American Airlines flight had to be diverted on Friday after a passenger attempted to fake a medical condition to get
a bigger seat on the flight.


    [link] => https://news.yahoo.com/american-flight-bound-miami-diverted-141039018.html
    [guid] => american-flight-bound-miami-diverted-141039018.html
    [pubdate] => Mon, 02 Dec 2019 09:10:39 -0500
    [image_link] => http://l2.yimg.com/uu/api/res/1.2/KlEsfV9j1sIWVNJCHNGMIQ-
-/YXBwaWQ9eXRhY2h5b247aD04Njt3PTEzMDs-/http:/d.yimg.com/hd/cp-video-transcode/1009217/917f24d6-5b1b-419d-a1e9-
beeb70a15946/2e7a9f3f-7089-5962-bfdc-a8be763303e9/data_3_0.jpg?
s=f0f9191ec401536cdd141d00b9216b50&c=e10c644e0ef637d6a3e8445e44d3e2e2&a=tripleplay4us&mr=0
```

Figure 3. A feed item is converted into an array to be stored on the database.

What the news feeds functionality for Yioop does is aggregate all of these items. For each source that we have, FeedsUpdateJob grabs the latest items on the page and adds it to a database. Here, certain processing such as deduplication, sorting by publication date, and calculating trending words is performed on these items. The results from this database are added into a single index, which gets replaced with every run of FeedsUpdateJob. This is a limitation of the current implementation of Yioop's news feed functionality, as it means that there is an upper limit to how many feed items end up getting stored. The goal of this project will be to overcome this limitation and allow for a scalable storage for news feed items.

Chapter 4

Yioop Storage

Thus far in this report, we have discussed the workflow and processes that control crawls and indexing. Here we will explore the crux of this project, which is how storage is actually done in Yioop. Earlier there was a mention on the inverted index, which maps terms to documents. In a general setup, we might imagine a hash map or a dictionary, where the key is a hashed form of the term and the value is the posting list. Since posting lists contain most of the information for the index, we cannot usually store all posting lists in memory and must read them on an as needed basis. Yioop handles this by using several data structures, which will now be explained in more detail.

4.1 IndexShards

IndexShards are the lowest level data structure for a particular index. A shard has two access modes: a read-only mode and a loaded-in-memory mode. While the shard is loaded in memory, data can also be in a packed and unpacked state. New data can be added to this shard during the unpacked state, with it only changing to packed when the data is ready to be serialized to disk. Each shard consists of three major components: *word_docs* entries, *doc_infos* entries, and *words* entries.

- The *doc_infos* is a string structure that holds document ids along with a summary offset, and the total number of words that were found in that document. Each record starts with a 4 byte offset, followed by 3 bytes to hold the length of the document, followed by 1 byte containing the number_doc key strings, finally followed by the actual doc key strings themselves. A doc key string is 8 bytes containing the hash of the URL for a page plus a hashed stripped version of the document.

- The *word_docs* entry is a string consisting of a sequence of postings, where a posting is an offset into a document for which a term appears in, plus the number of occurrences of that term in that document. It is only set when an IndexShard is loaded up and in a packed state.
- Bringing both components together is the *words* entry, which is an array of word entries that is being stored in this shard. In a packed state, each word entry is made up of the term id, a generation number, an offset into the word_docs where the posting list of this term is stored, and how long that posting list is. In the unpacked state, each entry is just a string representation of a term and its associated postings. Incidentally, many of these items are stored as strings as they have been found to be more memory efficient than associative arrays, at least in regards to PHP. When serialized to disk, a shard produces a header which holds document statistics followed by a prefix index into the *words* component, followed by the actual *words*, *word_docs*, and *doc_infos* components in that order.

Indexing and storing items mostly takes place in one method of IndexShard, fittingly called addDocumentWords(). In this method we assume that we have just finished processing a page in a fetcher process, and those processed results are being used as the arguments for this method. The first of these arguments which are relevant to us is *doc_keys*, which is a string of concatenated keys for a document. Keys for a given document can include the hashed id and the host url of a link. The second important argument passed in is the *word_lists* array, which is a regular associative array of terms to positions within a document. During this storing process, terms are hashed and the positions are converted into a concatenated string before being added into the *words* array. Additional parameters are also stored into the shard at this point including meta words, description scores, and user ranks. We can also define the summary offset for this document at this point, but we do not have all the necessary information right now to calculate this, so this is ignored for now. This field eventually gets updated later on during the process of adding to an IndexArchiveBundle.

4.2 IndexArchiveBundle

An IndexShard by itself could realistically store any amount of postings. However, it was explained earlier that if an index gets too big, then it becomes unwieldy to read into memory. Because of this, an IndexShard has a limit of how many documents it can store, which is calculated based on how much memory the machine running the crawl has. To get around this problem is a very simple solution: if we can't make a big

IndexShard, why not make multiple IndexShards instead? Yioop supports this by calling each shard a generation, and when a shard is full, then a new generation, a new shard, is created. Handling this is a separate data structure, which is called an IndexArchiveBundle.

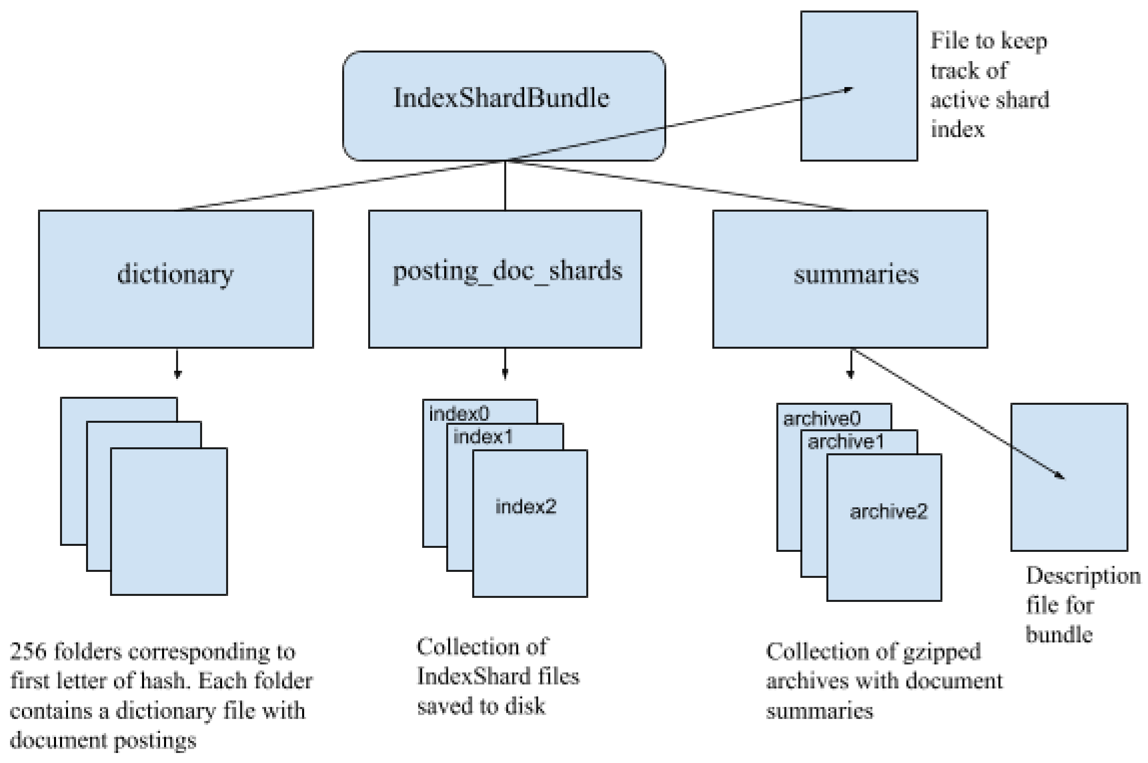

IndexShardBundle
File to keep track of active shard index
dictionary
posting_doc_shards
summaries
index0
index1
index2
archive0
archive1
archive2
Description file for bundle
256 folders corresponding to first letter of hash. Each folder contains a dictionary file with document postings
Collection of IndexShard files saved to disk
Collection of gzipped archives with document summaries

Figure 4. A diagram overview of how an IndexShardBundle is structured.

In truth, the main function of an IndexArchiveBundle is to help store multiple IndexShards and facilitate retrieval on them. The first step to understanding IndexArchiveBundles is to observe the file structure that it generates. Each bundle is stored as a folder with several sub folders: a dictionary folder, a posting_doc_shards folder, and a summaries folder. Additionally a text file stating the current shard in use for indexing is stored. The posting_doc_shards folder is relatively straightforward as it simply holds all of our IndexShards saved to disk. Looking into the dictionary folder reveals something much more interesting as there are 256 different folders. This dictionary folder is actually handled by yet another data structure, this time named IndexDictionary.

4.3 IndexDictionary

As one might expect, the IndexDictionary acts like a dictionary, storing records in the form of: word id, index shard generation, posting list offset, and the length of the posting list. All of the words that had been added to IndexShard will be found here, and in some cases there may be more than one record per word since the word might appear in multiple generations. Each one of the sub folders is used to store the hashed word ids that begin that certain character, hence 256 different folders, and inside each sub folder is a dictionary file representing the data being stored. The last enigmatic folder included with each IndexArchiveBundle is the summaries folder. This folder actually only stores two things, a description file, and a web archive. The description file holds most of the basic parameters for a given bundle; for instance, a user defined descriptor used as a label for humans to read and the number of documents stored in total for this IndexArchiveBundle. The description is mostly trivial in that the information can be extrapolated elsewhere, but it is used as a quick way to grab information about a bundle and display on the web app. The web archive is where the summaries that were mentioned earlier are stored. Each summary is all added into one text file, which is then compressed using the standard GNU zip (gzip) compression algorithm.

Now that we have thoroughly gone off track by discussing the IndexDictionary class and its particulars, we can focus back onto how IndexArchiveBundle is coordinating these steps. Picture that we have just crawled a set of pages and produced an IndexShard. This shard could be either full or it could be close to empty. The first thing that IndexArchiveBundle will do is see if the most recent IndexShard in the bundle has enough space to store this new shards data. If there is enough space, then we can simply merge the two shards together. Otherwise we will save the current active shard, perform the necessary additions to the IndexDictionary, and start a new generation using the new shard. Earlier it was mentioned that the addDocumentWords() method allows for summary offsets to be set, but we held off on it at that point as we did not actually have summaries to reference at that point. Instead we now add these offsets back in, but not before adding the summaries to the web archive first. Once each summary has been safely added to the appropriate summaries file, we can generate an associative array of document ids to the offset within the summaries file. This array is passed back into an IndexShard, where each entry within the *doc_infos* is finally updated with a correct offset value. Now that the finishing touches have been put on the IndexShard, we can finally add it into the IndexArchiveBundle.

The current implementation of the news feed makes use of only one IndexShard, but it does not make use of the IndexArchiveBundle or IndexDictionary. Instead it is stored as a solitary index file. Besides size limitations, this also requires special concessions to be made if we want to use items from the index for anything outside of news feed. We could simply alter the code such that we utilize IndexArchiveBundles as well, but the other problem has to do with how existing records are stored and retrieved. For a given news feed, it would be logical to have the more recent items appear before older items. We will refer to new items as having a higher freshness value attached to them. However, recall the current process of crawling and indexing. Sites that are seen first will have priority when being stored. Retrieving will similarly start at the beginning of each index, meaning that fresh items will be harder or, in most cases, impossible to get.

To extend the use of IndexArchiveBundles for news feed items, there were two proposed solutions. The first is to alter the current index construction methods. Instead of adding new items to the tail end of the index, we will prepend them to the head. The second way was to alter the methods associated with traversing index, such that we can go either forwards or backwards. By doing it this way instead, index construction would remain largely the same; however, the bulk of the work would lie in modifying retrieval such that it obtains records in a reverse format. For this project, the solution that was picked was the second one, so let us also go over how exactly index traversal works.

Chapter 5

Implementation

Since we have decided to proceed with modifying the way Yioop goes over indexes, our implementation will instead largely focus on classes and methods related to that process. There will be some modifications that need to be made to the existing news feed index construction, but those differences are comparatively small. Additionally, we will need to alter other parts of Yioop to both accept the results of a reverse traversal and to properly read IndexArchiveBundle during a news feed instead of a single IndexShard.

5.1 IndexShards, but in backwards

There should be no surprise that any change to index retrieval would start in IndexShards, as they hold the actual index data within their files. In a standard information retrieval setting, we usually define several methods for an abstract index data structure which would allow us to access the posting list. These include:

- *first(t)* returns the first position at which the term *t* occurs in the collection.
- *last(t)* returns the last position at which the term *t* occurs in the collection.
- *next(t, current)* returns the position of the first occurence of *t* after the position *current* in the collection.
- *prev(t, current)* returns the position of the first occurence of *t* before the position *current* in the collection.

The first two methods are trivial to implement, as they only require that we know the start and end of each posting list. Presently, only the *next()* method is supported in Yioop, so ideally we also want this *prev()* method for reading in reverse. It should be noted that there are also slight differences in the implementation for Yioop. For instance, no specific term is specified during a *next()* call. Instead the term is processed earlier in a method called *getPostingsSliceById()*. Here a postings slice refers to an array of postings for the given term and any positional information is assumed to be stored as byte offsets. In this method, we make use of two additional helper methods, the first of which is *getWordInfo()*. *getWordInfo()* is a method that retrieves the starting offset and the ending offset of a posting list for a given term. It does this by first converting the term into its hash and then looking it up in the IndexShard's components. If the shard has already been stored on disk, then we first read blocks of the shard into memory. Otherwise we can work directly with the *words* component. Once this is finished, we go into the second method, *getPostingsSlice()*.

5.1.1 getPostingsSlice()

In *getPostingsSlice()*, we are provided with the offsets found using *getWordInfo()*. We also can specify a *next_offset* argument, which is functionally similar to the *current* argument described in the abstract data structure. Since we know the offsets of the posting list, we can actually derive other pieces of information as well. For instance, we can be sure that every posting that we get will be the term that we are searching on. We can also determine the exact number of postings in each slice, as we assume that each posting will only have a size of 4 bytes. *getPostingsSlice()* starts by checking if the *next_offset* is greater than the ending offset, as that would mean we have exceeded the entire posting list. If that is cleared, then we create a variable that stores our current offset, similar to a pointer in an array. We can access the posting at this offset using a method called *getPostingAtOffset()*, which when given an offset will return a substring from the *word_docs* component which is a posting. The current offset value is then incremented and the process loops until we reach the end of the posting list. In order to go backwards, we start off the same way by getting the start and end offsets. If *next_offset* is unset, then we set the current offset to the end offset at the beginning of reading. Most of the work comes in manipulating this current offset such that it decrements, and it checks to see if the offset has gone below the start offset, as opposed to going above the end offset.

5.1.2 nextPostingOffsetDocOffset()

A method that is not called internally by an IndexShard but is still used for reading the index is *nextPostingOffsetDocOffset()*. Whereas *getPostingsSliceById()* returns all the postings for a term, this method returns an offset tuple containing the offset to the next posting and the offset for the document that contains this posting. It takes the same two arguments, a start offset and end offset, but instead of an optional *next_offset*, we will specify a *doc_offset*. What this method is trying to do is return the first posting offset between the start and end offset and is in a document whose specified offset is either equal (meaning that we want the next offset in the same document) or greater (meaning that we want the very first posting offset for the next document). This is done using the exponential search algorithm, or also known as galloping search.

Exponential search is a two stage searching process on a sorted list. In the first stage, we first define a range in which a search key is likely to be in by using an iterative doubling process. In the beginning we make no assumption about where the search key is located. Instead we set our current index to 2^0 and check whether or not this index is equal or smaller than the search key. If it is not, then we double the current index, 2^{0+1}, and check again. This continues until we have either found a current index, 2^i, that is bigger than the search key or if we reach the end of the list, which would confirm that the key does not actually exist in the list. When we do find such an index, then we know that the search key is definitely in the range between 2^{i-1} and 2^i. In the second stage, we perform a binary search within this range to get the actual search key. Just like *getPostingsSlice()*, this process is also reversible so that range finding stage starts at the end of the list rather than the beginning.

Rather than create a new IndexShard that uses all of these reversed methods by default, we have opted to add these methods as a side option. To enable them, in the constructor of an IndexShard there is now an additional flag called *forward_direction* that specifies which set of methods to use and is set to true by default. If we are simply creating an IndexShard then the value of this flag does not matter, as we still want new items to be added to the end of the shard. However if we are constructing an IndexShard to be read in memory, then we must be explicit on whether we want to read it forwards or backwards. We also do not let the reading direction be changed midway, as it would probably disrupt how the offsets are read in right now.

5.2 Iterating in the reverse

Beyond IndexShards, let us not forget that there is still an IndexArchiveBundle structure responsible for handling a collection of IndexShards. The interface for choosing the correct shard to read with the associated documents also needed to be revamped. However, IndexArchiveBundle actually contains minimal methods that relate to reading and retrieving items from the index, in favor of methods having to do with construction. Instead, Yioop uses iterator classes to handle this final piece of the puzzle. There are several different iterator types, some of which are used to combine the results from other iterators. The original standard iterator used to handle normal indexes is called WordIterator. In order to allow for reverse traversal, we introduce new methods of traversal which are based on existing ones within WordIterator. In this section we will go over how iterators are used and the methods needed for this project.

When constructing an iterator, we have two constants that need to be defined: what term is this iterator for and which IndexArchiveBundle is this iterator working on. If we have multiple terms, such as in the case of conjunctive queries, then we make use of IntersectIterator, which can handle several iterators. In the beginning of an iterator, we make a call to the IndexDictionary to find the shard generations that contain this term. We also set the current generation that we should be reading to the latest shard, seeing as it should contain the freshest documents. From here, we can call *findDocsWithWord()*, a method that is used to read in a block of a shard and return a list of the document ids and score using the *getPostingsSlice()* from IndexShard. If we are currently looking at the oldest generation and the first offset of that generation, then we know that there will be no more documents to check.

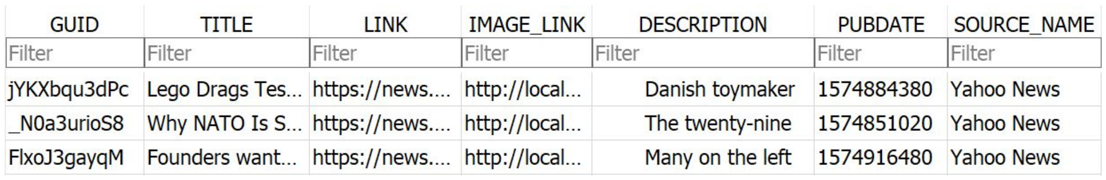
Every iterator also makes use of a method called advance(). Its implementation can be different depending on which iterator it is, but in all cases this method is what allows an iterator to iterate. Upon a call to *advance()*, the iterator attempts to read in a block of an IndexShard using a start offset and last offset. Using a current offset to hold our current position in the shard, we then try to figure out how many documents or links are stored within this block. Since every document or link takes 4 bytes on average to store, calculating this value is trivial. Normally in a forward we will move forward in chunks of up to 800 bytes, as Yioop puts a limit of 200 results maximum that should be returned in a single call. As we move, the current offset is being updated from the start of one block to another. When we have finally reached the end of a shard, that means it is time to jump to the next generation. With the current offset value, we can call *currentGenDocOffsetWithWord()* which in turn calls *docOffsetFromPostingOffset()* from Indexshard. This returns a tuple of the current generation paired with the results from the shard.

Chapter 6

How it comes together

With this, we have finally defined all the necessary parts that are required to get our improved news feed storage working. To illustrate the detailed workflow of these classes and methods, we have constructed a simple example.

Say that during a fresh run of the FeedUpdateJob, we end up finding only three documents. These documents are processed, we produce a unique hash and a short summary for each, and they are subsequently added to the FEED_ITEM table of Yioop's database.



GUID	TITLE	LINK	IMAGE_LINK	DESCRIPTION	PUBDATE	SOURCE_NAME
Filter	Filter	Filter	Filter	Filter	Filter	Filter
jYKXbqu3dPc	Lego Drags Tes...	https://news....	http://local...	Danish toymaker	1574884380	Yahoo News
_N0a3urioS8	Why NATO Is S...	https://news....	http://local...	The twenty-nine	1574851020	Yahoo News
FlxoJ3gayqM	Founders want...	https://news....	http://local...	Many on the left	1574916480	Yahoo News

Figure 5. Feed items being stored in the database before being processed.

Here we can see the seven parameters that make up each record. Because the hash has already been created, we generally assume that a simple deduplication process has occurred. Based on simple observation, we will assume that all three are unique in content. We retrieve each item sorted in descending order on the publication date. For each one of these items, we will create a list of the terms that appear in the URL and the description along with the positions that they appear. All of this data is added to a new IndexShard in memory using its *addDocumentWords()* method. This updates the components inside, and in particular the *words* array now contains the mappings of word_id to postings. The entire array is later converted into a string in the format of

wordid$_1$len$_1$postings$_1$	wordid$_2$len$_2$postings$_2$	…….....	wordid$_k$len$_{\mathrm{k}}$postings$_{\mathrm{k}}$

Figure 6. How an array of postings is stored as a string.

where *k* is the length of the *words* array, which should correspond to all the unique terms we have seen thus far. Before concatenating everything to a string, the array is sorted by lexicographical order, so that the *k-1* item of the string is “smaller” than the k^{th} item. This, admittedly strange, process is done because strings are more memory efficient than associative arrays in PHP.

Once we are done adding documents to the IndexShard, it is time to add the shard to an IndexArchiveBundle. First we check to see if there is an existing IndexShard that we should add to. Then we add the page descriptions of each document to a summaries file also stored in the bundle. When we add the description, we are also keeping track of the offset into the summaries file where we are adding. For our example we should have three summary offsets, which are now added into the *doc_infos* component of IndexShard. We are finally done writing to the shard, and we save it to disk and add it to the IndexArchiveBundle. Now we have a small index to read on.

If we now head to Yioop, we can use our new index to get our news feed. To access our results, we have to make a query to Yioop, even if it is just an empty query with no words. Behind the scenes, Yioop will still add some meta words in our search, and each meta word counts as a separate query term which means we will need to make a WordIterator for each meta word plus our own empty query. In particular, every item in the news feed IndexArchiveBundle has the meta word “media:news” to denote that this was added during a FeedUpdateJob. Each iterator will go through a loop of calling *findDocsWithWord()* and *advance()*, giving us all the pages that contain this term or meta word. This involves a tedious process of looking in the IndexDictionary to check all shards that have postings for this term. For every shard that does, we read it into memory, block by block, to get the actual postings. Since we are using an empty query and, in this example, all the documents have the same basic meta words, the results returned from each iterator will be the same. If we had supplied an actual word to query on, we might not expect this to be true. In order to handle this, we also use the IntersectIterator which handles conjunctive queries. It will go through the results of eachWordIterator, get the first item that appears in all of them, and then add its own results. Now that we have all the correct documents, we will retrieve its summary data by looking it up in the summaries file using the offset stored earlier. This combined pack of data is then sent to the front end of Yioop and displayed on the page to look nice and pretty.

Chapter 7

Testing

The purpose of the project is to extend the scalable storage solution that Yioop uses for main crawls to the news feed function. Additionally, since we want to read the results of this index in decreasing order from latest to oldest we also need to modify the existing index traversal. For this we define two criterias of success: scalability and correctness.

For the first tests, we isolated each component to check for correctness. Starting from the smallest piece of an index, the functions of IndexShard were tested. First off, unit testing was done to check if *getPostingsSlice()* was viable going from a backwards perspective. First I made two IndexShards: one being a normal, forward shard with nothing special done, and the other being a reverse shard with its direction flag set. Then I added the same set of documents to both, which should produce two identical shards excluding the flag. Then it should follow that the results returned from calling the method on the normal shard would be the same as if we had called the same method from the other shard, only in reverse. It was important that the methods of a reverse shard worked correctly, as any mistake would make its way into every other part of the project.

Once I was suitably satisfied with that functionality, I moved onto testing the actual performance of this new system, especially compared to the old one. Recall that previously, Yioop needed to first add all new feed items to a database before then adding all items in the database to a singular IndexShard. In theory we should expect two things: the process of adding items is slower, due to the extra step in the procedure, and we cannot handle as many items since we are capped by one IndexShard worth of data. To test this, we set up a fake local RSS feed running on the same machine as Yioop. The feed is populated with miscellaneous data that is randomly generated, and the amount of items is user specified. During a feed update, Yioop will pull from this feed and try to populate its news section using items from this RSS feed.

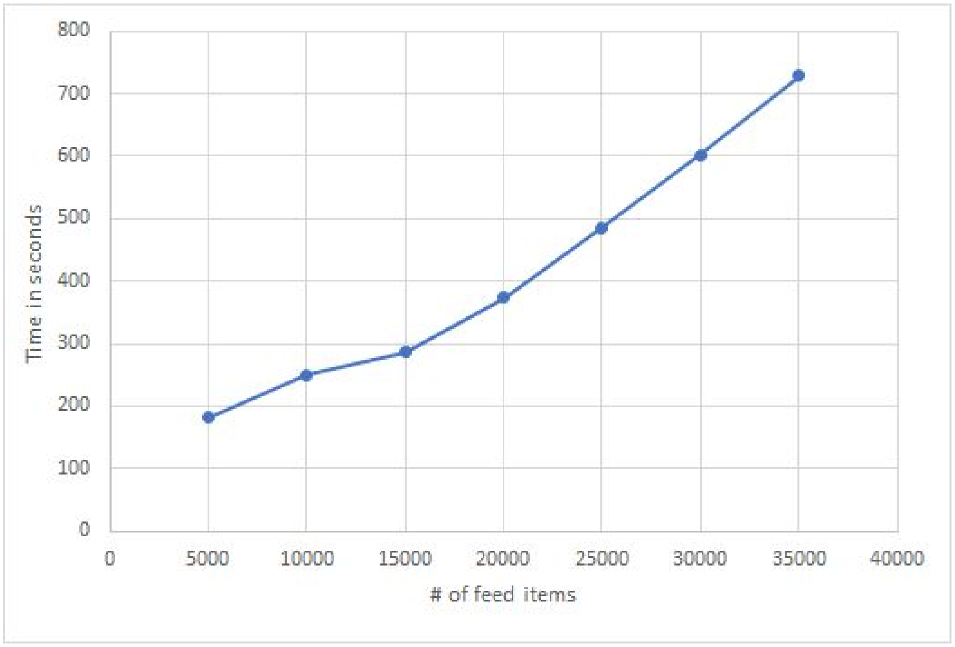


Figure 7. Time needed to add items in old Yioop

In our testing, we found that the old feed system of Yioop was indeed slower than our new method by a great deal. On top of that, our testing revealed that a single IndexShard will hold approximately 37,500 feed items. Since the old system only held one shard, this meant that we were effectively hard capped at 37,500 items for Yioop News at any given time. Because of this, we had to continuously discard old data to make space for more recent items. Trying to add more than 37,500 items in one update will also cause the system to hang indefinitely. In the new system, adding more items will create additional shards to retrieve from, with the intent of never having to throw away anything.

Aside from comparisons to the old Yioop, we also took a look at the scalability of the new feed system when it comes to adding large amounts of data. In order to simulate this, we again set up the fake RSS feed for many items, increasing in increments of 5,000.

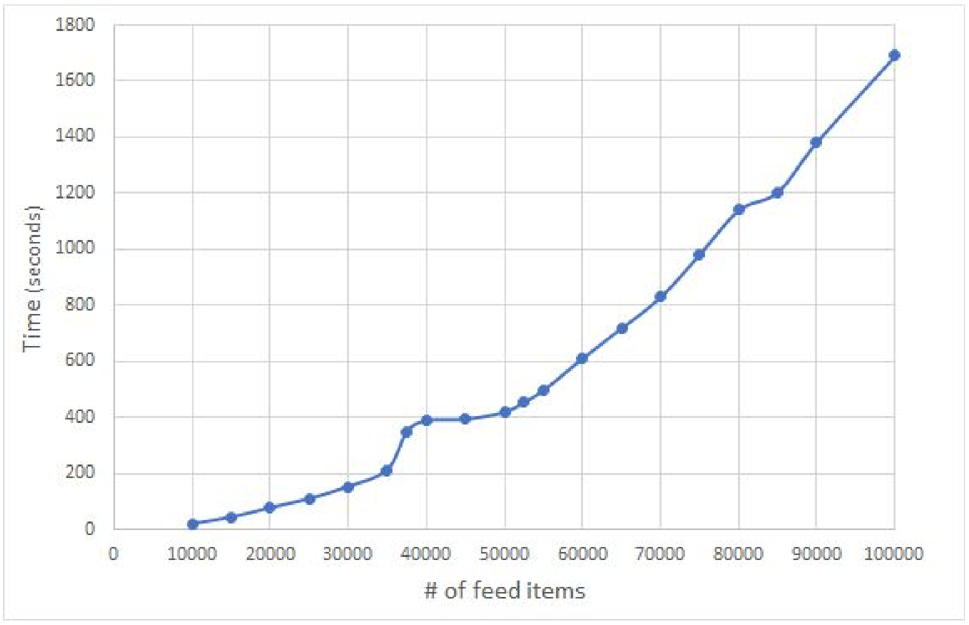


Figure 8. Time needed for adding items in new Yioop

In our findings, we see that adding items to the IndexArchiveBundle is a process that can potentially take a long time. In fact, there almost seems to be an exponential jump in time required to add feed items, especially from 35,000 to 40,000 which could be explained since that is where the first split into a second shard appears. Note that there is a similar bump as we reach the 90,000s, which would approximately be where the second split occurs. However the bump is not as severe as the previous one, relative to how long adding more items already takes. A possible explanation for this phenomenon is that one of the processes that takes more time while adding items is adding new terms to the IndexDictionary. The later an item appears, the less likely it will have a new term to add to the dictionary. Moving bigger shows that the speed dropoff is significant the higher we go with 90,000 items taking upwards of 15 minutes to complete, a far cry from the 19 seconds that adding 10,000 items would take. However, it also seems that the slowdown mostly occurs when adding from only one source. Rather than have Yioop slowly pull from one RSS feed that has 50,000 items, perhaps it could be faster if we had two feeds with 25,000 each, or even four feeds with 12,500.

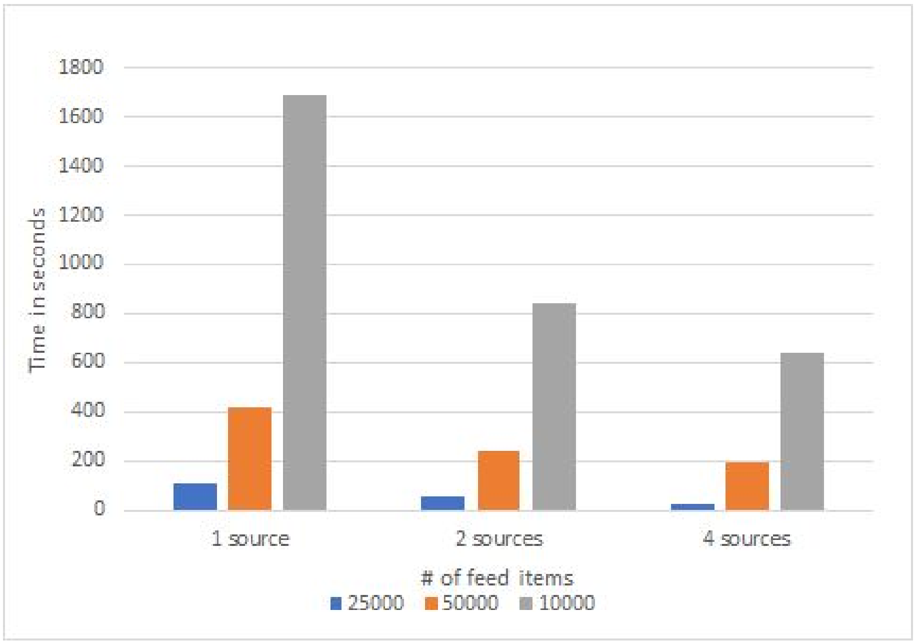
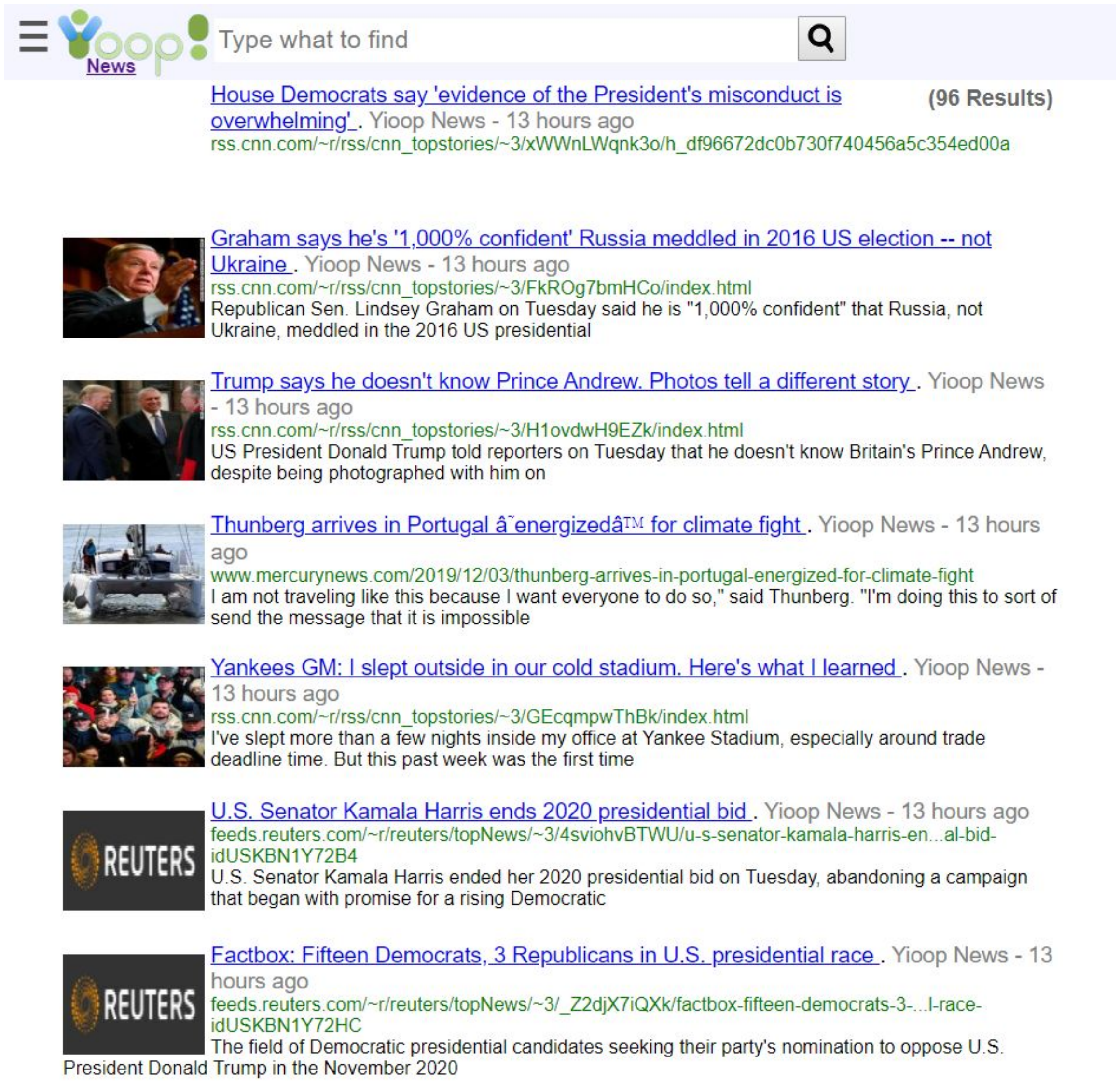


Figure 9. Comparison of having items from multiple sources

Additional testing confirmed that this indeed was the case, with a drastic improvement in speed. From this we conclude that adding many feed items into Yioop during one update cycle can be slow, but this can be mitigated if pulling these items from multiple sources. Of course, none of this accurately reflects real life usage, as most feeds online limit around 30-50 items during retrieval, and garnering enough feeds to reach 10,000 items, let alone 50,000 items, is improbable. Nonetheless, these tests show that even for these given cases, the new storage solution is capable and up to the task.

Apart from testing the retrieval part of the system, we also want to check the index construction for the news feed as well. Testing this was simple: run the updated FeedsUpdateJob and observe if the process properly constructs an IndexArchiveBundle for news feed storage. By using built-in tools from Yioop, I was able to check if the bundle has a functioning dictionary, index shard, and summary storage setup. Following that, I tried iterating over this news feed bundle using the normal iterator and ReverseIterator. They both worked fine over this bundle, leaving one final experiment, which is an end-to-end check of this project. As specified in the previous chapter, I should now be able to use the FeedUpdateJob to create a IndexArchiveBundle, set it as the index for the news feed feature of Yioop, and finally return results back to the user.


Type what to find

House Democrats say 'evidence of the President's misconduct is overwhelming' . Yioop News - 13 hours ago (96 Results)
rss.cnn.com/~r/rss/cnn_topstories/~3/xWWnLWqnk3o/h_df96672dc0b730f740456a5c354ed00a

Graham says he's '1,000% confident' Russia meddled in 2016 US election -- not Ukraine . Yioop News - 13 hours ago
rss.cnn.com/~r/rss/cnn_topstories/~3/FkROg7bmHCo/index.html
Republican Sen. Lindsey Graham on Tuesday said he is "1,000% confident" that Russia, not Ukraine, meddled in the 2016 US presidential

Trump says he doesn't know Prince Andrew. Photos tell a different story . Yioop News - 13 hours ago
rss.cnn.com/~r/rss/cnn_topstories/~3/H1ovdwH9EZk/index.html
US President Donald Trump told reporters on Tuesday that he doesn't know Britain's Prince Andrew, despite being photographed with him on

Thunberg arrives in Portugal â˜energizedâ™ for climate fight . Yioop News - 13 hours ago
www.mercurynews.com/2019/12/03/thunberg-arrives-in-portugal-energized-for-climate-fight
I am not traveling like this because I want everyone to do so," said Thunberg. "I'm doing this to sort of send the message that it is impossible

Yankees GM: I slept outside in our cold stadium. Here's what I learned . Yioop News - 13 hours ago
rss.cnn.com/~r/rss/cnn_topstories/~3/GEcqmpwThBk/index.html
I've slept more than a few nights inside my office at Yankee Stadium, especially around trade deadline time. But this past week was the first time

U.S. Senator Kamala Harris ends 2020 presidential bid . Yioop News - 13 hours ago
feeds.reuters.com/~r/reuters/topNews/~3/4sviohvBTWU/u-s-senator-kamala-harris-en...al-bid-idUSKBN1Y72B4
U.S. Senator Kamala Harris ended her 2020 presidential bid on Tuesday, abandoning a campaign that began with promise for a rising Democratic

Factbox: Fifteen Democrats, 3 Republicans in U.S. presidential race . Yioop News - 13 hours ago
feeds.reuters.com/~r/reuters/topNews/~3/_Z2djX7iQXk/factbox-fifteen-democrats-3-...l-race-idUSKBN1Y72HC
The field of Democratic presidential candidates seeking their party's nomination to oppose U.S. President Donald Trump in the November 2020

Figure 10. The search results page for news feed items.

Chapter 8

Conclusion and Future work

At the beginning of this project, we posed one goal: to improve on Yioop's news feed functionality by improving its storage solution. From the rudimentary tests and experiments that I have performed, I would say this has been largely achieved. To recap, the previous storage solution relied on storing all feed items on one IndexShard which was rebuilt during every update interval using the database. The shard acts more as a temporary index, as items continuously come and go with this setup. In this project, these shards have been moved into a larger data structure, the IndexArchiveBundle, which is built to handle the storage of a discrete amount of shards. The previous solution also assumed that items added to the shard would be in order of new to old, which was why we could use the standard iterator. The new storage does not make this assumption and thus goes through the index backwards to simulate a similar effect.

In terms of future work, I believe there are two areas that could be improved, one related to storage and the other related to retrieval. In regards to storage, IndexShards and IndexArchiveBundles could likely be further optimized. For instance, the index allows for storing many intermediate values that could be used for other functions of Yioop. Not all these values are always needed, and so the space ends up being wasted in this case. Another thing that could possibly be changed is the use of posting_strings to store an entire index. As stated before, PHP uses a rather inefficient array implementation that makes use of hash tables regardless of whether we want it to be associative or not. To overcome this, Yioop makes use of arrays that are serialized as strings, which are then then exclusively accessed using offsets. Without taking great care with these strings and offsets, it is very easy to mess up postings retrieval, not to mention its readability is nearly nonexistent. Seeing as the original IndexShard and IndexArchiveBundle code was concocted nearly a decade ago, it might be worth experimenting with new alternative methods of posting storage.

As for retrieval, the current news feed uses a rudimentary weighting system for showing news feed results back to the user. This weight is influenced by the publication date of each item that is retrieved from the shard, which acts as our freshness. In other news feed systems, such as Facebook, machine learning algorithms are applied to figure out a much more comprehensive freshness rating combined with a relevancy score which takes advantage of additional information from users, the sources they are subscribed to, their interests, and many other little factors. Yioop does none of these tasks and ends up losing out in terms of convenience and personalization. Implementing the same scheme that Facebook uses would be out of our scope, due to lack of time, userbase, and hardware, but it would be possible to add some model that extracts temporal features from a feed item to produce a better ranking based on freshness. Another worthwhile task would be to do something similar to what Google does, in clustering feed items together based on how relevant they may be in terms of topic. This is by no means a comprehensive list of all possible improvements that can be made to Yioop, but it gives some idea as to what directions Yioop can grow towards in the future.

List of References

[1] B. Long, Y. Chang. “Relevance Ranking for Vertical Search Engines”. 2014.

[2] B. Wolf. "RSS and the Open Web". The Old Reader blog. https://blog.theoldreader.com/post/67563942900/rss-and-the-open-web. 2013.

[3] C. Pollett. "Open Source Search Engine Software!" Open Source Search Engine Software. https://www.seekquarry.com/. 2019.

[4] D. Chiu, C. Lee and Y. Pan, "A Classification Approach of News Web Pages from Multi-Media Sources at Chinese Entry Website-Taiwan Yahoo! as an Example," 2009 *Fourth International Conference on Innovative Computing, Information and Control (ICICIC)*, Kaohsiung, 2009, pp. 1156-1159.

[5] G. Singh and S. Sahu, "Review on "Really Simple Syndication (RSS) Technology Tools"," *2015 IEEE International Conference on Computational Intelligence & Communication Technology*, Ghaziabad, 2015, pp. 757-761.

[6] J. Lee, D. Xu, M. T. A. Amin and T. Abdelzaher, "iApollo: A Newsfeed Summary Service on NDN," *2017 14th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, San Diego, CA, 2017, pp. 1-2.

[7] J. Yang, H. Park and J. K. Choi, "A Web-based content syndication platform for IPTV," *The International Conference on Information Networking 2011 (ICOIN2011)*, Barcelona, 2011, pp. 494-497.

[8] L. Giurgiu, G. Barsan and D. Mosteanu, "Web syndication in educational environment," *2008 50th International Symposium ELMAR*, Zadar, 2008, pp. 353-356.

[9] N. Ferro, Y. Kim, M. Sanderson. "Using Collection Shards to Study Retrieval Performance Effect Sizes". ACM, 2019.

[10] P. Kim et al., "Controversy Score Calculation for News Articles," 2019 *First International Conference on Transdisciplinary AI (TransAI)*, Laguna Hills, CA, USA, 2019, pp. 56-63.

[11] R. Monti. "Facebook Discusses How Feed Algorithm Works". SearchEngine Journal. https://www.searchenginejournal.com/facebook-news-feed-algorithm/. 2018

[12] S. Saha, A. Sajjanhar, S. Gao, R. Dew and Y. Zhao, "Delivering Categorized News Items Using RSS Feeds and Web Services," *2010 10th IEEE International Conference on Computer and Information Technology, Bradford*, 2010, pp. 698-702.

[14] Wu Di, Luan Tian, Bai Yan, Wei Liyuan and Li Yanhui, "Study on SEO monitoring system based on keywords & links," *2010 3rd International Conference on Computer Science and Information Technology*, Chengdu, 2010, pp. 450-453.