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

CS297

By

Ishaan Aggarwal

May 2021

ABSTRACT

Databases are a core part of any application which requires persistence of data. A lot of applications which have become an indispensable part of lives of people, like search engines, banking systems, social media apps, etc., cannot operate without databases. Due to such heavy involvement of databases, the performance of these applications is directly proportional to how fast their database operations are. There are different kinds of databases for different requirements from applications. Search engines require high data read speeds for the most part as data updating is rare.

The aim of this project is to implement a high-performance document store for the open-source search engine Yioop!. We are using Rust to make a document store which is fast, robust, and memory efficient. This semester, we focused on implementing the modules which will be required to achieve our aim. These include - a server which can return a document based on the key provided in the request, implementing linear and consistent hashing, and developing a reader-writer for WebArchive (.warc extension) files.

Index Terms: **High Performance Document Store, Linear Hashing, Consistent Hashing, WebArchive I/O, Rust**

TABLE OF CONTENTS

1. INTRODUCTION

In recent times, the volume of data that is being produced and consumed daily, has increased aggressively. According to a recent survey by Visual Capitalist [1], in one internet minute, roughly 4 million Google searches happen, around 700,000 hours' worth of videos are watched on Netflix, around 2 million snaps are created on Snapchat, and 4.5 million videos are uploaded and viewed on YouTube. 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 aim for creating a high-performance database system for our application, Yioop! search engine.

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. According to worldwidewebsize.com, there are at least 5.27 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.

Yioop! is a GPLv3, open source, PHP search engine software. It provides many features as done by larger search portals like, search results, media services, social groups, blogs, wikis, web site development, and monetization via ads. 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. This project aims at building a high-performance document store for Yioop!. This semester, the focus was on

implementing the individual modules which will be required in achieving our aim at the end of the project.

We chose Rust for this implementation. It lets developer decide whether they want to store data on the stack or on the heap and determines at compile time when memory is no longer needed and can be cleaned up. This allows efficient usage of memory as well as more performant memory access [2]. The remaining document is organized in four sections. In Section 2, we talk about how we implemented a single-node document query server. Section 3 explains the work done in implementing linear hashing. Section 4 elaborates on the module which can read and write web archive files (.warc files). Section 5 describes the implementation of consistent hashing.

2. DELIVERABLE-1: A SINGLE NODE DOCUMENT QUERY SERVER

The aim of this deliverable was to get a hands-on with Rust by starting with the implementation of a simple single node server which can act as a document key-value store. This will be utilized in future as ultimately, we intend to create an efficient, robust, and high-performance document store.

The code utilizes a Rust library (called as Rust crate) - unqlite. UnQLite [3] is a software library which implements a self-contained, serverless, zero-configuration, transactional NoSQL database engine. UnQLite is a document store database like MongoDB, Redis, CouchDB etc. as well as a standard key-value store like BerkeleyDB, LevelDB, etc. It is an embedded NoSQL (key-value store and document-store) database engine. Using this, we create a key-value store on disk and try storing some key-value pairs. The code then iterates over those stored values and based on some comparison it deletes the entries. At the end, it returns the remaining pairs in the store. This implementation shows the capability of storing/deleting/modifying the entries in the key-value store which will be handy for the ultimate aim of the project of storing warc files and reading them as required.

3. DELIVERABLE-2: LINEAR HASHING IMPLEMENTATION

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. The purpose of this deliverable is to explore the implementation of dynamic hashing scheme known as Linear Hashing. This will help in extending the simple key value store developed in the previous deliverable and lead to development of consistent hashing, as explained in Section 5.

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.

In our implementation, we first hash the key, and take however many bits the hashtable 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– eg. 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. We implemented the code in the form of a library which can be used directly. The tests included inserting one million 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 8 bytes took around 25 seconds on an average and the retrieval of the values took around 10 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. DELIVERABLE-3: WEB ARCHIVE FILE READER-WRITER

The web archive (.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
```

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'.

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. 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
```

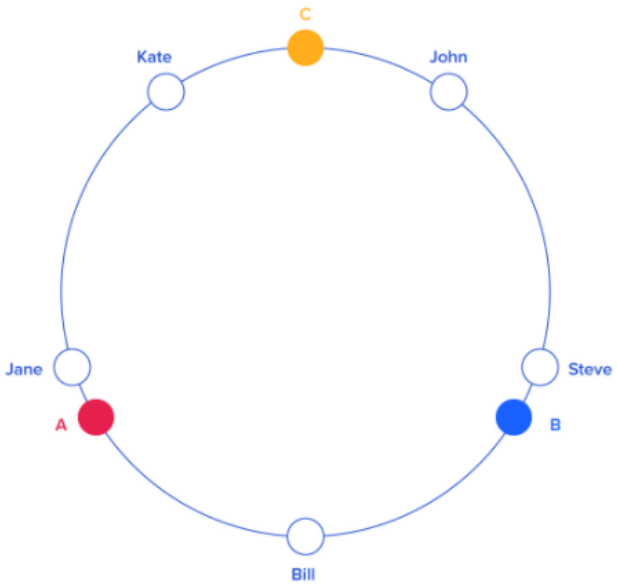
Example of a CDX file

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.

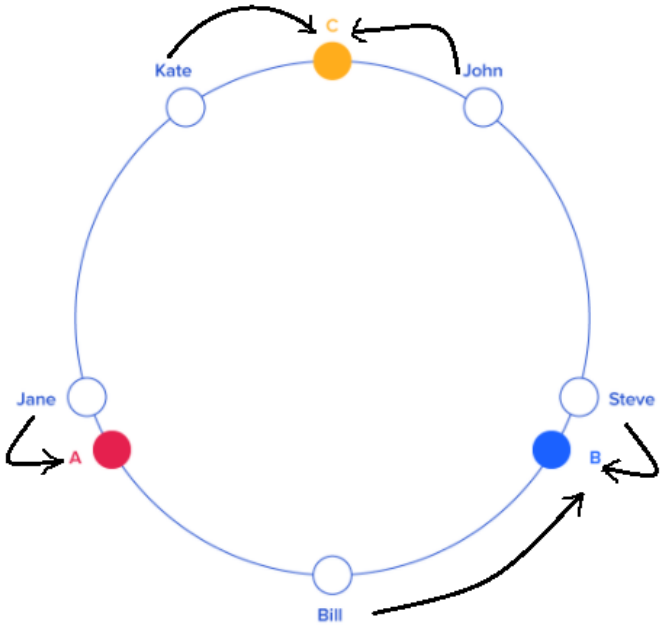
5. DELIVERABLE-4: CONSISTENT HASHING IMPLEMENTATION

Consistent Hashing is a distributed hashing scheme that operates independently of the number of servers or objects in a distributed hash table by assigning them a position on an abstract circle, or hash ring. This allows servers and objects to scale without affecting the overall system. It 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, we mapped the hash output range onto the edge of a circle. That means that the minimum possible hash value, zero, would correspond to an angle of zero, the maximum possible value (let us say INT_MAX) would correspond to an angle of 360 degrees, and all other hash values would 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:



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.



In consistent hashing, when a server is removed (let us say due to some failure), then only the keys from that server are relocated. For example, if server S3 is removed then, all keys from server S3 will be moved to server S1 but keys stored on server S1 and S2 are not relocated. But there is one problem when server S3 is removed then keys from S3 are not equally distributed among remaining servers S1 and S2. They were only assigned to server S1 which will increase the load on server S1. Similar thing happens when a new server is added. In general, only the K/N number of keys are needed to be re-mapped when a server is added or removed. K is the number of keys and N is the number of servers (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.

6. 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. This semester we worked on developing the ingredients, in the form of individual working modules, for our aim of building a high performance data store for the open source search engine, Yioop!. We worked on developing a single node document query server, implemented linear hashing, WARC and CDX file reader-writer, and explored consistent hashing.

In the second semester of the project, we will focus on understanding the current implementation of document store in Yioop!. This is currently written in PHP. We will require to migrate that implementation to Rust and utilize the modules implemented in this semester to enhance the performance of the search engine by reducing query times. We will require this implementation to support running on multiple nodes to ensure high availability and fault tolerance. 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