HIGH PERFORMANCE DOCUMENT STORE IMPLEMENTATION IN RUST

A Project Report

Presented to

Dr. Chris Pollett

Department of Computer Science

San Jose State University

In Partial Fulfillment

Of the Requirements for the Class

CS298

By

Ishaan Aggarwal

Dec 2021

The Designated Project Committee Approves the Master's Project Titled

High Performance Document Store Implementation in Rust

By

Ishaan Aggarwal

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

DECEMBER 2021

Dr. Christopher Pollett	Department of Computer Science
Dr. Robert Chun	Department of Computer Science
Mr. Akshay Kajale	Amazon

ACKNOWLEDGEMENT

I would like to thank Dr. Chris Pollett for the guidance, motivation, and support that he provided over the course of this project. I got to learn a lot while working with him and consider myself fortunate for the same. His knowledge and approach to problem solving helped me learn how to troubleshoot and move forward in situations where no solution seems to work. He suggested me the topic for this project and I was curious because all the things were new to me, including the implementation language. He has been patient throughout and always provided the moral as well as technical support that I needed to continue the project.

I express my gratitude to the members of my defense committee, Dr. Robert Chun and Mr. Akshay Kajale, and to the CS faculty for providing support over the two years of my degree in one form or the other. Finally, I would like to thank my friends, peers, and family for the support and motivation.

ABSTRACT

Databases are a core part of any application which requires persistence of data. The performance of applications involving the use of database systems is directly proportional to how fast their database read-write operations are. The aim of this project was to build a high-performance document store which can support variety of applications which require data storage and retrieval of some kind. This document store can be used as an independently running backend service which can be utilized by search engines, applications which deal with keeping records, etc. We used Rust to make this document store which is fast, robust, and memory efficient. The document store is a server which can return documents based on the key provided in the request. It has the capability to read WebArchive (warc extension) files as a feed to the linear hashing based datastore. The paper focuses on the implementation of this application and how it is a relevant backend system for a search engine.

We performed various tests for the functionalities that are offered by the application and documented the noteworthy results. The linear hash-table has the capability to insert 10,000 records with key-value pairs sized 16 bytes in 10 seconds where a similar implementation in JavaScript takes around 40 seconds for the same. The insertion time in our implementation increases logarithmically. The hash-table supports retrieval of 10,000 similar sized records in under 5 seconds. The WebArchive parser utility supports the parsing of 10,000 records of a compressed (gzip extension) warc file in an average of 70 seconds. This is approximately the same as the warcio library in Python. This time increases linearly with the number of records that are read, in our application. Along with the conversion of warc records into a format suitable for the linear hash-table, it takes the application an average of 80 seconds. The warc utility also has the ability to write warc files. It can write 10,000 warc files with a body of size around 60bytes in an average of 2 seconds. Detailed performance comparisons with other similar tools are also documented in the paper.

Index Terms: **High Performance Document Store, Linear Hashing, Hash-Table, Buckets, WebArchive, Rust, JavaScript, Python, Data Storage, Data Retrieval**

LIST OF FIGURES

TABLE OF CONTENTS

I. INTRODUCTION

The volume of data being produced and consumed daily has increased quite aggressively in recent times. According to a recent survey by Visual Capitalist [1], 4 million Google searches are performed, 7 hundred thousand hours-worth of videos are consumed on Netflix, Snapchat creates around 2 million snaps and users upload over 4.5 million videos on Youtube, in just one internet minute today. Apart from these, there are many other essential services which require error-free storage and processing of data like banking, social media, government records, etc. The above examples have one thing in common, proper storage and retrieval of data, and this is what databases are required for. In this project, we built a high-performance database system.

Search engines are such applications which depend heavily on their database systems. Due to the importance of databases in search engines, their performance becomes solely dependent on how fast their database systems are. They find, store, and update the content of web pages in their databases, called web crawling. When a user searches for a particular content, search engines match that with content in web pages that they have stored and return relevant results. To allow the finding of relevant results, the document store needs to have a provision for creating index based on various parameters. Datastore needs to ingest the data in a compact format, allowing fast retrieval. Also, the focus will be to allow higher read speeds while we can take more time for doing writes. According to worldwidewebsize.com, there are at least 20 billion pages on the internet. To deal with this much amount of data, search engines require not only sophisticated algorithms, but databases which can support high read-write speeds. Hence, we will be building a custom database which can fulfil these requirements.

A search engine needs to search inconsistent web pages and show the search results as quickly as possible. This requires a high-performance document store. Document store is a type of database which allows schema-free organization of data. This kind of database is most suitable for a search engine because web pages do not follow any consistent format (schema) or size. Currently, there are various document-based data stores available like MongoDB, DynamoDB, etc. Some of these are open-source as well and provide lots of functionalities.

These databases are built for general use purposes where reads and writes are equally important. This project aims at building a high-performance document store which can serve as a reliable, performant and memory efficient data related backend for any application involving data storage and retrieval. At the start of the project, the focus was on implementing the individual modules which will be required in achieving our aim and later they were integrated to make the functionalities work together.

The main reason why we built this system is the lack of open source data stores which focus more on higher read speeds. Our datastore will focus on making reads fast while also caring about writes. We chose Rust for this implementation. It allows the developer to decide to either store the data on the stack or on the heap. It also determines when memory is no longer needed and allows for fast clean up at compile time. This feature allows for efficient usage of memory as well as relatively faster memory access [2].

The remaining document is organized in six chapters to describe the project and its timeline in a more intuitive way. Chapter II introduces the concept to document stores, their history of development and purpose. It also covers the individual modules that we implemented to make the document store work, including a single-node document query server, linear hash table datastore, and consistent hashing for further extension. Chapter III elaborates on web archive (warc) files. It sheds light on a brief history of warc files and covers the modules implemented for reading and writing of warc files as well as gzip files. Chapter IV explains the architecture of the whole project and the work flow which makes clear the use of the project. Chapter V covers the experiments that we did to test our application. Chapter 6 and 7 elaborate on the challenges faced while implementing the code and what future work is required to make this application production ready and extensible.

II. DOCUMENT STORES AND SEARCH ENGINES

For the purpose of understanding the motive behind this project, it is necessary to understand the concept of document stores, which are a type of NoSQL databases. It will also help to understand about how search engines work, their requirements for data persistence and retrieval, and characteristic functionalities. This chapter will cover, in brief, the evolution of NoSQL databases, document stores, and features of search engines.

Document stores are data storage systems that facilitate storing, retrieving, and managing document type records. This kind of databases are an important part of NoSQL database systems because the documents are semi-structured data. Document stores are a subclass of key-value stores which form another important part of NoSQL database. The difference between the two lies in how the retrieval of data happens. In key-value data stores, value is stored directly against a key and just reaching that key gives you the value. In document stores, additional steps are required to retrieve the value based on the structure in which data is there in the document. A document-oriented database offers the ability to query or update based on the underlying structure of the document through APIs or a query/update language. This difference may be insignificant for customers who do not require document databases' richer query, retrieval, or editing APIs. Modern key-value stores frequently integrate metadata-related features, blurring the distinctions between document and key-value stores.

2.1 Evolution of NoSQL and Document stores

The term NoSQL origitanted in 1998 and was given by Carlo Strozzi, to refer to his open-source, lightweight "relational" database that didn't require a structured query language. In 2009, Eric Evans and Johan Oskarsson used the same term to label non-relational databases. SQL systems are commonly used to refer to relational databases. The word NoSQL can refer to either "No SQL systems" or "Not exclusively SQL," to underline that certain systems may offer structured query languages similar to SQL.

NoSQL emerged as a response to web data, the need to process unstructured data, and the requirement for speedier processing, at least in the beginning. A distributed database system, or a system with numerous computers, is used in the NoSQL model. The non-relational system is faster, takes an ad-hoc approach to data organization, and can handle enormous amounts of data of various types. Due to their flexibility and speed in that condition, NoSQL databases are a better option to choose than SQL databases for huge, unstructured data sets in general.

NoSQL systems are capable of handling both structured and unstructured data, as well as processing unstructured Big Data fast. As a result, companies like Facebook, Twitter, LinkedIn, and Google have adopted NoSQL systems. Though it is true that if we can structure the data, SQL databases give better performance. These companies collect and generate massive amounts of unstructured data in order to uncover patterns and generate business insights. In 2005, the term "big data" was coined.

As per today’s requirements, many of the system need to focus on providing high availability to the users. Providing consistency with availability is a cumbersome process where the limitations were enlisted by Eric Brewer in the form of CAP theorem. The CAP Theorem, sometimes known as Brewer's Theorem is a fundamental concept in non-relational databases and distributed systems. It specifies that a distributed data store "cannot" provide more than "two of three" defined guarantees at the same time. Brewer introduced the theory in the fall of 1998 at the University of California, and it was published in 1999 as the CAP Principle. The following are the three guarantees that cannot be met at the same time: Consistency - even after an operation is completed, the data in the database remains consistent. Availability – the system is always on (always available), and there are no outages. Partition Tolerance - the system will continue to work even if communication between the servers is no longer dependable. This is due to the fact that the servers might be partitioned into distinct groups that are unable to connect with one another.

Open source non-relational data storage is non-relational, schema-less, horizontally scalable, and employs BASE for consistency. Data storage that is scalable, schema-free, and enables for rapid updates and replication is referred to as "elasticity." In general, these characteristics have been achieved by developing NoSQL data storage from the ground up

and optimizing it for horizontal scaling. These systems frequently only support low-level, simple APIs (such "get" and "place" operations). As a result, modelling with non-relational systems feels very different from modelling with relational systems, and it follows a different philosophy.

2.2 Search engines

Although the little text box on a web page is the most obvious aspect of a search engine, there are several behind-the-scenes features that make the text box work. Let's look at how a search engine differs from traditional database queries and why you should think about search when choosing a NoSQL database.

2.2.1 Search vs Query

A query is not the same as a search. A query obtains data based on whether it matches the query exactly. Good instances are looking for orders that include a specific item or goods in a specific price range. In contrast, a search is imprecise and does not necessitate tight adherence to a shared data model. With potentially sophisticated Boolean logic, terms may be essential or optional. Rather than just reporting "match" or "not a match" in a query, a relevance score is usually calculated. These relevancy computations are adjustable, and they vastly improve the search experience of the average user. Search engines often give you recommendations on how to narrow down your search. You may be supplied with information about the entire result set, which are termed facets, in addition to a page of results and relevancy scores for each result. Facets can be used to filter down a product search by product category, price range, or date of addition, for example. Facets improve users' experiences by allowing them to discover new methods to improve their original search criteria without needing to know anything about data structures.

2.2.2 Web Crawling

Web crawlers are automatic programs that monitor recognized websites for changes and follow links to new websites and pages, indexing whatever content they come across. They may also make a duplicate of the current state of a web page when indexing it. The first order of business when experimenting with search is to find some material! Content on e-

commerce websites, for example, can be highly structured relational database records. It could be completely free text, or something in between, such as a Microsoft Word document or a structured web page.

Crawling the authoritative source system is the first step. Some databases include built-in indexing, allowing for real-time indexing. However, the vast majority of search engines index remote information and update their indexes on a regular basis. These indexes may be out of date, but because a lot of stuff remains the same, you don't need to discover it right away.

A collection of URLs is the starting point for web search engines (Uniform Resource Locators). They read the text on the web pages and index it, as well as adding newly discovered links to the queue of indexed sites, when crawling these URLs. This is the only form of crawler that can find all linked web sites on the internet. It's possible that your company will need to crawl a number of systems. If that's the case, keep in mind that several search engines, like IBM OmniFind, HP Autonomy, and Microsoft FAST, provide interfaces to a number of sources. Crawlers are usually given by independent search engines. Some search engines, such as MarkLogic's NoSQL document database, are integrated within databases. Because these embedded search engines scan their own content and don't have crawlers for other systems, you'll have to move or copy content into the NoSQL database to get it indexed.

The benefits of a real-time index for material stored in a NoSQL database may exceed the disadvantages of a separate search engine, despite the additional storage requirements. The main problem of using a different search engine is that crawling happens on a regular basis, resulting in inconsistent data results and false positives (documents that no longer exist or don't fit the search query).

2.2.3 Indexing

After you've found some content, you'll need to determine what to index. You only need to save the title of the document, the link, and the date the content was last updated. You can also extract the text from the content and create a list of the words mentioned in the document. You can even save a copy of the page as it was when it was indexed. This is a

common feature in Google search results, especially if the author of a page has recently deleted it and you still want to access it as it was when it was indexed.

A standard index keeps track of a document's ID and a list of the terms that appear in it. This strategy, however, isn't especially beneficial for running a query. It's much preferable to have document IDs in a list next to each distinct term. A search engine can quickly find the set of documents that match a query by doing so. This is referred to as an inverted index, and it is essential for optimum search performance. The more content you index, the more critical it is to store inverted indexes to save query times. Inquire about the index structure of your vendor's product as well as the size at which it operates successfully. Of course, there's a chance you'll want to index more than just words. Date fields (made, updated, indexed at), numbers (page count, version), phrases, and even geographical coordinates of places mentioned in content can all be indexed. These are referred to as terms, and the lists of documents that match them are referred to as term lists.

If the search engine must match all words, dates, and numeric terms in a query, the search engine will execute an intersection of the matching term lists, which means the search engine will only return results for documents that include all three term lists. AND Boolean logic is the name for this type of logic. OR Boolean logic, which means that documents with more matching terms are given a higher relevancy score, is perhaps more useful. The method for determining relevancy differs based on the context. For example, you might believe that findings from recent news stories are more relevant than those from older news stories. Furthermore, you may want to give more weight to those instances where the terms you entered are matched more frequently in the same text. However, a document of 20 words (say, a tweet) that mentions your word once may be more relevant than a paper of 20 pages that also includes the same word once. A document with a frequency of one word in every twenty has a high frequency.

2.2.4 Searching

The key to making searching simple is to employ a text format that is easy to grasp, which is referred to as a search grammar. Most search engines use a Google-like free-text syntax. Enhanced search services, which allow users to query over hundreds of phrases in a single

search, are frequently provided by specialist publications. This feature is especially beneficial for specialized datasets like financial services news items and filings data. Many search engines give a default grammar, with some allowing you to alter the grammars and how they're processed, which is beneficial if you're considering switching to a more scalable search engine but don't want to burden your application's users with learning a new search grammar. There are further concepts like pagination, sorting, faceting, snipping, dictionaries, etc. which are used to make searches faster and this discussion is beyond the scope of the project.

2.3 Document Stores and Search Engines

Many modern search engines have a similar architecture to NoSQL databases. Their indexes and query processing are dispersed over multiple servers. Many search engines can also serve as a key-value or document store in their own right. NoSQL databases are frequently used to store unstructured data, documents, or data that can be saved in a variety of structures, such as social network posts or web pages, as we saw above. This indexed data has a wide range of structures.

III. WEBARCHIVE FILES

Just like we came to know about what we are building and for what kind of applications are we building our data store system, in this chapter, we will understand the input data that needs to be fed to our data store so that it can return the response to the queries. This chapter will elaborate on web archive files which is in short known as warc, a common format in which most of the web crawling happens. We will also understand the function of CDX files which are basically an index on the warc files for faster reads from the huge chunk of data that warc files are.

3.1 What is a WARC file?

A WARC (Web ARChive) is a container file standard for storing web content in its original context, maintained by the International Internet Preservation Consortium (IIPC). It is a container file that is used for multiple purposes. It is a digital file that you can store on your own local or networked storage, like a PDF document or an MP3 audio file, complete its own .warc file extension and application/warc mimetype. It is a container file that houses other files. It concatenates several files into one digital object, like you've seen elsewhere from container formats like ZIP, GZIP, TAR, or RAR. A WARC wraps around other files like the PDF and MP3 above, along with some additional information and formatting that we'll cover below. It is a container for files that are native to the web.

WARCs are produced by crawlers, proxies, and other utilities that retrieve files from a live web server. They can contain the PDF and MP3 files described above, for instance, but also the HTML, JS, CSS, and other structural elements that web browsers need to read in order to represent site contents to human computer users. It can also contextualize those contents. WARCs contain technical and provenance metadata about the collection and arrangement of their media so sites can be read and represented in live web browsing experiences like they were at the time of their collection.

WARC is a standard container format. The WARC file format standard was published by the International Organization for Standardization (ISO) committee on technical interoperability as ISO 28500. You might get other outputs from web scraping tools, but WARC is the generally agreed-upon way to contain web archives such that people and their software know how to interpret and read the contents today and into the future. It is a standard maintained by web archivists. Keeping up the WARC file format standard is the responsibility of the International Internet Preservation Consortium (IIPC). This coalition of practitioners does the 'agreeing upon' above, that keeps the WARC relevant and vital to how we collect and preserve web archives.

3.2 Brief History of WARC files

The WARC was preceded by the ARC file format, which the Internet Archive used to contain its collected web archives as far back as 1996. The ARC file was the Internet Archive's original container file for web-native resources, so it conformed to the first three bullet points in the definition above. Reflecting the needs of web archivists around the world to preserve more context about their collected resources, the WARC standard was formalized in 2009 to include the very detailed kinds of technical metadata. Much specificity and readability were added to the WARC standard for its 2017 upgrade to version 1.1.

3.3 CDX files

As mentioned earlier, CDX files are used to index records in WARC files. A CDX file contains a header line specifying the format of all subsequent lines. Figure 1 shows

```
A canonized url
B news group
C rulespace category ***
D compressed dat file offset
F canonized frame
G multi-column language description (* soon)
H canonized host
I canonized image
J canonized jump point
K Some weird FBIS what's changed kinda thing
L canonized link
M meta tags (AIF) *
N massaged url
P canonized path
Q language string
R canonized redirect
S compressed record size
U uniqueness ***
V compressed arc file offset *
X canonized url in other href tags
Y canonized url in other src tags
Z canonized url found in script
a original url **
b date **
c old style checksum *
d uncompressed dat file offset
e IP **
f frame *
g file name
h original host
i image *
j original jump point
k new style checksum *
l link *
m mime type of original document *
n arc document length *
o port
p original path
r redirect *
s response code *
t title *
v uncompressed arc file offset *
x url in other href tages *
y url in other src tags *
z url found in script *
# comment
```

Figure 1. Meanings of letters used in CDX header

All subsequent lines universally contain the URL of a WARC record, information about the record, and offset plus length of the record in a WARC file. Thus, you can read a CDX index

line and then read the subsequent offset specified to retrieve the WARC record in a WARC file quickly. CDX files are simple to parse as you just need to read the first line in the CDX file then parse generate the format structure based on the official CDX file specifications. Then read line by line retrieving and parsing the line according to that format structure. In the CDX structure there should be a WARC file name and offset where we can use to quickly retrieve the CDX record associated with this index.

```
CDX A b e a m s c k r V v D d g M n
0-0-0checkmate.com/Bugs/Bug_Investigators.html 20010424210551 209.52.183.152 0-0-0checkmate.com:80/
Bugs/Bug_Investigators.html text/html 200 58670fbe7432c5bed6f3dcd7ea32b221 a725a64ad6bb7112c55ed26c
9e4cef63 - 17130110 59129865 1927657 6501523 DE_crawl6.20010424210458 - 5750
0-0-0checkmate.com/Bugs/Insect_Habitats.html 20010424210312 209.52.183.152 0-0-0checkmate.com:80/Bu
gs/Insect_Habitats.html text/html 200 d520038e97d7538855715ddcba613d41 30025030eeb72e9345cc2ddf8b5f
f218 - 47392928 145482381 4426829 15345336 DE_crawl3.20010424210104 - 6356
0-0-0checkmate.com/Hot/index.html 20010424212403 209.52.183.152 0-0-0checkmate.com:80/Hot/index.htm
l text/html 200 52242643710547ff4ce2605ed03ed9e2 b06d037c06e7ffd7afc6db270aca7645 - 21301376 623055
47 1855363 6627262 DE_crawl6.20010424212307 - 6317
```

Figure 2. Example of a CDX file

IV. ARCHITECTURE AND WORK FLOW

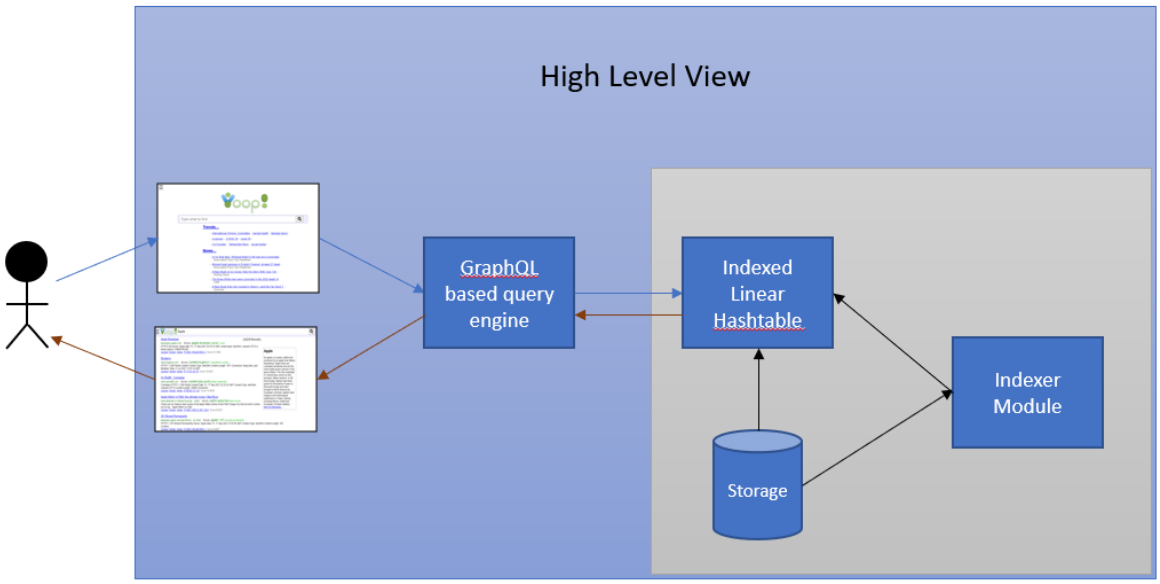

Figure 3. System architecture

The figure shows the architecture of the application that we aimed to build. A user will visit Yioop search engine and write keywords to be searched in the text box. That query will be served by the GraphQL based query engine and sent to the document store module of the application. There, the keyword-based search will happen in the linear hash table which is already indexed. The document store module also has a separate functionality where, from the storage of gzipped WARC files on the disk, indexing happens and the location of records is fed into the linear hash-table. This functionality is independent of the query part of the application. This module will run when new crawl data is available and will be run by a scheduler job. In below sub-sections, we will cover the individual components of the modules to get a better understanding of the functionalities and later, how different modules are working together.

4.1 Linear Hashing

The reason we chose Linear Hashing for our hash-table implementation is that linear hashing is a dynamic hashing technique that grows the number of initial buckets one at a time according to some criteria. Hence, the name Linear Hashing. Our application employs the implementation of dynamic hashing scheme known as Linear Hashing.

Elaborating more on how linear hashing works, a typical hash function's output will always give a fixed number of bits. Let us assume a hash function gives a 32-bit hash output from some key. In Linear Hashing however, we will only use the first I bits to address to N initial buckets. If we start with N =2 bucket, then I = 1 bit. So, we will only use the first bit of the hash function's 32-bit output to map to a bucket. Let our criteria for adding a bucket be passing a load factor threshold that is,

Load factor = number of items / (number of buckets * average items in a bucket)

Once the number of insertions exceeds the threshold, we add a bucket to N. If N becomes another power of 2: N > (2^I -1) we increment I to address the new buckets. When any bucket is added we split the bucket at an index S. S is initially the first bucket. When we split a bucket, we rehash all the keys at bucket S add if the keys rehash to the address of the newly added bucket, we move the key there. Once N buckets has doubled from its initial position, we reset the S index to 0. This logic lets the linear hash-table to occupy minimum extra size while providing the ability to grow as the input grows. Also, the hashing mechanism leads to fast reads where the worst-case time complexity becomes a function of a constant.

4.1.1 Implementation Overview

In our implementation, we first hash the key, and take however many bits the hash-table is currently configured to take. This tells us which bucket to place the record in. A bucket is a linked list of pages, which are chunks of bytes on disk. Pages in the linked list may be full, so we now need to figure out which page the record should go in. Once we figure out which page it should go in, this page is fetched from disk. We then make the necessary changes to the page in memory – e.g., write record, increment number of records in page's header) – then save it out to disk. Getting the value is very similar and uses the same method that we use to figure out which page in the bucket the record should be placed in.

4.1.2 Pages and Buffer Pool

A file is divided into pages of size 4KB. When we want to read or write some chunk of bytes in a page, we have to read the page into memory, make whatever changes we want to make on the copy resident in memory, then save the updated page out to disk.

Now, we will usually have a little more than 4KB of memory at our disposal, so what we can do is buffer the pages we read and write. So, instead of flushing the page out to disk once we do one operation, we keep the page in memory as long as possible.

```
pub struct Page {
    pub id: usize,
    pub storage: [u8; PAGE_SIZE],
    pub num_records: usize,
    // page_id of overflow bucket
    pub next: Option<usize>,
    pub dirty: bool,

    keysize: usize,
    valsize: usize,
}
```

Figure 4. Page struct in our implementation

Notice that besides the bytearray (storage) we have some other metadata about the page as well: 'id' specifies which page in the file this page is, 'storage' is all the bytes in the page copied out to a byte array, 'num_records' specifies how many records this page has, 'next' is what strings the overflow pages we talked about in the last post. It is also used to keep track of pages that used to be overflow pages but are not in use, 'dirty' specifies whether the page here is out of sync with its corresponding page on file. The next two fields 'keysize' and 'valsize' specify what length the key and value bytearrays are in the records the page stores.

The metadata is stored in the page itself. To read and write this metadata we have the methods shown in figure.

```
pub fn read_header(&mut self) {
    let num_records : usize = bytearray_to_usize( b: self.storage[0..8].to_vec());
    let next : usize = bytearray_to_usize( b: self.storage[8..16].to_vec());
    self.num_records = num_records;
    self.next = if next != 0 {
        Some(next)
    } else {
        None
    };
}

pub fn write_header(&mut self) {
    mem_move( dest: &mut self.storage[0..8],  src: &usize_to_bytearray( n: self.num_records));
    mem_move( dest: &mut self.storage[8..16],  src: &usize_to_bytearray( n: self.next.unwrap_or( default: 0)));
}
```

Figure 5. Methods to read and write metadata

The main content of the page are records that are stored in it. These are read and written using the methods in figure below.

```
pub fn read_record(&mut self, row_num: usize) -> (&[u8], &[u8]) {
    let offsets : RowOffsets = self.compute_offsets(row_num);
    let key : &[u8] = &self.storage[offsets.key_offset..offsets.val_offset];
    let val : &[u8] = &self.storage[offsets.val_offset..offsets.row_end];
    (key, val)
}

/// Write record to offset specified by `row_num`. The offset is
/// calculated to accomodate header as well.
pub fn write_record(&mut self, row_num: usize, key: &[u8], val: &[u8]) {
    let offsets : RowOffsets = self.compute_offsets(row_num);
    mem_move( dest: &mut self.storage[offsets.key_offset..offsets.val_offset],
              src: key);
    mem_move( dest: &mut self.storage[offsets.val_offset..offsets.row_end],
              src: val);
}
```

Figure 6. Methods to read and write records

4.1.3 Buckets

For writing the records, we need to be able to write to buckets without having to know in advance where and in which page the record will go to. The code keeps fetching the next page in the bucket until it finds the record with the key it's looking for. What is perhaps interesting is what it returns when it doesn't find the record (because the record hasn't been inserted)– if there's space, it indicates in SearchResult which page and row to insert the record in; if there isn't any space, it returns what the last page it looked in was, which is meant to be used when creating an overflow page. Below images show the SearchResult struct and the method which searches the bucket. After we find out where the record should be placed in a bucket, it becomes easy to put the record in our hash-table. Which bucket should the record go in depends on which how many bits we're looking at and how many buckets we have.

```
pub struct SearchResult {
    pub page_id: Option<usize>,
    pub row_num: Option<usize>,
    pub val: Option<Vec<u8>>
}
```

Figure 7. SearchResult struct

```rust
pub fn search_bucket(&mut self, bucket_id: usize, key: &[u8]) -> SearchResult {
    let mut page_id : usize  = self.bucket_to_page(bucket_id);
    let mut buffer_index : usize ;
    let mut first_free_row = SearchResult {
        page_id: None,
        row_num: None,
        val: None,
    };
    loop {
        buffer_index = self.fetch_page(page_id);
        let next_page : Option<usize>  = self.buffers[buffer_index].next;
        let page_records : Vec<(...)>  = self.all_records_in_page(page_id);

        let len : usize  = page_records.len();
        for (row_num : usize , (k : Vec<u8> ,v : Vec<u8> )) in page_records.into_iter().enumerate() {
            if slices_eq( s1: &k,  s2: key) {
                return SearchResult{
                    page_id: Some(page_id),
                    row_num: Some(row_num),
                    val: Some(v)
                }
            }
        }

        let row_num : Option<usize>  = if len < self.records_per_page {
            Some(len)
        } else {
            None
        };

        match (first_free_row.page_id, first_free_row.row_num) {
            // this is the first free space for a row found, so
            // keep track of it.
            (Some(_), None) |
            (None, _) => {
                first_free_row = SearchResult {
                    page_id: Some(page_id),
                    row_num: row_num,
                    val: None,
                }
            },
            _ => (),
        }

        if let Some(p : usize ) = next_page {
            page_id = p;
        } else {
            break;
        }
    }

    first_free_row
}
```

Figure 8. Method to search bucket

4.1.4 Performance and Limitations

We implemented the code in the form of a library which can be used directly. The tests included inserting one 10,000 records into the key-value store and then trying to retrieve a thousand values. The insertion of these many records of key and value sizes of 16 bytes took around 10 seconds on an average and the retrieval of the values took around 4 seconds on average for multiple runs. There is a scope of improvement in this based on future requirements. One of them is that the key-value store should be able to allow flexible value sizes. Current implementation supports only fixed size of the value. Another improvement would be to implement Least Recently Used (LRU) cache mechanism instead of FIFO to be able to return the values more efficiently. Another feature that would be useful is to be able to delete the records based on the key. If it seems that there is a requirement of these improvements, we will be implementing them.

4.2 WARC Reader/Writer

As explained above, the WARC files are the files which are aggregation of multiple web pages in a compressed format. These files are a file format tailor made to for archiving resources from webpages. They files have been used for historical storing of the web-crawl data as sequence of blocks, collected by the web crawlers. Each WARC file is a concatenation one many WARC records. Along with their index files (.cdx), it becomes easier to jump to the offset in memory which stores a relevant information, without needing to decompress the whole files. The Yioop! search engine stores its crawl data in WARC format, which makes this deliverable a useful tool to read-from and write-to the WARC files.

```
WARC/1.0
WARC-Type: response
WARC-Target-URI: http://shop.kaze-online.de/images/products/small/PB0281.jpg
WARC-Date: 2011-02-25T18:32:18Z
WARC-Payload-Digest: sha1:QG6N6SOXUHFK2BGT6EEGMNQMALUW5YAE
WARC-IP-Address: 87.119.197.90
WARC-Record-ID: <urn:uuid:bf19c5b5-3756-4885-bc8b-78b76669c987>
Content-Type: application/http; msgtype=response
Content-Length: 3287

HTTP/1.1 200 OK
Date: Fri, 25 Feb 2011 18:32:18 GMT
Server: Apache/2.2.9 (Debian) PHP/5.2.6-1+lenny9 with Suhosin-Patch mod_ssl/2.2.9 OpenSSL/0.9.8g
Last-Modified: Wed, 12 May 2010 17:10:21 GMT
ETag: "56eb28-b94-48668b78c7940"
Accept-Ranges: bytes
Content-Length: 2964
Connection: close
Content-Type: image/jpeg
```

Figure 9. Example of a WARC file's header

A WARC record can be broken down in two distinct parts a WARC header and the content block. With the WARC header containing some information about the block. WARC files are usually very large and so are gzipped. Hence, this deliverable also involved reading of gzip files. WARC record headers all start with this line 'WARC/1.0' or 'WARC/1.1'.

In our implementation, we utilized a Rust crate called libflate[4]. It is a Rust implementation of DEFLATE algorithm and related formats (ZLIB, GZIP). After decompressing the WARC file, we are currently storing the result in memory. This is considering the use case of a user querying some data and we need to return the webpages. This number is something which can be kept in memory and need not be stored in a separate file. We tested our code with the gzipped WARC files downloaded from archive.org and commoncrawl.org. Each file was of sizes around one gigabyte. Each WARC file was of five gigabytes. The WARC file parser was able to read the whole file in twelve minutes. Though in actual requirement, there are remote chances that we need to read the whole WARC file, we will be using CDX files to make reading more efficient.

4.3 Indexing

For the indexing purposes, we have PackedTableTools.rs which packs as a string an array of records. Here each record should be an associative array of field items, with field names and types according to this packed table tools signature. The format of the packed records string is: output of bool columns as bit string in order of columns as appear in signature, bit data on sizes to use for each int column (for each column two bit code 00 - 1byte, 01 - 2byte,

10 - 4byte, 11 - 8byte), text column length (1byte/column saying how long the data stored in that column is), This is followed by the actual column data (except bool columns) in the order it is listed in the signature. Int's use their high order bit as a sign bit and are stored using the number of bytes given by their code in the int column bit data. Real/doubles are stored as 8byte doubles. Given the format as an argument, this file also contains method to unpack a record and return it in its entirety.

4.4 GraphQL server

GraphQL is both an API query language and a runtime for executing those queries using your current data. GraphQL gives you a detailed and easy-to-understand description of the data in your API. It also lets you design your own schema and use it on the server. After a GraphQL service is up and running (usually at a URL on a web service), it can accept and execute GraphQL queries. The service analyzes a query to make sure it only refers to the types and fields defined, then executes the provided functions to generate a result. For us, the purpose of this server is to handle HTTP requests, incoming from Yioop, take the GraphQL query parameters provided and use them to retrieve the required data from the linear hash-table.

V. EXPERIMENTS

We ran tests on various components of the application to compare the performance with similar implementations in other languages. The one component which decides the performance of our application is the linear hash-table. Faster the data input and retrieval in the hash-table, faster our queries could be served. Below tables show the average performance data of 10 runs for our linear hash-table implementation with various number of inserts and using different key-value sizes.

We also tested our implementation with similar implementations in JavaScript. Below table shows the comparison of insertion and retrieval times for different numbers, from the hash-table for similar sized key-value pairs.

Number of Insertions	**Time (seconds) – 4 bytes**	**Time (seconds) – 8 bytes**	**Time (seconds) – 16 bytes**	**Time (seconds) – 32 bytes**
10,000	19.5812	13.7196	10.7140	11.0392
20,000	40.5785	29.5620	29.2961	29.1158
50,000	135.2164	115.9305	110.8547	111.2843
100,000	294.1901	286.4809	290.2886	295.3691

Table 1. Time(seconds) taken to insert number of key-values of different sizes

Number of Retrievals	**Time (seconds) – 4 bytes**	**Time (seconds) – 8 bytes**	**Time (seconds) – 16 bytes**	**Time (seconds) – 32 bytes**
10,000	8.4741	6.0003	4.0419	4.2534
20,000	22.6415	18.2303	11.5651	11.5278
50,000	77.5042	53.4573	50.5842	55.6155
100,000	103.5246	91.2837	92.4166	91.2974

Table 2. Time(seconds) taken to retrieve number of key-values of different sizes

We see that for our implementation, it performs better if the key-value sizes are around 16 bytes or more. This is due to the way page size, bucket size, and size and number of items

in the bucket are configured. Also, different results are seen when the bucket splitting threshold is changed. For now, the buckets split at a threshold of 80 percent capacity.

We also tested our implementation with similar implementations in JavaScript. Note that these comparisons are keeping similar key-values sizes and with 2 initial buckets. Below table shows the comparison of insertion and retrieval times for different numbers, from the hash-table between Rust implementation and JavaScript implementation.

Number of Insertions	**Time (seconds) – Rust**	**Time (seconds) – JS**
10,000	19.5812	45.0361
20,000	40.5785	101.1892
50,000	135.2164	296.7893
100,000	294.1901	687.6312

Table 3. Time(seconds) taken to insert key-values in Rust vs JavaScript implementations

As we will cover further in the future work chapter, this implementation of linear hash-table needs improvement in run time. Major improvement will come from increasing the number of initial buckets. In our current implementation, our linear hash-table starts form initial two buckets. This leads to high number of bucket splits and hashes which is taxing for the performance. Below table shows the improvement in time achieved in our implementation by increasing the initial number of buckets.

Number of Insertions	**Time (seconds) – (2 buckets)**	**Time (seconds) – (256 buckets)**	**Time (seconds) – (1024 buckets)**
10,000	19.5812	6.1263	6.0912
20,000	40.5785	14.3429	15.2482
50,000	135.2164	43.1485	41.6578
100,000	294.1901	195.3374	156.7210

Table 4. Time(seconds) taken to insert key-values in Rust with different number of initial buckets

Another important performance parameter apart from the linear hash-table is the warc parser utility. This is important in terms of getting the data in linear hash-table suitable format, using the packed table tools utility, from the gzipped warc files. Below table shows the performance data showing the time taken by the utility to read a number of records.

Number of records parsed	**Time (seconds) - Rust**
10,000	71.5156
20,000	148.0942
50,000	432.8688

Table 5. Time(seconds) taken to parse records from compressed warc file

VI. CHALLENGES

We faced a lot of challenges while implementing the application and hence, learnt a lot more in overcoming them. Some of them are mentioned in below sub-sections.

6.1 Limited Rust support on the Web

The most prominent obstacle that we had to face was the limited support that is there on the web for implementations in Rust language. Since it is a newer language, less people have expertise in this language and less contributions are there in comparison to older languages. Implementing the individual components in the first half of the project duration proceeded in a relatively smooth fashion since they find their use at other places as well, like linear hashing, WARC file reading and writing, etc. Problem of less support was also seen in terms of the knowledge available on the web regarding making custom database drivers and query engines. Support could be found for well-established databases like PostgreSQL, MySQL, etc. but none could be found for custom databases. Implementation of libraries for basic functionalities like WARC reading and writing, CDX reading and writing took more time since they were never done before. This posed a lot of learning opportunity as well.

6.2 Strongly typed vs Loosely typed languages

During the integration part of the project, we faced many hurdles considering that we were porting some of the code from PHP which is a loosely typed language. Rust being strictly typed language doesn't allow many of the default arguments and variables without strict definition of their type. This took more time in achieving the same functionality which was being achieved in the older code.

VII. FUTURE WORK

There are multiple things which need to be done in this application so as to make it production ready. Few of the important ones are mentioned in following sub-sections.

7.1 Distributed Document Store using Consistent Hashing

Consistent Hashing is a distributed hashing system that assigns each server or item in a distributed hash table a place on an abstract circle, or hash ring, regardless of the number of servers or objects in the table. This permits servers and objects to grow without harming the system's overall performance. It a employs hash function on keys to determine their distance from nodes and assigns the key-value pair to the nearest node to that key.

In our implementation, The hash output range was mapped onto the edge of a circle. That is, the smallest possible hash value, zero, corresponds to a 0 degree angle, the largest possible hash value (let's say INT MAX) corresponds to a 360 degree angle, and all other hash values linearly fit somewhere in between. So, we could take a key, compute its hash using the xxhash function, and find out where it lies on the circle's edge. An example could look like this where Kate, John, Jane, Steve, and Bill are keys while A, B, and C are server nodes:

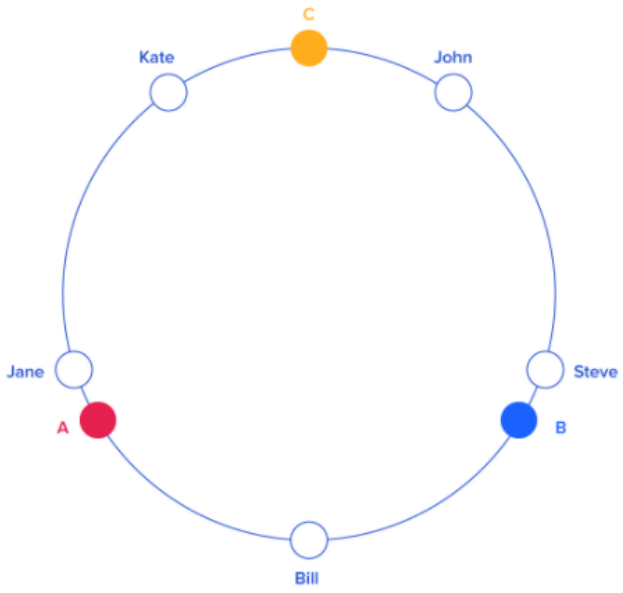

Figure 10. Consistent hashing – Server with keys unassigned

Taking a mechanism where the hash function assigns a key-value to the server node nearest to it on the circle, the allotment could look like the image below.

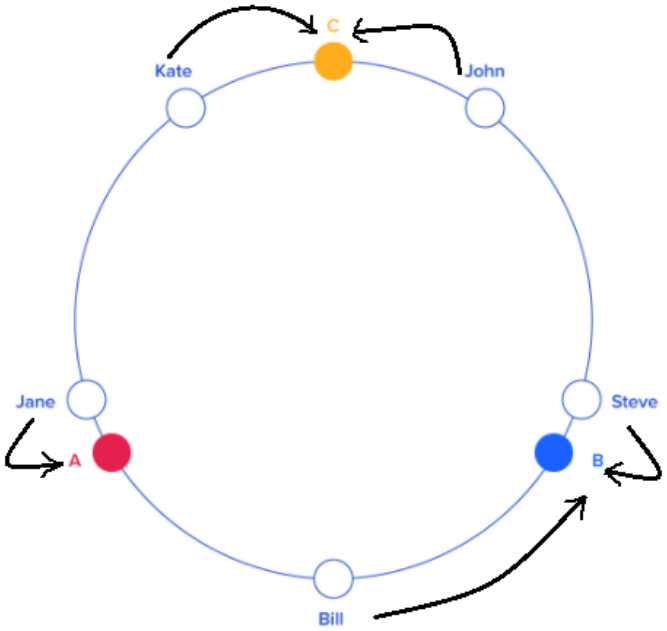

Figure 11. Consistent hashing – Server with keys assigned

When a server is deleted (say, due to a failure), only the keys from that server are migrated in consistent hashing. If server S3 is removed, all keys from that server will be moved to server S1, but keys from servers S1 and S2 will not be moved. However, when server S3 is deleted, the keys from S3 are not dispersed evenly among the surviving servers S1 and S2. They were exclusively assigned to server S1, causing server S1 to become overburdened. When a new server is added, the same thing happens. When a server is added or withdrawn, only the K/N number of keys must be remapped in most cases. The number of keys is K, while the number of servers is N. (to be specific, maximum of the initial and final number of servers).

To create a high-performance and robust document store, we will be using consistent hashing to distribute copies of documents to multiple server nodes.

7.2 Improving features of Linear Hash-table

As mentioned earlier, there is a scope of improvement in this based on future requirements. One of them is that the key-value store should be able to allow flexible value sizes. Current implementation supports only fixed size of the value. Another improvement would be to better Least Recently Used (LRU) cache mechanism instead of FIFO to be able to return the values more efficiently. Another feature that would be useful is to be able to delete the records based on the key.

7.3 Allowing custom indexing

Another better improvement in the current application would be to allow multiple kind of indexing on the data in the linear hash-table. This would result in even faster gets and faster returning of relevant documents to the user.

7.4 Allow admin to modify data

Another interesting extension of the functionalities would be to enable an admin user to run mutation queries on the data. GraphQL supports this functionality and it also allows custom schema which can be fed in initially and used later. Dynamic schema for data retrieval can also be created.

7.5 Performance tests

To actually be able to use this application on a production level, it is required that the performance be tested against other implementations, be it in other languages, or using other kinds of hashing mechanisms.

7.6 Flexible length key value pairs

Current implementation of linear hash-table requires the keys and values to be of same sizes. This value is required while initializing the hash-table so this cannot be changed later. For extended use of this application, flexible key value sizes can increase space efficiency of the application. Further improvement could be to make the sizes of keys and values independent of each other. We should only use number of bits which are required and none extra.

VIII. CONCLUSION

Databases are very important when it comes to the performance of a search engine. Along with proper algorithms, the internal implementation of the database system is also very important. Over the course of this project we worked on developing the ingredients, in the form of individual working modules, for our aim of building a high-performance document data store for applications which focus on data retrieval. We worked on developing a single node document query server using GraphQL, implemented linear hash table which speeds up the process of insertion and deletion by up to 25% keeping in mind the discussions we had above, WARC and CDX file reader-writer with at par speeds with existing Python implementations, and building indexes on WARC records. All of this, combined, make our application a promising choice for systems which require high amount of data reading and querying while not sacrificing on writes as much.

For production level viability, we will require this implementation to support running on multiple nodes to ensure high availability and fault tolerance. This will allow it to be usable for distributed and highly available systems. As mentioned in chapter VII, there are multiple improvements that can be done in the current application. We will deploy the application and run performance tests to compare the times with current implementation. Based on those results, we will further develop our implementation to improve the performance.

REFERENCES

[1] https://www.visualcapitalist.com/what-happens-in-an-internet-minute-in-2019/

[2] https://stackoverflow.blog/2020/01/20/what-is-Rust-and-why-is-it-so-popular/.

[3] https://crates.io/crates/unqlite

[4] https://docs.rs/libflate/1.1.0/libflate/

[5] https://samrat.me/posts/2017-11-04-kvstore-linear-hashing

[6] https://Rust-lang-nursery.github.io/Rust-cookbook/file/read-write.html

[7] https://archive.org/download/CC-MAIN-2021-04-1610703512342.19-0022

[8] https://commoncrawl.org/2021/03/february-march-2021-crawl-archive-now-available/

[9] https://iipc.github.io/warc-specifications/specifications/cdx-format/cdx-2015/

[10] https://www.toptal.com/big-data/consistent-hashing

[11] https://github.com/mattnenterprise/Rust-hash-ring/tree/master/examples